2009

Development and evaluation of Formula Editor (a tool-based approach to enhance reusability in software product line model checking) on SAFER case study

Sandeep Krishnan
Iowa State University

Development and evaluation of Formula Editor (a tool-based approach to enhance reusability in software product line model checking) on SAFER case study

by

Sandeep Krishnan

A thesis submitted to the graduate faculty

in partial fulfillment of requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Robyn R. Lutz, Major Professor
Vasant Honavar
Samik Basu

Iowa State University

Ames, Iowa

2009

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my sincere thanks to those who have helped me with various aspects of conducting this research and the writing of this thesis. First and foremost, I would like to thank Dr. Robyn R. Lutz for her outstanding guidance, patience and support throughout this research and in the writing of this thesis. Her constant motivation and creative insights have highly encouraged me in conducting this research. I would also like to thank her for providing me with a favorable environment for conducting this research.

This research has been possible because of the generous support of many people. Ben Di Vito provided us with the SMV code and CTL specifications for the SAFER case study. The CMU-SMV tool was provided by Carnegie Mellon University. Our lab's collaboration with Avaya Labs Research helped us to think of the research problems and solutions in an industrial setting. Particularly, I would like to thank Dr. David Weiss, Dr. Birgit Geppert, and Dr. Frank Rößler for their support in this research. I would like to thank National Science Foundation (grant 0541163), the Department of Computer Science, and Iowa State University for their financial support in this research.

I would like to thank my committee members Dr. Samik Basu and Dr. Vasant Honavar for their encouragement and help. The course on formal methods offered by Dr. Basu provided a strong base for conducting this research and developing the software. I would like to thank all the other members of the Laboratory of Software Safety: Janet Liu, Hongyu Sun, Jonathan Schroeder, Yogesh Nadkarni, Sudhindra Nagaraj Tirupati.

Their constructive criticism, timely suggestions and moral support has helped me immensely to remain enthusiastic during this research.

I would also like to thank Cindy Marquardt, Maria-Nera Davis, Jason Chen, and Jim Schlosser for their administrative support. I would like to offer my special thanks to Linda Dutton who has helped me in managing all the paperwork needed during my stay as a graduate student. Her enthusiasm and smile have always brightened my day.

I am extremely grateful to Dr. Kasthurirangan Gopalakrishnan, Dr. Siddhartha Khaitan, Ankit Agrawal, Abhisek Mudgal and Sparsh Mittal, Venkat Krishnan, Siddharth Jain, Amit Pande, Ganesh Ram Santhanam, Sivakumar Swaminathan, Chetan Hazaree, and Tanay Kishorewani for their unwavering support and encouragement which helped me to maintain my focus and enthusiasm in this research. I am very grateful to my parents Mr. M. S. Krishnan and Mrs. Jayalakshmy Krishnan and my brother Mr. Satish Krishnan for their love and moral support throughout my studies.

ABSTRACT

Although model checking is extensively used for verification of single software systems, currently there is insufficient support for model checking in product lines. The presence of commonalities within the different products in the product line requires that the properties and the corresponding specifications for these properties be verified for every product in the product line. Specification and management of properties for every product in a product line can incur high overhead and make the task of model checking very difficult. It is hence essential to exploit the presence of commonalities to our advantage by providing reusability in model checking of product lines. Since different products in the product line need to be checked for same or similar properties, reuse of properties specified for one product for other products within a product line will significantly reduce the overall property specification and verification time.

FormulaEditor is a property specification and management tool for enhancing the reusability of model checking of software product lines. The core of the technique is a product line-oriented user interface to guide users in generating, selecting, managing, and reusing useful product line properties, and patterns of properties for model checking. The previous version of the FormulaEditor tool supports Cadence SMV models, but not the typical CMU-SMV models. This work extends the FormulaEditor tool to allow verification of models written in CMU-SMV. The advantage of providing support to another model checker is twofold: first, it enhances the tool's capability to check design specifications written in different models; and second, it allows users to specify the same design in different modeling languages to detect problems.

CHAPTER 1. INTRODUCTION

It is becoming increasingly important to manage related products as members of a product line. Product lines provide successful reuse of assets and resources within an organization. A software product line is "a set of software-intensive systems sharing a common, managed set of features that satisfy the particular needs of a specific market segment or mission and that are developed from a common set of core assets in a prescribed way" [17]. In a product line [42], the common requirements which are to be met by all the products are called *commonalities*. The set of allowable differences amongst the products are called their *variations*. A wide variety of companies have decreased their software development and maintenance costs and simultaneously increased the quality of their products by the use of software product lines. The use of software product lines is also increasing in the field of safety-critical systems created in organizations such as NASA, GE, Avaya, etc.

Undoubtedly, the development of product lines also has led to the need for verification of the different products in the product line to ensure that the requirements for the product line are satisfied by the individual products. Verification that a new system built in a product line satisfies common properties takes many forms including inspection, state-based simulation and testing [42], [1], [31]. However, these techniques do not provide the necessary assurance needed for products in certain safety critical domains. Model checking is a rigorous verification technique that enhances the quality of software systems [13], e.g., by identifying flaws that would not have been caught otherwise ([24], [28]). Models for the software system are written in verification

languages like SMV [8], [33], SPIN [26], etc. and properties are specified for the model. The model checkers verify these properties against the provided model.

Although model checking is extensively used for verification of single software systems, currently there is insufficient support for model checking in product lines, most specifically, for property specification and management [32]. The presence of commonalities among different products in a product line requires that such commonalities be verified for every product in the product line. Specification and management of property for every product in a product line can incur high overhead and make the task of model checking very difficult. It is hence essential to exploit the presence of commonalities to our advantage by providing reusability in model checking of product lines. Since different products in the product line need to be checked for same or similar properties, specifying the properties for one product and reusing them for other products within the product line will significantly reduce the overall property specification and verification time. The difficulty with reuse across a produce line is that the variations among the products can complicate the implementation and verification of the properties.

Product line verification, like product line engineering in general, tries to reuse whatever is common across the product line to reduce the cost and increase the quality of each new product [32]. Thus product line verification urges reuse, enabled by the presence of commonalities and simultaneously provides very careful management of that reuse, demanded by the presence of variations among the products.

FormulaEditor is a property specification and management tool for enhancing the reusability of model checking of software product lines. It was originally developed by Jing Liu in work initiated while on an internship at Avaya Research Labs under the guidance of Birgit Geppert, Frank Rossler and David Weiss [32]. [32], the technical report for FormulaEditor, describes FormulaEditor as follows:

"The core of the technique is a product line-oriented user interface to guide users in generating, selecting, managing, and reusing useful product line properties, and patterns of properties for model checking. The tool also associates the properties with the requirements, models and verification results of each product in the product line so that any changes can be readily traced and the properties updated accordingly."

The previous version of the FormulaEditor tool supports Cadence SMV models, but not the typical CMU-SMV models. Cadence SMV [10] is an extension of CMU-SMV [34]. Cadence SMV has more expressive mode description language and it also supports synthesizable verilog as a modeling language, allowing RTL designs to be verified [9]. In addition, Cadence SMV allows several forms of specification, including the temporal logics CTL and LTL, finite automata, embedded assertions, and refinement specifications. The previous version of FormulaEditor with Cadence SMV was tested on two product lines, the first being a family of communication protocols that resulted from an Avaya refactoring project and secondly on a cardiac pacemaker product line. However, the tool's provision to support a single model checker forced users to specify Cadence-SMV models. By providing users with the flexibility of using multiple model checkers, the effectiveness of the FormulaEditor would be increased.

Contributions:

1. The first key contribution of this thesis is thus to extend the FormulaEditor tool to allow verification of models written in CMU-SMV. The need to extend the FormulaEditor to include CMU SMV as the second model checker came from the effort to model check legacy systems and their extensions. A Legacy system [3] becomes especially important when the cost incurred in redesigning or replacing the system is large. An example of legacy systems is the prevalent use of the NASA technologies developed two or three decades ago. Such technologies have already completed expensive integration and certification requirements for use and any new technology would have to go through the entire process which would require extensive tests. However, if any extension to an existing system is to be made using the legacy code of the system, verification of the extended system is necessary to ensure that the changes are safe. Cadence SMV being a successor of CMU-SMV, it would be preferable to model check new systems using Cadence SMV. However, system developers who are comfortable developing models in CMU-SMV may prefer to use CMU-SMV if the features of CMU-SMV prove to be sufficient.

Our motivation to extend FormulaEditor to include model checking using CMU-SMV was thus twofold: first, to provide the benefit of model checking legacy system extensions which are modeled in CMU-SMV; and second, to allow model checking of newer systems modeled in CMU-SMV. The work on extending FormulaEditor to support CMU-SMV was started as a part of the previous version. However, the implementation was not complete and several errors also had to be corrected in order to ensure that the new version functioned correctly.

2. The second key contribution of this thesis is to provide an initial evaluation of the use of the expanded FormulaEditor in a product line setting by using it to specify several properties for a simplified product of manned maneuvering units named SAFER. The SAFER product is then evolved into a product line by introducing variations. This product line is then used as a test-bed for testing the improvements made to FormulaEditor. The product line models are specified in CMU-SMV, and FormulaEditor is used to specify and verify the commonalities and variations for this product line.

Tests on the improvements to FormulaEditor show important advantages of FormulaEditor. The test results show reduction in specification and verification time by the use of FormulaEditor. Reuse of similar patterns and dynamic instantiation of properties provide flexibility in property specification. FormulaEditor features provide ease of property specification and reuse of properties both within a single product and among multiple products in the product line. Advantages such as flaw detection in the underlying model by analysis of the generated false positives and false negatives, and ability to adapt to evolution of product lines are provided by FormulaEditor.

The rest of this thesis is organized as follows. Chapter 2 provides an evaluation of the FormulaEditor tool. Specifically, it describes the problems existing in the previous version of FormulaEditor, explains the solutions implemented to address these issues, and provides an evaluation of FormulaEditor on the SAFER (Simplified Aid for EVA Rescue) case study [22] as a single product. In [2] Ben Di Vito explains the application of PVS theorem proving technique to verify the properties of SAFER. In our evaluation of FormulaEditor we demonstrate the effectiveness of FormulaEditor in property verification for SAFER. We corroborate our claim that FormulaEditor provides better

ease and flexibility for specification and verification of SAFER's properties than the approach taken by Ben Di Vito. We evaluate and explain how the individual property specification for each model needed in the theorem proving technique is tedious as compared to the reusable property specification technique provided by FormulaEditor. Chapter 3 describes the potential evolution of a product line for SAFER which is used as the test-bed for testing FormulaEditor. In both Chapter 2 and Chapter 3, we focus on the reusability aspect of FormulaEditor and look at it from two aspects: reusability within a product and reusability within the product line. Chapter 4 explains related work. Chapter 5 describes future work and Chapter 6 concludes the thesis.

CHAPTER 2. RELATED WORK

In this section, we discuss the literature survey for the topics of software product lines, model- checking and the use of model checking to verify software product lines.

2.1 Software product lines

Significant work has been carried out on the topic of software product lines. Research in the field of product lines was motivated by the visionary success of CelsiusTech Systems AB, a long-time European defense contractor in the 1980s. The case study is explained in [35]. CelsiusTech was faced with the dilemma of building two large command and control systems, each larger than anything that the company had attempted before, and it had barely enough resources to build one. CelciusTech laid the foundation for the massive use of product lines in industries. Companies such as Boeing [18], Nokia [25], Philips [44], Hewlett Packard [41] and many others have used the concept of product lines to build their products in an efficient manner. Product lines enable the reuse of the common requirements among the products. Reuse of the underlying architecture, requirements and the algorithms and safety analysis reduce the overall production time while simultaneously providing better quality for the products. Studies suggest that product line engineering can reduce the overall development and production time and the production cost while improving the quality by a factor of 10 or more [38].

Extensive work to understand the features of product lines, to formalize them, to develop efficient methods to utilize the commonalities in product lines, and on product line engineering have been carried out. Several textbooks have been written on the

subject of software product lines. Some of the prominent ones are [17], [23], [35], and [42]. Weiss and Lai [42] describe an approach for developing product families called Family-Oriented Abstraction, Specification and Translation (FAST) approach. This approach is based on investing resources in the early design of a set of systems to identify their commonalities and variabilities. The FAST approach advocates this strategy because it claims that the high investments of resources in the early design stages are amortized over the set of product line members that are produced. The FAST approach partitions the design and development of a product into two phases: domain engineering and application engineering. The goal of the domain engineering phase is to list the product line requirements, define its design and architecture and identify other software engineering assets that pertain to the entire product line [42]. This process requires domain knowledge and skilled experts [17], [35]. This is an investment phase which allows the practitioners to quickly realize a variety of products within the product line for a competitive advantage. The goal of the application engineering phase is to build the product line member(s) from the product line requirements identified during the domain engineering phase [42]. The new product is built by selecting values for the parameters of variation and defining the constraints among the selected variabilities.

2.2 Model Checking

Extensive work on model checking has been conducted to date. Temporal-logic model checking ([12], [16], [30], [36], [39]) is a method for verifying whether a specification is satisfied by a finite-state program. Clarke et al. ([14]) presented a model checking algorithm for propositional branching-time temporal logic CTL. The algorithm was used to verify a simple version of the alternating bit protocol with 20 states. Since

then, the size of the programs that have been verified by this means has increased dramatically. Special programming languages [26], [33] have been developed to check examples with several thousand states. The use of binary decision diagrams (BDDs) ([5]) led to the ability to verify programs of greater size. Representing transition relations implicitly using BDDs made it possible to verify models that would have required 10^{20} states with the original version of the algorithm [6]. Refinements of the BDD-based techniques [7] pushed the state count up over 10^{100} states.

2.3 Model checking Software product lines

Existing work has indicated the possibility of successfully conducting model checking for software product lines. Kishi and Noda [40] proposed an approach that models product line variations in UML models and then translated them into SPIN models. Li, Krishnamurthi, and Fisler [29] have exploited compositional verification in the product line context by automatically checking interfaces of separate features using the labeling algorithm in CTL model checking. Robby, Dywer, and Hatcliff [37] have constructed Bogor, an extensible model checking framework that can be customized to different application domains, e.g., to be used as a back-end model checker for Cadena – an integrated environment for building and modeling CORBA Component Model systems – that can be used to develop model-driven component-based product lines.

Techniques have been developed to ease the difficulty of translating informal (natural language) specifications into formal ones (e.g., temporal logic formulas [27]), such as the Property Specification Patterns [19]. Work on reuse of specification patterns has been conducted in recent years. Blazy, Gervais, and Laleau [4] describe an approach for defining and reusing specification patterns in B language. Farkash, et.al [21] describe

writing reusable property specifications and the circumstances in which such specifications can be reused for the PSL language. However, in these techniques, to the best of our knowledge, the issue of management of property specification at the product line scope remains unaddressed. Furthermore, they do not treat property specification reuse at the implementation level. Liu, et.al [32] address both these concerns by implementing the reusability feature in property patterns for a family of products in a product line. Their application tests the reuse of property patterns written in the temporal languages LTL and CTL for systems modeled in Cadence SMV modeling language. Our work extends this work by providing additional flexibility to model check product lines in multiple modeling languages.

CHAPTER 3. FORMULAEDITOR ON *SAFER* PRODUCT

In this chapter, we describe our work to evaluate FormulaEditor on the SAFER case study. The evaluation describes the importance of FormulaEditor in specification and verification of properties for the single product SAFER. A single requirement for a product can result in many properties, i.e., the mapping from requirements to properties can be many to one. Such properties can have the same skeleton with the atoms (variables) being instantiated to different values. These properties can also become complicated as we will show by examples later in this section. FormulaEditor provides the convenience to reuse the properties and reduce the specification and verification time by providing *property patterns*. The goal of FormulaEditor is to provide this reusability. This section first explains the functioning of FormulaEditor and the improvements to the previous version of FormulaEditor. Further, the features of reusability, flaw detection and other advantages of FormulaEditor when evaluated on the SAFER case study are described.

3.1 Improvements to previous FormulaEditor version

3.1.1 Background of FormulaEditor

Prior to explaining the improvements to FormulaEditor's previous version, we give a brief description of FormulaEditor. As described earlier, FormulaEditor is a tool designed to increase reusability in product- line model checking. In this section, we give a brief description of the architecture of FormulaEditor, the different components of FormulaEditor and their functionality. The architecture of the FormulaEditor is shown in Figure **1**. FormulaEditor takes as input the product/product line model written in an SMV

language and the property to be verified. It provides features such as property to requirement mapping, property pattern reuse, dynamic atom selection and flaw detection in the property or model. The inputs are given to the underlying model checker (Cadence SMV or CMU SMV). The model checker verifies the property against the model and returns the verification results. FormulaEditor combines all the above features and the results and displays the results to the user.

FormulaEditor runs one of the two model checkers: Cadence SMV and CMU-SMV in the background for the verification of properties. The speed of verification provided by FormulaEditor is hence dependent on the speed of the underlying model checker. The advantage that the FormulaEditor provides is in the ease of specification and verification of properties.

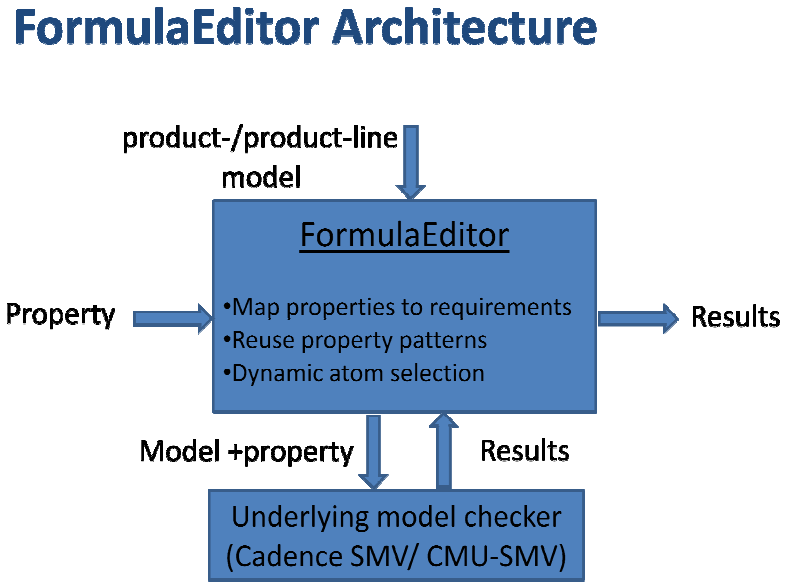

Figure 1: FormulaEditor Architecture

FormulaEditor has the following 4 components or panels [32].

i. *Model Panel*- The model panel allows the user to specify all the information associated with the model such as the output directory for the files generated during model checking, the common pattern location to allow reuse of property patterns, the model checker type, the location of the model checker and the location of the model file.

ii. *Properties Panel*- Properties panel shows all the properties specified for that model. Information associated with a specified property includes: the temporal logic formula itself, its type, description, truth value, current status and the category to which it belongs to.

iii. *Atom Selection Panel*- FormulaEditor recognizes the variable declarations in the model as atoms. The states of each variable are atomic formulas (which we call atoms) that can be used individually or combined together with Boolean operators or temporal operators to assess meaningful properties of the system. The atoms can be manually selected as per requirement from the atom selection list which displays all the atoms in the model.

iv. *Property Editor Panel*- The property editor panel allows specification of properties by providing the commonly used patterns and also the user-defined common patterns for reuse. Properties can be saved after editing them. Two views, namely *text view* and *tree view*, are provided for editing a property. These two views comprise the *Syntax directed property editing* area. Manual editing is also provided in the *Free-style property editing* area. Variables selected in the atom selection list are also displayed in this panel. These atoms can be used in instantiating the common or user-defined patterns.

3.1.2 Background of Computation Tree Logic (CTL)

Linear Temporal Logic (LTL) and Computation Tree Logic (CTL) are both temporal logics that are used for model checking. Properties to be verified are written in temporal logic and are verified against the models. LTL and CTL are two such languages that provide connectives that allow us to refer to the future [27]. Both languages model time as a sequence of states extending infinitely into the future. While Cadence SMV allows specification of properties in both LTL and CTL, CMU SMV allows specification only in CTL. LTL and CTL formulae are evaluated on paths. CTL is advantageous over LTL as it allows verification of properties which assert existence of paths. A state of a system satisfies an LTL formula if all paths from that state satisfy it. Thus, LTL implicitly quantifies universally over paths. Properties which mix universal and existential path quantifies cannot in general be model checked using LTL. CTL solves this problem by allowing us to quantify explicitly over paths. However, there are many LTL properties which cannot be expressed in CTL and vice versa.

The formal Backus Naur definition of CTL [27] is

$$\phi := \bot \mid \top \mid p \mid (\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \rightarrow \phi) \mid AX\phi \mid EX\phi \mid AF\phi \mid EF\phi \mid AG\phi \mid EG\phi \mid A[\phi U \phi] \mid E[\phi U \phi]$$

The general description of the operators is as follows:

A $\stackrel{\text{def}}{=}$ along all paths

E $\stackrel{\text{def}}{=}$ along at least one path

X $\stackrel{\text{def}}{=}$ Next state

F $\stackrel{\text{def}}{=}$ Some future state

G $\stackrel{\text{def}}{=}$ All future states

U $\stackrel{\text{def}}{=}$ Until

3.1.3 Enhancements to previous version of FormulaEditor

This section now provides detailed explanation of the improvements to FormulaEditor's previous version. Previously, the tool had been tested on models which were written in Cadence SMV. Although the work to incorporate CMU SMV models was begun, sufficient testing had not been performed and many of the features were untested. The work reported here provides support to another model checker namely CMU SMV, thereby enhancing the tool's capability to check design specifications written in two different models. It also allows the users to specify the same design in different modeling languages to detect problems. This document describes the enhancements done to the FormulaEditor tool that enables it to check CMU-SMV models in addition to the existing support to Cadence SMV models.

Specifically, this section explains in sequence:

- Suggestions to improve the design of the CMU SMV model files to be verified, and
- Detection of errors in the previous version of the tool and the implemented solutions.

Updates in writing the model file:

As mentioned earlier, the previous version of the tool had not been tested for models written in the CMU-SMV language and hence the tool needed modification to solve errors relating to CMU-SMV.

Some of the problems faced while testing the tool on models written in CMU-SMV were:

- *Extracting variables from DEFINE block*- The FormulaEditor uses the model file and extracts the atoms from the model and displays in the *Property Editor panel* which can then be used by the users to create properties and patterns. However, the FormulaEditor only displays the variables which are created in the VAR block and assigned in the ASSIGN block. The tool does not display variables which are created in the DEFINE block. This is logical since the variables defined in the DEFINE block are usually internal variables which are created to reduce the state space.

  To facilitate writing specifications which make use of these internal variables defined in the DEFINE block, we need to remove these variables from the DEFINE block and define them explicitly in the VAR and ASSIGN block. For example, in the model file which is being used for testing, a variable named *all_axes_off* is defined in the DEFINE block of the MAIN module. Since this variable is very commonly used in many of the specifications which were written to test the model, these properties could be specified using the FormulaEditor by extracting the *all_axes_off* variable from the DEFINE block and creating it in the VAR and ASSIGN block.

- *Naming of variables*- An additional aspect to be taken care of when creating the model is to define variables without any delimiters such as '_' (underscore) in them. The FormulaEditor uses the delimiter '_' to finally convert the syntax-edited formula into the temporal formula. During this operation every _ is converted to '.'(dot). Hence, in order to avoid confusion for the FormulaEditor, it is better if we model the system in such a way that the variables themselves in the model do not contain '_'. We can use the naming Scheme in Java where variable names having multiple words or phrases have the first

word in lowercase and the first letter of the subsequent words in uppercase. For example, the variable all_axes_off can be defined as *allAxesOff* to avoid errors.

- *Defining Constants* - Another problem created because of switching the variables from the DEFINE block to the VAR- ASSIGN block is as follows. In SMV, it is not possible to define variables in the VAR block which take a constant value. The data types supported by SMV are Boolean, Enums and a Sub-range. Booleans can take values 0 or 1. Enums are a set of values and the variable can take any value in that set. Sub-range is a range of values that a variable can take (e.g. 0…100). Thus, if we want to define a constant value like 10 or 100, then this can be done only in the DEFINE block but not in the VAR – ASSIGN block. While specifying the properties for the model file, we required the use of these constant variables. To extract such variables which were needed in the specifications, we expressed such a variable as an enum with a single value in the set. For example, in the model used, a variable *max_ticks* is extracted from DEFINE block of *main* module and specified in the VAR-ASSIGN block as follows

max_ticks :{ 100}

This modification enabled the use of this variable max_ticks in our specifications.

- *Defining variable of type sub-range*- The simple data types in SMV are *Boolean*, *enumerated* and *subrange*. If a variable is of type *Boolean*, then the FormulaEditor creates two atoms for the corresponding variable; one atom is the variable with true value and the second is the variable with false value. For every such Boolean variable declared in the model, the Atom list created by FormulaEditor for that model contains two atoms.

For a variable of type *enumerated*, the Atom list contains as many atoms for this variable as the number of symbols in the enumerated set. Thus a variable

state: {On, Off, Standby};

will have three atoms; one for each symbol in the set.

While creating variables of subrange type, the FormulaEditor cannot specify atoms for each value in the range. For example, if

count : 0..100;

it is not possible to create atoms for every value in the range of 0 to 100. Hence the FormulaEditor gives three options for the initial and final value in the range. Thus three atoms are created for the value 0 where *count* can be equal to, greater than, or greater than equal to zero (the minimum value in the range) and three atoms for *count* equal to, less than and less than equal to 100 (the maximum value in the range). It is important to note that the FormulaEditor currently does not give the facility to specify properties which have any other value to such a sub-range variable like *count*.

Updates Performed to the FormulaEditor tool

The following section describes the extensions and improvements made to the FormulaEditor tool to enable use of CMU-SMV as the underlying model checker.

- *EVENTUAL (F) properties in CMU-SMV* – The properties such as *AF* and *EF* in CTL were not being verified in the previous version of the tool as the property specifications were not inserted in the model in the correct manner. As a result, when

verifying the property using the FormulaEditor, the CMU-SMV model checker in the background returned an error.

Update- After scanning the source code, the error was traced to incorrect input of the final property to the model file. This corresponds to line 257 in the *adjustProperty* module in *CMUSMVFileRenderer* class in the *modelChcking.cmuSMV* package. In this line, instead of 'F', 'N' was being inserted into the model and hence the model checker was not able to recognize this new character and returned the error.

Testing- The modification was tested by specifying AF and EF properties using FormulaEditor. The modification proved to be successful as both *AF* and *EF* properties were successfully verified providing the expected results. Figure 2 shows the verification results for AF and EF properties.

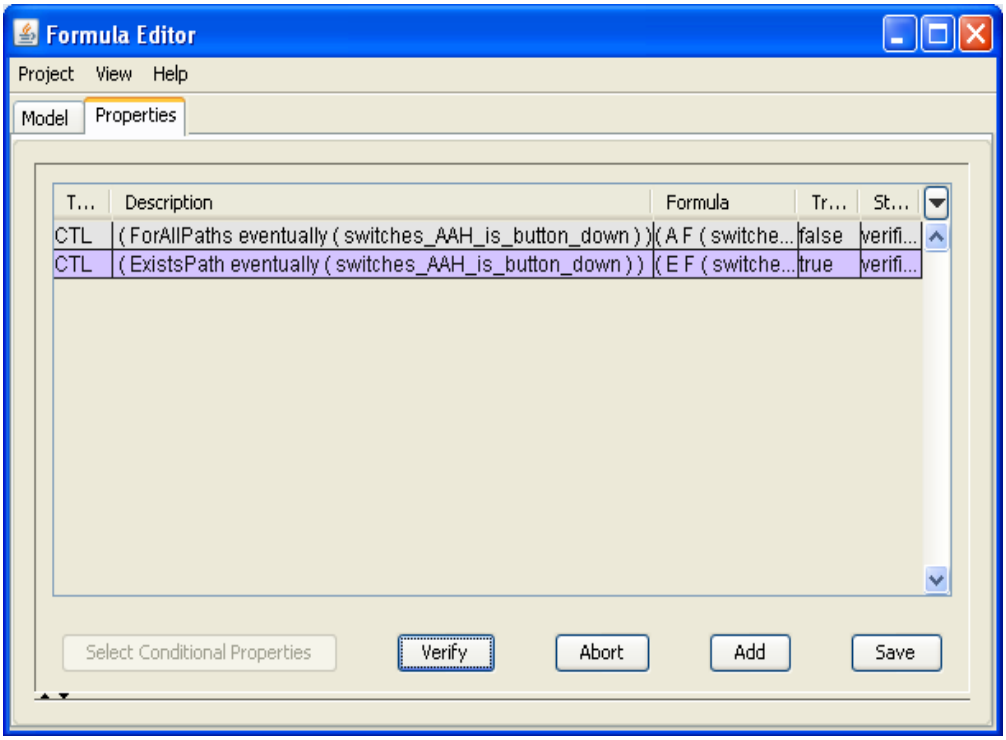

Figure 2: Verification of AF and EF properties

- *Verification of UNTIL (U) properties* - The AU and the EU i.e. the *Always UNTIL* and the *Exists UNTIL* properties could not be verified in the CMU-SMV version of the FormulaEditor in the previous version. Although, the properties were inserted into the newly created smv file, the FormulaEditor failed to verify the properties which had *UNTIL* operators. A detailed inspection of the source code revealed a syntax error while generating the *UNTIL* properties. In the FormulaEditor, the *UNTIL* properties were specified using the same syntax as the other CTL properties. But the CMU-SMV model checker uses square brackets '[ ]' for *UNTIL* properties.

Update- The *initCTLpatterns* method in *EditorViewer* class of *propertyEditor* package was modified to incorporate this change.

Testing- After making this modification to the source code, the correctness of the modification was tested using the UNTIL property. Specifications for both AU and EU properties were fed to the FormulaEditor and the results were observed. A total of 19 properties having either AU or EU connectives were verified. The tests confirmed that the modification was successful and the properties were being verified giving both true and false results as expected. The screenshots displaying the results of verification of AU and EU properties are shown in Figure 3. The first property in the figure verified that there exists a path where the current state is AAH_Off until the AAH button is in down position and along all the paths, the next state is AAH_started. The CTL representation for this property is as follows

(E [(AAHState.toggle.engage = AAHOff) U ((switches.AAH = buttonDown) & (A X (AAHState.toggle.engage = AAHStarted)))])

This is an example of one of the simple properties involving the EU connectives which we tested. The 19 properties which we tested included other, complicated properties. Similarly, properties including the AU connectives were also verified.

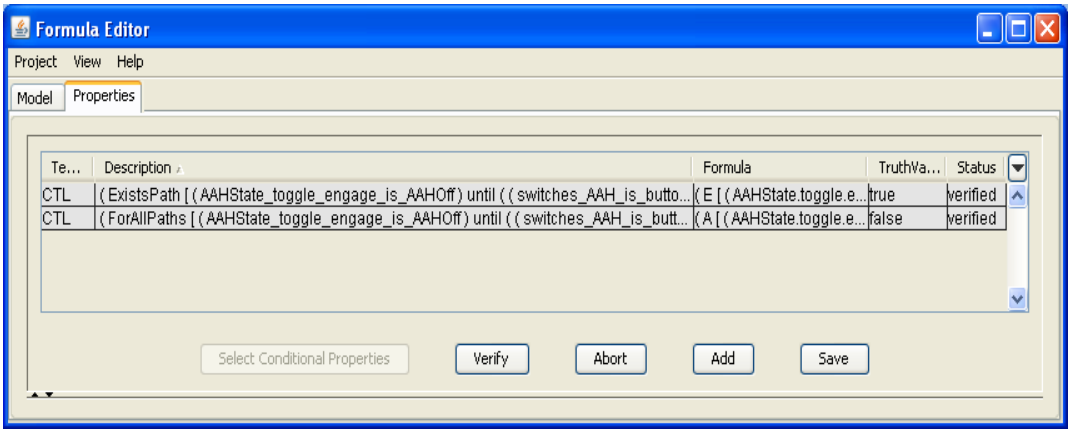

Figure 3: Verification of AU and EU properties

- *Inability to Edit properties* – The FormulaEditor provides easy access to edit the properties. It prevents incorrect modification of properties in such a way that when CTL properties are being edited, the default LTL patterns in the property editor panel are made inactive so that the user accidentally does not edit CTL properties with LTL patterns and similarly, CTL patterns are made inactive while editing LTL patterns. However when new properties were being added using the previous version of FormulaEditor with CMU-SMV, the value of the *Temporal Logic* field was incorrectly shown as LTL although CMU-SMV supports only CTL. Hence when these properties were being edited, no actions could be performed as the default CTL patterns were made inactive.

Update- The error was noted to be in the *getTemporalLogicTypeFromFormula* method in the *CMUSMVFormulaTranslator* class of the *modelChecking.cmuSMV* package. This was again noted and modified in the source code

Testing- This change enabled the CTL properties to be edited when the current version of the FormulaEditor is used with CMU-SMV as the CTL default patterns and user defined patterns were not becoming inactive anymore. Figure 2 and Figure 3 show that the temporal logic for the properties is correctly defined as CTL as compared to LTL which was inserted earlier.

- *Tree view in CMU-SMV*- For property specification, the FormulaEditor provides two views in the *Syntax Directed Property Editing* panel. These two views are named as *Text view* and *Tree view*. The *Tree view* gives a tree hierarchy representation for the ease of selection and initialization of atoms. The earlier version of FormulaEditor could not successfully represent the *Until* properties in CTL in the Tree view. Both *AU* and *EU* properties in CTL resulted in "*error*" nodes in the *Tree view*.

Update- The reason for this error was due to handling of the UNTIL properties in the same manner as OR, AND, and IMPLIES properties were handled. The difference between the UNTIL properties and the latter properties is that in CTL, UNTIL properties are always accompanied by the connective A or E. Thus, in AU and EU connectives, the A and E are inseparable from U. This condition was not taken into account and it resulted in error in the tree view. To incorporate this condition, the UNTIL properties were handled separately from the AND, OR, and IMPLIES properties. Changes were made to the *parseExp* method in the *PropertyTree* class of *PropertyEditor.propertyPanel* package to incorporate this change.

Testing- The modification was tested by specifying UNTIL properties and noting the effects in the tree view of FormulaEditor. The modifications proved successful as the tree view for UNTIL properties were not showing error nodes anymore and the editing of the properties was also possible from the tree view. The modification results are shown in Figure 4. It shows the tree view of the current FormulaEditor. The specification of an EU property is demonstrated. The figure shows the ability to use tree view to specify and edit UNTIL properties.

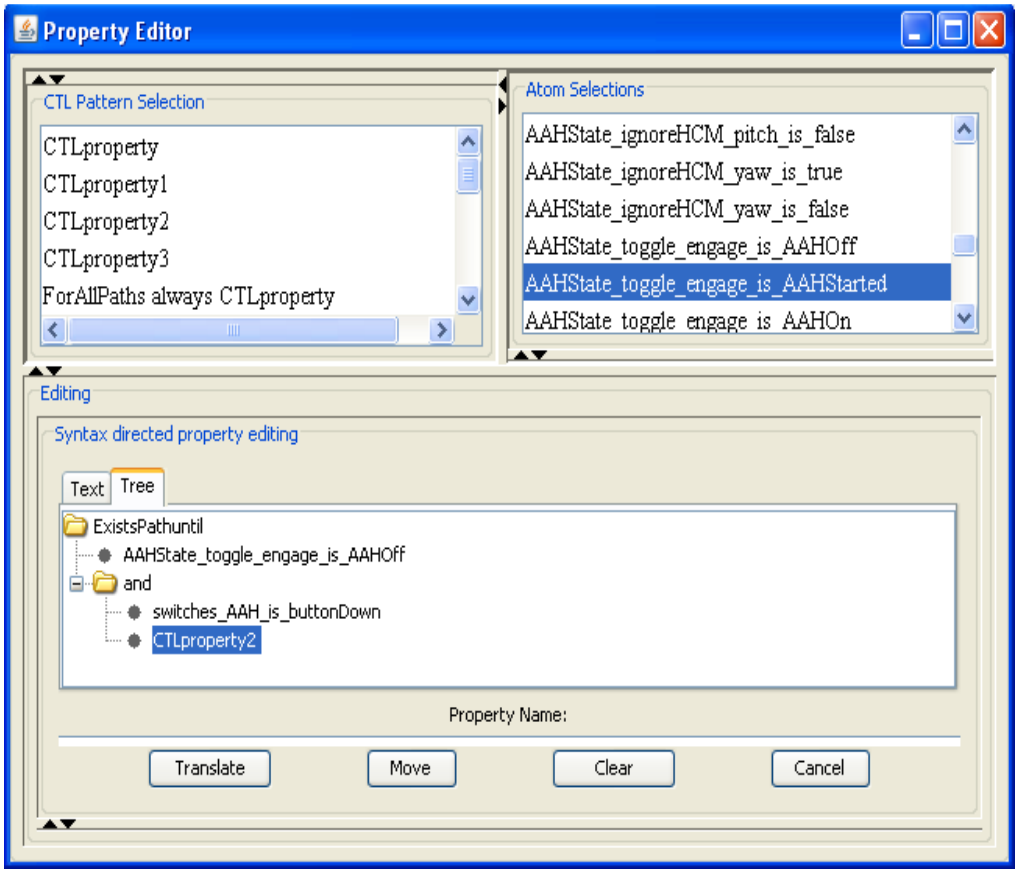

Figure 4: Tree view of FormulaEditor with CMU-SMV

- *Difficulty due to MAIN module-* Currently the FormulaEditor functions as follows. It uses the model file e.g. *model.smv,* extracts the atoms from the file and displays to the user. The user can use these atoms and the LTL and CTL common patterns to create properties. When these created properties are saved, the FormulaEditor creates a new .smv file, say *newmodel.smv,* with the same model and the newly created property inserted in the MAIN module in newmodel.smv as a specification. This newmodel.smv is given for model checking to the background model checker and the results from the model checker are displayed by the FormulaEditor.

A difficulty which was faced by the previous version of FormulaEditor was that the newly created properties were inserted after the MAIN module of the original file only when the MAIN module was followed by another module in the model. If MAIN was the last module in the model being verified, then the newly created properties were not inserted in the newmodel.smv file and it was same as the original model.smv file. Attempt to verify the inserted property resulted in a failure.

Update- The model file which was used to test the CMU-SMV implementation of FormulaEditor was from the SAFER case study. This model file was earlier used with the command-line CMU-SMV model checker to test the correctness of the properties and to demonstrate the effectiveness of FormulaEditor. As a result the model already included some properties/specifications to be verified. This model file was then used to verify additional properties specified using the FormulaEditor. Although the newly specified property was not inserted into the model file as described above, the underlying CMU-SMV model checker for FormulaEditor verified the existing properties in the model file and did not result in any failure or output any errors. Hence the error went unnoticed.

In our improvement work, we corrected the error by inserting the property into the model irrespective of whether MAIN is the last module in model or not. The *insertSMVContentToFile* method in *CMUSMVFileRenderer* class in the *modelChecking.cmuSMV* package was modified to correct this error.

Testing - An attempt to remove the existing properties in the model and then verify the properties inserted using FormulaEditor into this new model resulted in failure and the detection of this error. After performing the modification, the testing of this modification was successful as the properties were inserted into the model irrespective of the position of the MAIN module. The testing was carried out by placing the MAIN module in different locations in the model file and removing the existing properties in the model file. This model file was then used as input and new properties were specified using FormulaEditor. The verification of these properties gave the expected true/false results instead of the N/A result which was earlier displayed.

3.2 Evaluation of FormulaEditor on SAFER

In this section, we explain the results of evaluation of FormulaEditor on the SAFER case study. We first give a brief description of SAFER, the application we used to evaluate the use of FormulaEditor. We then proceed to explain the results of our evaluation.

SAFER- SAFER is a small, lightweight propulsive backpack system designed to provide self rescue capability to a NASA space crewmember separated during Extra Vehicular Activity (EVA) [22]. EVA is any activity performed by a pressure-suited crewmember in unpressurized or space environments [20]. SAFER provides six-degree-of-freedom

maneuvering control. A single hand controller is used to control SAFER operations. Propulsion is available either on demand, i.e. in response to hand controller inputs, or through an automatic attitude hold (AAH) capability. Hand controller inputs command either translations or rotations, while attitude hold is designed to bring and keep rotation rates close to zero. Figure 5 shows the Automatic Attitude Hold State Diagram described by Ben Di Vito [2]. The diagram indicates how SAFER reacts to the position of the AAH pushbutton and several other conditions. The AAH cycle begins in the state *AAHOff*. If the pushbutton is in the down position, then a transition is made to the *AAHStarted* state. The states pressedOnce and pressedTwice are used to model the deactivating of AAH by a double click of the pushbutton. Timeout is a counter which ensures that AAH is deactivated only when the pushbutton is double clicked within a period of 0.5 seconds. If the button is not double clicked within a period of 0.5 seconds, then the timeout variable ensures that SAFER returns to the state *AAHOn* after the timeout.

To illustrate the feature of reusability provided by FormulaEditor, we model the AAH state diagram in CMU-SMV and verify the properties for AAH on this model using FormulaEditor. The model is given in the Appendix. The module *buttonState* handles the switching between the six states as per the position of the AAH button. The button can be either in the up or down position. The rotational axes are modeled again using a module named *rotCommand* which provides three variables, one for each of the three rotational axes yaw, pitch and roll. The variables change values non-deterministically. The module *AAHTransition* maintains the values of the rotational axes which are active and the *timeout* variable.

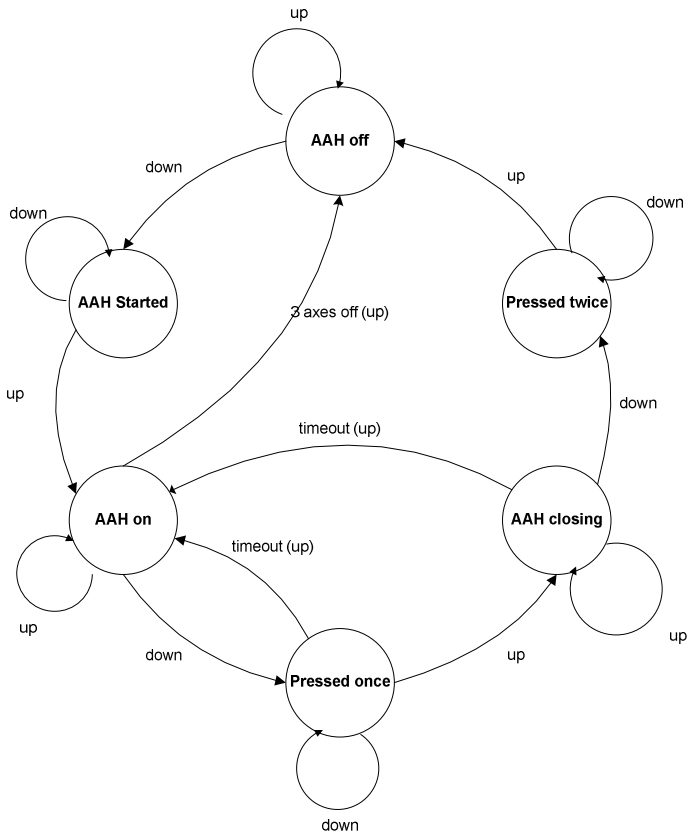

Figure 5: Automatic Attitude Hold State Diagram [2], [22]

Ben Di Vito [2] classified the SAFER properties related to the AAH feature into the following 5 categories:

1. Transition function outputs that result from specific inputs.
2. Relationships between pairs of successive states.
3. Unconditional state invariants applying to all states.
4. Hold-until invariants over sequences of states bracketed by triggering and terminating conditions.
5. Hold-until invariants concerning frames which include input, output, and previous/next states.

Ben Di Vito also provided the CTL representation of 30 properties, for the AAH state diagram in [2], [22], which are listed in the Appendix. He mapped these 30 properties to the above five classes as follows:

Property Number	Category
P1, P2 and P3	Category 1
P4 to P9	Category 2
P10 and P11	Category 3
P12 to P20	Category 4
P21 to P30	Category 5

Table 1: AAH property classification

An example of these properties is the property *on_to_off_direct* numbered as P9 in the Appendix. This property is included in the 2nd category as per Ben Di Vito's classification. The property is defined as follows.

```
( A G ( ( ( AAHState.toggle.engage = AAHOn )
        & ( A X ( AAHState.toggle.engage = AAHOff ) ) )
-> ( allAxesOff ) ) )
```

Intuitively, the property states 'It is always the case that if the current state is AAHOn and along all the paths, the next state is AAHOff, then it implies that there is no acceleration along all the rotational axes'.

The property verifies the transition from state *AAHOn* to the state *AAHOff*. Such a direct transition occurs when all the rotational axes are turned off. The connectives AG ensure that this property holds in all the states. The connective AX is used to refer to all the immediate future states. The left hand side of the implication is dependent only on

state information and not on any input parameters and hence this property is classified in the 2nd category.

The CMU-SMV model and the CTL properties provided by Di Vito were used as our inputs to the FormulaEditor and for the evaluation of FormulaEditor on SAFER.

We now explain the results of our evaluation of FormulaEditor on SAFER. Our evaluation results were compared with the work in [2]. The results showed that the features of FormulaEditor such as property patterns, dynamic atom selection and mapping of properties to their requirements reduce property specification time.

Reusability using Pattern File

The FormulaEditor has the option of creating a pattern of a property while a property is being specified. This property pattern can then be reused to specify other properties which are similar in nature. As mentioned earlier, the mapping from requirements to properties can be many to one. Hence to ensure that a system satisfies a particular requirement, it is necessary to ensure that all the properties associated with that requirement are verified correctly. The patterns of the property are saved in a file called a pattern file. We explain the idea of pattern files using examples.

Example 1

In the CMU-SMV model provided by Ben Di Vito, one of the properties specified in the model is *ignore_stays_on_starting_roll*. The property is given as follows:

```
ignore_stays_on_starting_roll :=
AG (AAH_state.toggle.engage = AAH_started &
   (AX AAH_state.ignore_HCM.roll)
```

```
-> ! E [!(AAH_state.toggle.engage = AAH_off &
        (AX AAH_state.toggle.engage = AAH_started)) U
        !(AX AAH_state.ignore_HCM.roll) &
        !(AAH_state.toggle.engage = AAH_off &
         (AX AAH_state.toggle.engage = AAH_started))]);
```

Intuitively, the property states that 'It is always the case that if current state is AAHStarted and along all paths in the next state if the roll acceleration from the hand controller module is ignored, then there does not exist a path such that a state is reached where the roll acceleration is not ignored and the state remained in AAHStarted'.

The complexity of the property is clearly visible. The specification of such a property is both time consuming and error prone. Furthermore, there is a variant of this property which also needs to be verified which is named as *ignore_stays_off_starting_roll*. This property is given as

```
ignore_stays_off_starting_roll :=
 AG (AAH_state.toggle.engage = AAH_started &
     !(AX AAH_state.ignore_HCM.roll)
   -> ! E [!(AAH_state.toggle.engage = AAH_off &
          (AX AAH_state.toggle.engage = AAH_started)) U
          (AX AAH_state.ignore_HCM.roll) &
          !(AAH_state.toggle.engage = AAH_off &
           (AX AAH_state.toggle.engage = AAH_started))]);
```

Intuitively, the property states that 'It is always the case that if current state is AAHStarted and along all paths in the next state if the roll acceleration from the hand controller module is not ignored, then there does not exist a path such that a state is reached where roll acceleration is ignored and the state remained in AAHStarted'.

The difference between the two properties is small. The only difference is that in the first property the variable (AX AAH_state.ignore_HCM.roll) is present before the

implication while its negation is present after the implication and vice versa for the second property. It would be inconvenient to specify the entire property in each case. In both cases, the variable AAH_state.ignore_HCM.roll is replaced by its negation. The rest of the property remains the same. When such properties need to be written again, it increases the property specification time.

FormulaEditor gives the convenience of creating a partially instantiated pattern for a property which can be reused in the future. While specifying the first of these two similar properties, the pattern file can be created. The steps to be followed while creating a pattern are 1) Use common patterns to generate un-instantiated properties 2) Instantiate required parameters with atoms 3) Move the partially instantiated property to the 'moved properties' section using the MOVE button, and 4) Save the moved property as a pattern in a new or existing pattern file. This saved pattern can now be reused for specifying the second property.

In this example, the pattern can be partially instantiated as follows.

```
Pattern =

 AG (AAH_state.toggle.engage = AAH_started & Clause 1
-> ! E [!(AAH_state.toggle.engage = AAH_off &
   (AX AAH_state.toggle.engage = AAH_started)) U
   Clause 2 &
   !(AAH_state.toggle.engage = AAH_off &
    (AX AAH_state.toggle.engage = AAH_started))]);
```

This pattern is partially instantiated, and only two parameters need to be instantiated with atoms. Thus, instead of having to specify and instantiate seven parameters, the user only has to choose a single pattern and instantiate two parameters. In addition, the user has the flexibility to reuse this pattern for any similar property.

Example 2

A second example demonstrates the utility of patterns for verification of SAFER properties. We explain this example with the help of snapshots of the FormulaEditor tool for better understanding.

This property, which is named as "*no_rot_no_ignore_roll*" in the CTL properties provided by Ben Di Vito (refer to P24 in Appendix) checks that if in the beginning of a cycle, (a cycle starts when the *AAH off* state is passed followed by *AAH started* state) the roll command is not active, then it does not become active until the *AAH off* state is passed in a new cycle. The property is written as follows

```
( A G ( ( ( ( AAHState.toggle.engage = AAHOff )
        & ( A X ( AAHState.toggle.engage = AAHStarted ) ) )
        & ( rotGrip.roll = ZERO ) )
-> ( ! ( E [ ( ! ( A X ( AAHState.toggle.engage = AAHOff ) ) )
        U ( ( A X ( AAHState.ignoreHCM.roll ) )
        & ( ! ( A X ( AAHState.toggle.engage = AAHOff ) ) ) ) ] ) ) ) )
```

Intuitively, the property states that 'It is always the case that if there is no roll acceleration in the beginning of the cycle, then there does not exist a path such that the roll acceleration from hand controller module is ignored without the state becoming AAHOff'.

Another similar property which needs to be monitored is named as "*rot_cmd_ignore_roll*" (refer to P25 in Appendix). This property is written as

```
( A G ( ( ( ( AAHState.toggle.engage = AAHOff )
        & ( A X ( AAHState.toggle.engage = AAHStarted ) ) )
        & !( rotGrip.roll = ZERO ) )
-> ( ! ( E [ ( ! ( A X ( AAHState.toggle.engage = AAHOff ) ) )
        U ( !( A X ( AAHState.ignoreHCM.roll ) )
```

& (! (A X (AAHState.toggle.engage = AAHOff))))]))))

Intuitively, the property states that 'It is always the case that if there is roll acceleration in the beginning of the cycle, then there does not exist a path such that the roll acceleration from hand controller module is not ignored without the state becoming AAHOff'.

We can identify the pattern for these two properties and utilize the pattern file feature of Formula Editor to save this pattern and reuse it while specifying similar properties.

The pattern is as follows

(A G ((((AAHState.toggle.engage = AAHOff)
& (A X (AAHState.toggle.engage = AAHStarted)))
& **Clause 1**)
-> (! (E [(! (A X (AAHState.toggle.engage = AAHOff)))
U **Clause 2**)
& (! (A X (AAHState.toggle.engage = AAHOff))))]))))

Figure 6 shows the creation of the pattern for these properties. Using the highlighted *move* button, the pattern is placed in the moved properties tab. As explained in the first example, the four steps are followed to create the pattern. Using the common patterns available in the CTL pattern Selection tab (as shown in Figure 6), the un-instantiated pattern is created. The un-instantiated pattern as displayed in the syntax directed property editing area looks as follows:

(*ForAllPaths always* ((**CTLproperty1** and ((*ForAllPaths next* **CTLproperty**) and **CTLproperty2**)) *implies* (*Not (ExistsPath [(Not (ForAllPaths next* **CTLproperty**)) *until (* **CTLproperty1** *and (Not (ForAllPaths next* **CTLproperty**)))]))))

This above un-instantiated property is then partially instantiated using the atoms present in the Atom Selections tab (as shown in Figure 6) to produce the property shown in Figure 6. Figure 7 shows how the pattern is saved. The pattern can be saved into any directory. This saved pattern can be added to the list of available patterns by specifying this directory in the model panel of Formula Editor as shown in Figure 8.

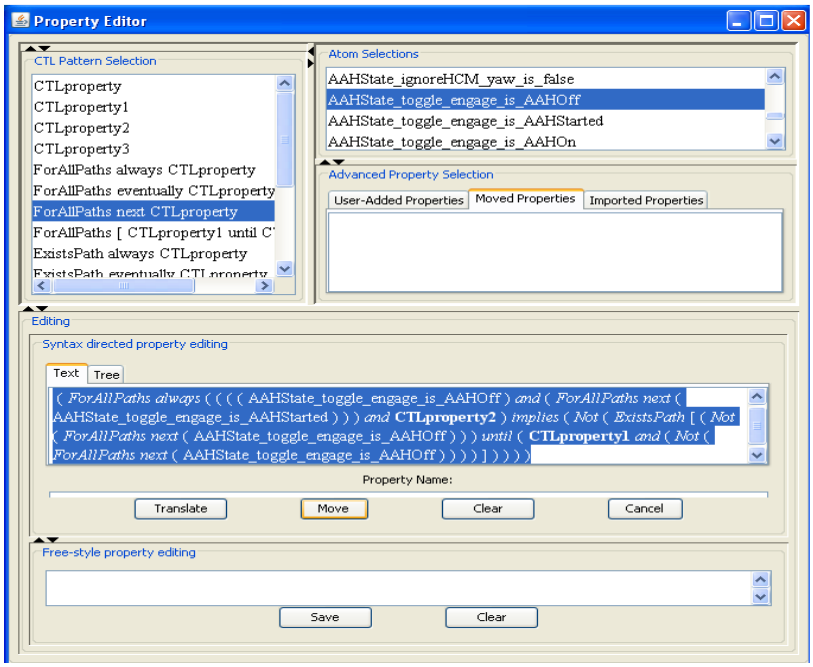

Figure 6: Pattern Creation

Figure 9 shows how the pattern is used to specify the properties more easily. Only two conditions in the pattern need to be instantiated which is convenient as compared to rewriting the entire property again. The instantiation of these conditions for the property "*no_rot_no_ignore_roll*" is shown in Figure 10. In a product line where a product with many such similar properties is present, it becomes highly convenient to have such patterns.

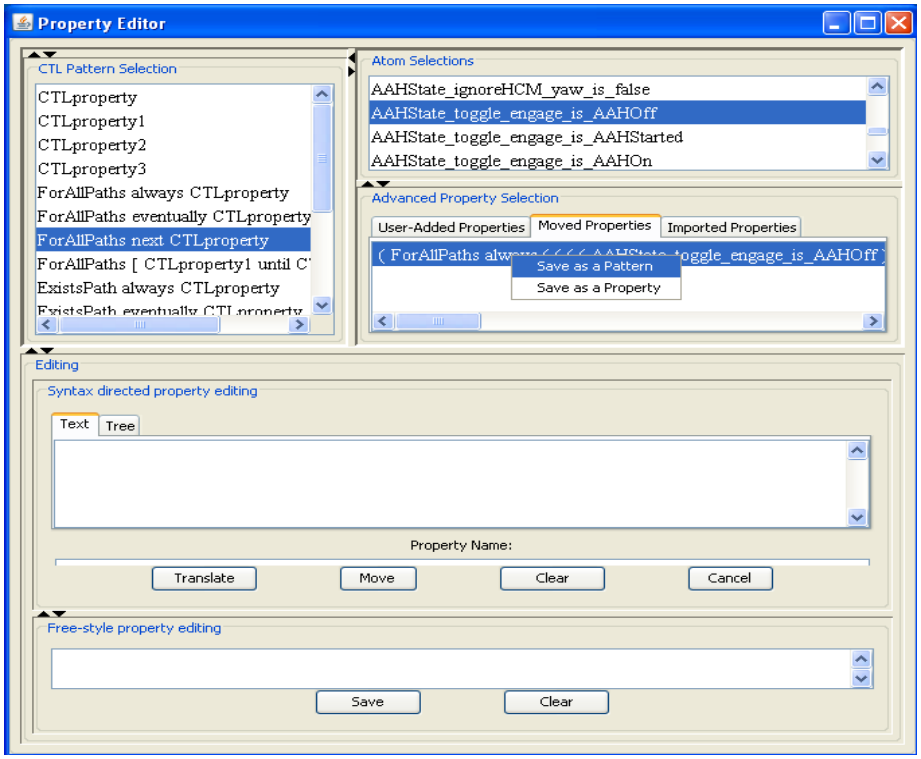

Figure 7: Saving the Pattern

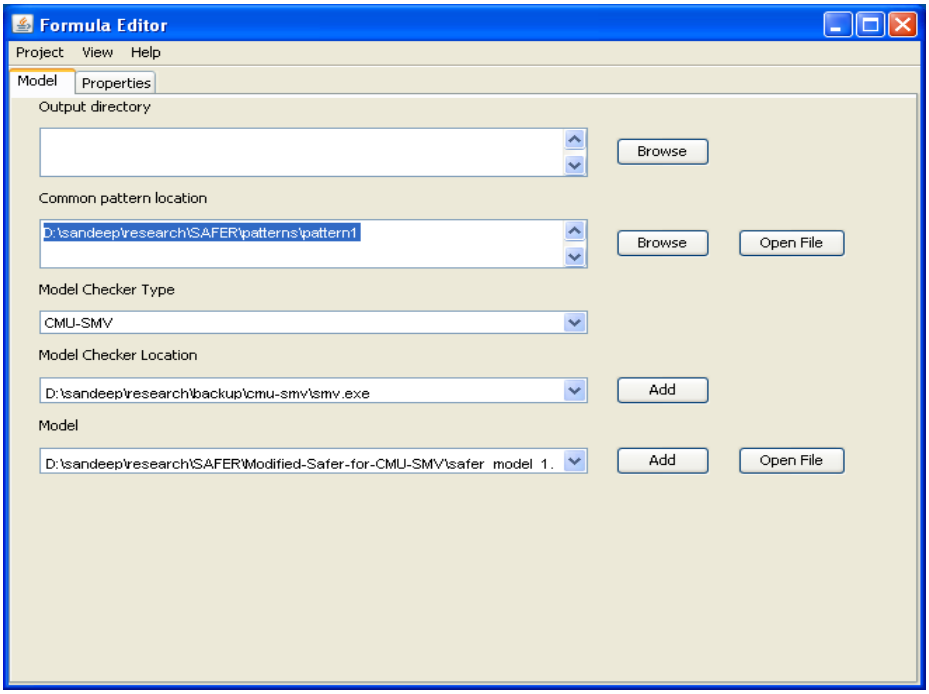

Figure 8: Reusing the pattern

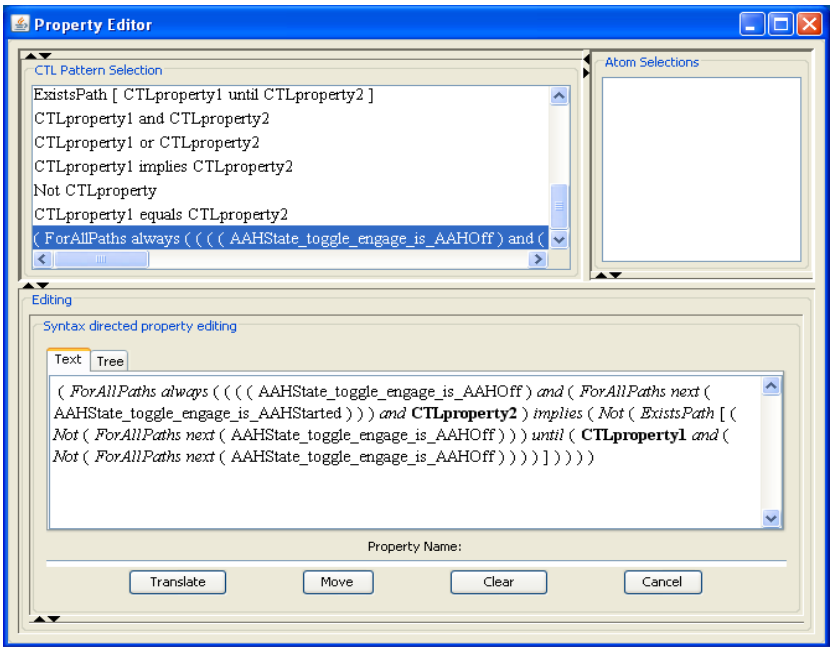

Figure 9: Using saved pattern

Dynamic Atom Selection

The FormulaEditor has another useful functionality. FormulaEditor provides the advantage to user to not remember all the variables in the specified model. When the location of the model is specified in the model panel, at runtime, FormulaEditor automatically locates all the variables in the models and displays them module-wise to the user. The user can select the required variables (atoms) from the generated list for property specification. Also, the development of a product or a product line, as well as the property specification and verification of these specified properties can be achieved in stages. To allow separation of privileges, these tasks could be executed by different groups of people. For example, one set of individuals could work on the development of a product/product line. Once this task was completed, the task of property specification could be allocated to a different set of individuals. Further, the task of verification of

these properties against the product model could be allocated to a third set of individuals. In such an environment, the tasks of product/product line development and property specification would require these two sets of individuals to have expert domain knowledge. However, the individuals who verify the properties may not require expert domain knowledge. Their task could be to report back the results of the verification to the previous two groups.

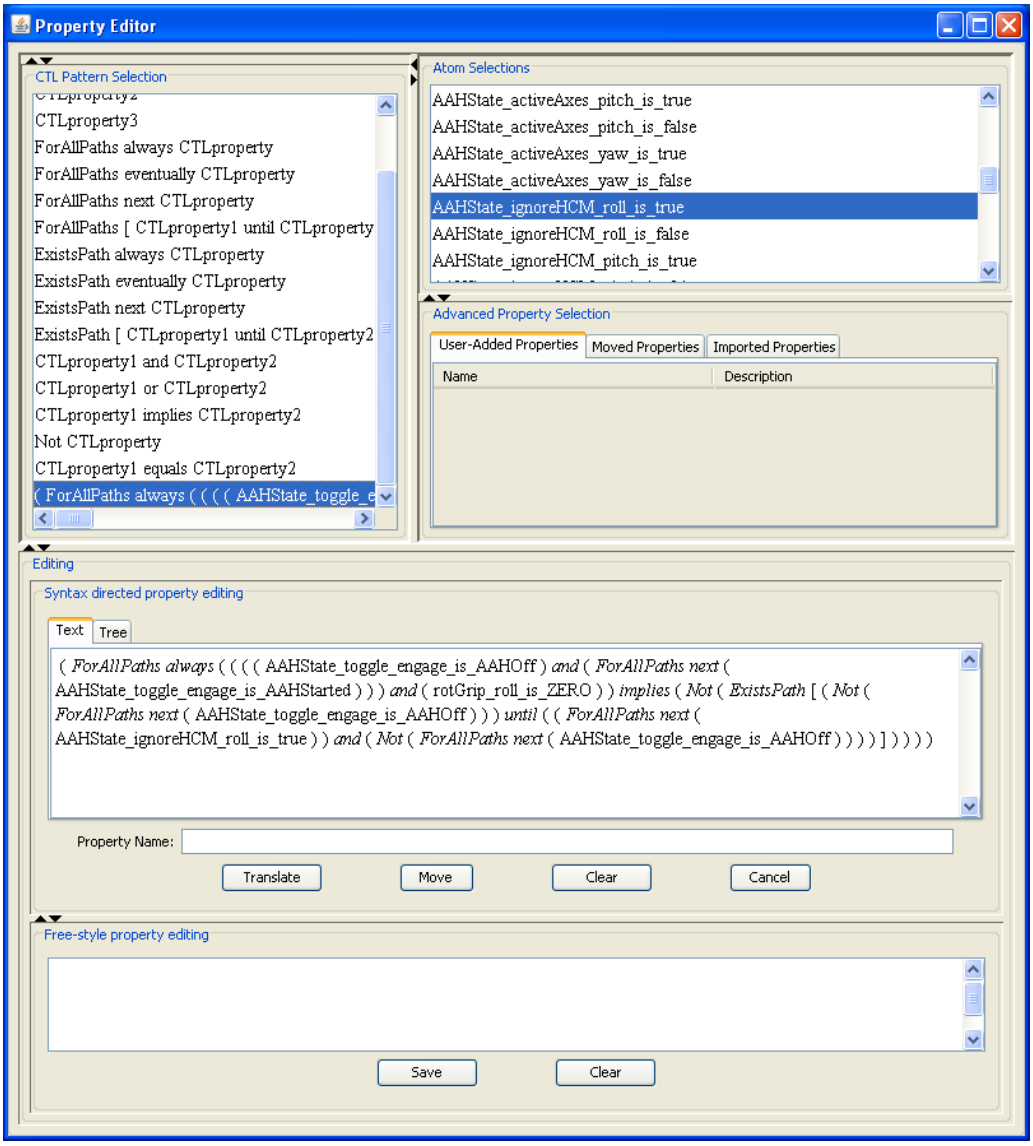

Figure 10: Instantiating saved pattern

FormulaEditor provides a good user interface which allows the facility to select the atoms dynamically when the properties are being created. The tester can look at the list of the atoms which are dynamically generated from the model file to verify the properties.

If a meaningful naming convention is followed by the developers of the model and the individuals who create the properties, then it will become easy for the testers to look at the list of the properties and dynamically select the atoms to verify these properties.

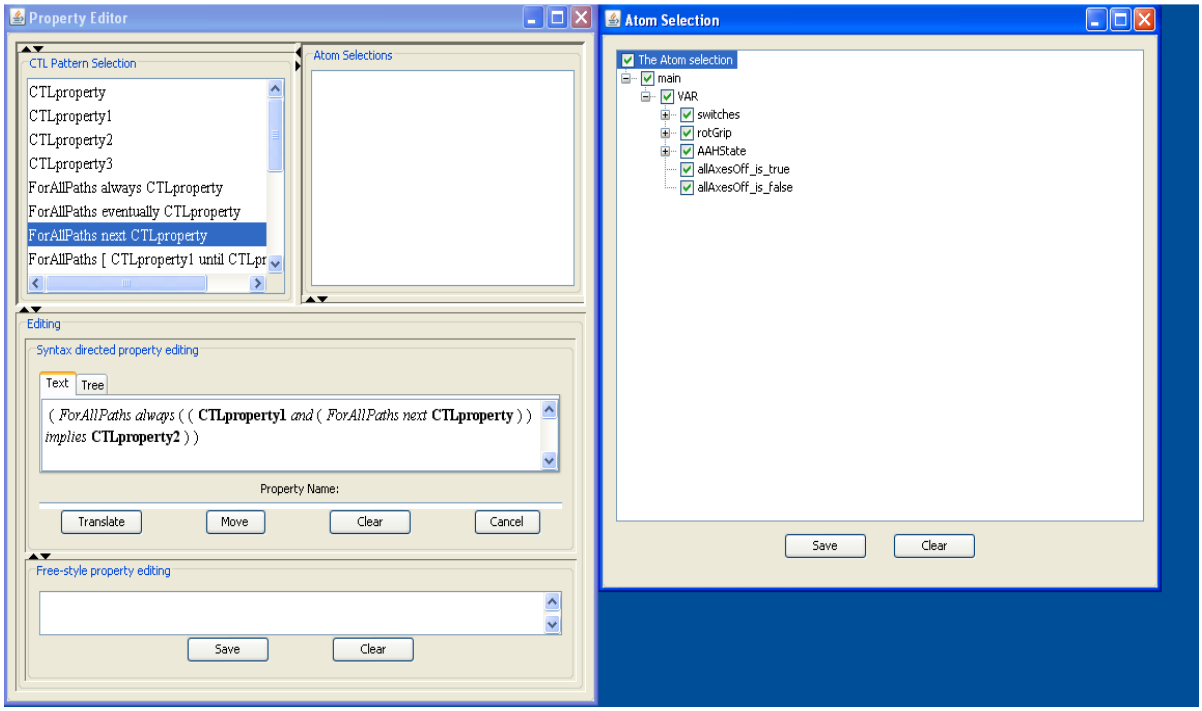

Figure 11: Atom Selection

In a very complicated model with hundreds of atoms, it would be very difficult to remember all the names of the atoms. This facility to select the atoms from the Atom Selection list can provide a convenient approach to property verification.

Figure 11 and Figure 12 show the use of dynamic atom selection using FormulaEditor described above while verifying the property named as *on_to_off_direct* and numbered as P9 in the Appendix. The right window in Figure 11 shows the listing of the variables in the specified model in a module-wise manner. In Figure 12, the atom selections show the list of the selelcted atoms from Figure 11. The property being verified is

(A G (((AAHState.toggle.engage = AAHOn) & (A X (AAHState.toggle.engage = AAHOff)))
-> (allAxesOff)))

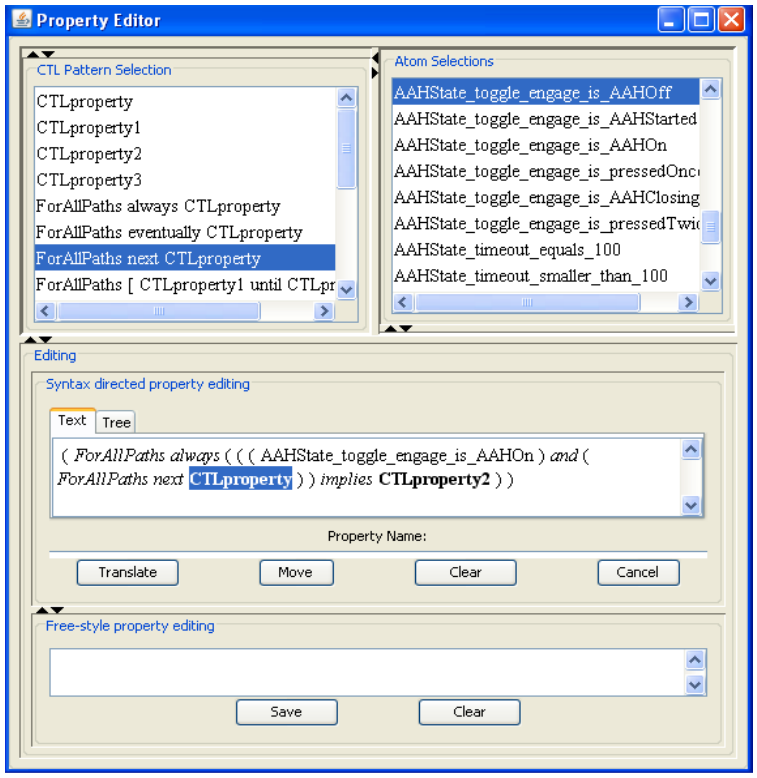

Figure 12: Use of selected atoms in property specification

Finding flaws in a specified model

Apart from being advantageous for specifying properties, the FormulaEditor also helps in finding faults in the model specification. The results of verifying the properties, helps the developer understand if the model is behaving in the intended manner. The model uses SMV model checking to verify the properties against the given language. The outputs from the model checker are associated with the property for ease of use. All the information pertaining to a property before its verification, while verification is being carried out, and after the verification is associated with the property. The information about the property after verification includes the results of the verification, information about the time needed for verification and states explored, and the counterexample if generated. FormulaEditor conveniently maps all the output information to the property which can be viewed by right clicking the property in the property panel.

Failed Property

If a failed property is encountered, i.e., if a property was required to be satisfied but the model checker produces a counterexample, then the FormulaEditor gives the corresponding output from the model checker. This output provides a trace for the specified property. The trace helps the user to track down the flaw in the model or in specifying the properties and correct the model. This feature is available because of the underlying model checker used to verify the properties against the model. But the FormulaEditor gives the user a convenient method to map the properties to the traces received from the FormulaEditor.

Example: The detection of flaws in specification of properties is shown by the following example. Ben Di Vito identified a flaw in the original SAFER model ([22]) in his work ([2]). While verifying the AAH properties on the older version of SAFER that he used, he identified an incomplete specification in the model. The AAH state diagram in Figure 5 shows that when the current state is "*pressed_once*", there are two possible transitions when the *AAH_switch* is in the up position. The transition can be either to the "*AAH_on*" state or to the "*AAH_closing*" state. This transition depends on the value of the counter *timeout.* In the state diagram for the original SAFER version, the transition from "*pressed_once"* to "*AAH_on"* was not considered. Consequently, this improper specification allowed a button-up transition, while in the "*pressed_once"* state to make a transition to the "*AAH_closing"* state, where a button-down transition would change the state to "*pressed_twice"* without considering the 0.5 second period. We recreated the error for analyzing the use of FormulaEditor with CMU-SMV. While specifying the properties related to transition from "*pressed_once*" state, we omitted this dependency on the timeout counter in our modeling effort.

Consequently, the incorrect property that we tried to verify was

```
( A G ( ( ( AAHState.toggle.engage = pressedOnce )
        & ( switches.AAH = buttonUp ) )
        -> ( A X ( AAHState.toggle.engage = AAHClosing ) ) ) )
```

The FormulaEditor, supported by the CMU_SMV model checker, correctly threw the result as false. The error produced gave a trace on the value of the timeout counter as follows.

….

state 1.2:
allAxesOff = 0
switches.AAH = buttonUp
AAHState.activeAxes.roll = 1
AAHState.activeAxes.pitch = 1
AAHState.activeAxes.yaw = 1
AAHState.ignoreHCM.roll = 1
AAHState.ignoreHCM.pitch = 1
AAHState.ignoreHCM.yaw = 1
AAHState.toggle.allAxesOff = 0
AAHState.toggle.stateA = AAHOn
AAHState.toggle.upTransition = AAHOn
AAHState.toggle.engage = AAHStarted

-- loop starts here –
state 1.3:
switches.AAH = buttonDown
AAHState.toggle.downTransition = pressedOnce
AAHState.toggle.engage = AAHOn

state 1.4:
AAHState.toggle.stateB = AAHClosing
AAHState.toggle.upTransition = AAHClosing
AAHState.toggle.engage = pressedOnce
AAHState.timeout = 100

state 1.5:
AAHState.timeout = 99

state 1.6:
AAHState.timeout = 98

state 1.7:
AAHState.timeout = 97
….
state 1.102:
AAHState.timeout = 2

state 1.103:
AAHState.timeout = 1

state 1.104:
switches.AAH = buttonUp
AAHState.toggle.stateB = AAHOn
AAHState.toggle.upTransition = AAHOn
AAHState.timeout = 0

In the above counterexample note that, from state 1.3 to 1.4, the engage variable (representing the 6 states of AAH state diagram) moves from state *AAHOn* to *pressedOnce*. The button position did not change from buttonDown. In states 1.5 till 1.103, only the value of the timeout kept decrementing. In state 1.104, the button position moved to buttonUp, and the value of engage remained as pressedOnce. However, the property was not satisfied and the counterexample trace pointed that the next state must be AAHOn. Evaluation of this trace points out that the transition is not only dependent on the position of the button but also on the value of the timeout counter. This showed that the property must also include a condition on the timeout counter. The property was then modified as follows to correct the flaw.

```
( A G ( ( ( ( AAHState.toggle.engage = pressedOnce )
        & ( AAHState.timeout > 0 ) )
        & ( switches.AAH = buttonUp ) )
        -> ( A X ( AAHState.toggle.engage = AAHClosing ) ) ) )
```

This property was verified to true. The counter-examples are provided by the underlying model checker, in this case CMU-SMV. However, FormulaEditor provided the ease of encapsulating the properties with their corresponding results and this provided efficient management of properties.

Finding Design Flaws in the Model

As explained in the section of failed property, distribution of work can be achieved by assigning different tasks to different individuals in an industry environment. In such cases, it is possible that the person who develops the model does not take into account certain features of the model whereas the person who specifies the properties includes them. The third person verifying the properties may obtain contrary to the

expected results. Such discrepancies could be reported back to the other teams and it could be understood whether the flaw is in the model or in the property specification. This can be illustrated with the following example.

In Figure 5, the AAH state diagram shows that when the AAH is in state "*AAH on*", then there is a transition to "*AAH off*" state if all the three axes are off. Suppose that the person modeling the state diagram and the person specifying the properties receive the state diagram shown in Figure 5 and the person modeling the state diagram misses this transition from "*AAHOn*" to "*AAHOff*" although the model was required to include this transition. This would result in an incorrect model specification. If the person specifying the properties for the AAH state diagram correctly specifies the requirement in the property the design flaw could be identified. Similarly, the correctness of the model can be explored by specifying properties that should not be satisfied by the model. One such property can be considered below:

```
( A G ( ( AAHState.toggle.engage = AAHOn )
        -> ( ! ( E X ( AAHState.toggle.engage = AAHOff ) ) ) ) )
```

This property checks that if the current state is "*AAHOn*", then there is no path such that the next state is "*AAHOff*". We expect a counterexample to be found as this property violates the requirement that there is a path from the state "*AAHOn*" where the next state is "*AAHOff*". The original state diagram shows that there is a direct transition from "*AAHOn*" to "*AAHOff*". As a result, the person verifying this property will observe that this property is verified to true although a counterexample should have been produced. This results in identification of the design flaw, i.e. the missing transition from "*AAHOn*" to "*AAHOff*". This often occurs due to incorrect description of the model or

incorrect specification of the property. Here we saw an example of incorrect description of the model. If the property is incorrectly specified (generally because of certain missing conditions) but the model is correct, the unexpected result could help detect the incorrect property.

The improvements made to the FormulaEditor after incorporating CMU-SMV have allowed additional flexibility for the user to specify models in multiple languages and model check them using FormulaEditor. Ease of specification and verification of properties using pattern files as well as detection of flaws are two of the main advantages of FormulaEditor. These features have been further enhanced by the addition of CMU-SMV model checker to FormulaEditor. The advantages of using FormulaEditor for specification and verification of properties on a single product have been explained earlier in this chapter. The next chapter describes the evolution of a product line for SAFER and the application of FormulaEditor on the SAFER product line.

CHAPTER 4. FORMULAEDITOR ON *SAFER* PRODUCT LINE

FormulaEditor has been built with the aim of enhancing reusability in specification and verification of properties. The previous chapter discussed reusability during verification of properties for the single product of SAFER. To demonstrate the usefulness of the updates to FormulaEditor, we here propose a product line of SAFER-based system and use the reusability feature of FormulaEditor for specification and verification of properties. We demonstrate the usefulness of the pattern files in the verification of the commonalities of the product line.

4.1 Proposed SAFER product line

In Chapter 3, we examined the advantage of Formula Editor for verifying properties for a single model of *SAFER*. *SAFER* was designed for the sole purpose of rescue in extra-vehicular activity. The available articles and technical reports on the SAFER device explore the features of SAFER [2], [22]. However, no attempt to develop a product line for SAFER has been made prior to this work. In this chapter we extend SAFER into a product line based on the maneuverability of *SAFER*.

SAFER Product line

The product line that we use to evaluate FormulaEditor in a product line context is based on variations in the maneuvering capability of *SAFER*. We describe four products in the product line. The products have features that are common to them. These are known as *commonalities* for the product line. Each product also has certain features which are specific to it in order to accomplish a particular task or provide specific

functionality. These features are known as *variabilities* of the product line. In this section we explain the essential features of each of the four products in the product line showing the commonalities and the variabilities among the products.

The original *SAFER* device was designed to provide 6-degree of freedom propulsion capacity. Maneuverability was allowed along three translational axes namely X, Y and Z and along three rotational axes namely yaw, pitch and roll. It also provided the feature of Automatic Attitude Hold (AAH) which helps to bring and keep the rotation rates close to zero. The primary or base product in the product line has the mandatory features for an Extra-Vehicular Activity (EVA) device. The product line evolves by adding features incrementally to the base product [23].

Our *SAFER* product line consists of the following four products:

1) ***Base-SAFER***

2) ***Base-SAFER-Cruise***

3) ***AAH-SAFER***

4) ***AAH-SAFER-Cruise***

We use the ***FAST*** (Family-Oriented Abstraction, Specification, and Translation) process described in [42], [43] for the generation of the members of the SAFER product line. This process gives a formal approach to the development of the members of the product line. The different steps in the *FAST* process are

1. Commonality analysis
2. Module guide
3. Mapping from parameter of variations to modules

4. Use relationship
5. Decision model table
6. Dependency graph

Step 1- Commonality Analysis: We first describe the commonalities and variabilities for the product line as per the *FAST* process.

Commonalities: The commonalities for the SAFER product line are adopted from the features of the original SAFER discussed in the previous chapter [2], [22]. The common features for every product in the product line are:

[C1] Every product provides a six degree-of-freedom maneuverability; 3 for translation along the X, Y and Z direction and 3 for rotation to enable yaw, pitch and roll.

[C2] Translation commands are prioritized so that only one translational axis receives acceleration, with the priority order being X, Y and then Z.

[C3] If both translation and rotation commands are present simultaneously, rotation takes priority and translations will be suppressed. This feature is applicable to all the products in the product line when they function in the basic mode (neither AAH nor Cruise option is selected).

These common features form the commonalities of the product line. We chose these features as they describe the maneuvering capability of SAFER. We maintain the complexity of SAFER by retaining the six degree-of-freedom maneuvering capability (C1). We model our systems and test them for each of the commonalities. Specifically,

every product in the product line is tested using FormulaEditor to verify whether the commonalities C2 and C3 are satisfied by each of them.

Variabilities- The SAFER product line has essentially two variations: *Automatic Attitude Hold (AAH)* and *Cruise Control (CC)*. The AAH feature is the same as the one present in the original SAFER case study. We introduce a new variability named Cruise Control. This feature enables the crewmember using this device to enable a mode similar to an auto-pilot mode. The auto-pilot mode differs from the AAH-mode by allowing the crew-member to maintain velocity of the device along the X, Y and Z directions. The AAH mode in the original *SAFER* device was used to bring and keep the rotation rates close to zero. The AAH feature automatically adjusted the orientation with respect to the Intertial Reference Unit (IRU). Also, the *SAFER* case study ([22]) explains that translation commands have to be explicitly given to continue motion in a particular direction as the AAH module handles the orientation about the rotation axes and not translational axes. We extend this feature to the translational axes. In the CC mode, the translational accelerations of the device are maintained until acceleration is given along any of the 3 axes, namely X, Y and Z. Rotation has to be given explicitly to maintain the orientation of the device.

Table 2 gives a compact format of the commonalities, variabilities and parameters of variation for the SAFER product line.

	Commonalities
C1	Six degrees-of-freedom
C2	Prioritized translation

C3	Rotation to translation priority
	Variabilities
V1	AAH mode
V2	Cruise mode
	Parameters of variation
PV1	AAH: present, absent
PV2	Cruise: present, absent

Table 2: Commonalities, Variabilities, and Parameters of Variation

Step 2- Module guide: After the commonality analysis, the second step is the creation of the module guide. The module guide lists the modules in the product line. In the case of a hierarchical model, the module guide describes the module hierarchy which shows how modules are decomposed into submodules for information hiding. The product line that we are developing has variations based on the maneuverability of the device. The modules that we are interested in are hence related to the maneuverability aspect of the product. In general, the device allows rotation and translation capability. In addition it also provides the AAH and the crusie capabilties, and the module guide lists the modules for all these features.

Module List	Name
M1	Translation
M2	Rotation

M3	AAH-transition
M4	Cruise-transition

Table 3: SAFER modules

The modules M1 and M2 handle the commonality requirements for providing maneuverabiltiy along the six directions. The modules M3 and M4 provide the AAH and cruise functionalities respectively.

Step 3 - Mapping from Parameter of Variations to modules: We give a mapping from the parameters of variation to the modules. We have two variabilities, and their parameters have boolean values of present or absent. Table 4 describes the mapping from the parameters of variation to the corresponding modules.

Parameter of Variation	Modules
PV1.present	M1, M2, M3
PV1.absent	M1, M2
PV2.present	M1, M2, M4
PV2.absent	M1, M2

Table 4: Mapping from parameters of variation to modules

Step 4- Uses Relationship: The next step is to show the uses relationship among the modules. As shown in the module guide our focus is on four modules namely M1, M2, M3 and M4.

The translation module is used by cruise transition module and since translation uses the rotation module, the rotation module is indirectly used by the cruise transition module. Irrespective of whether cruise or AAH option is present in the product, the translation and rotation modules will be used by the device.

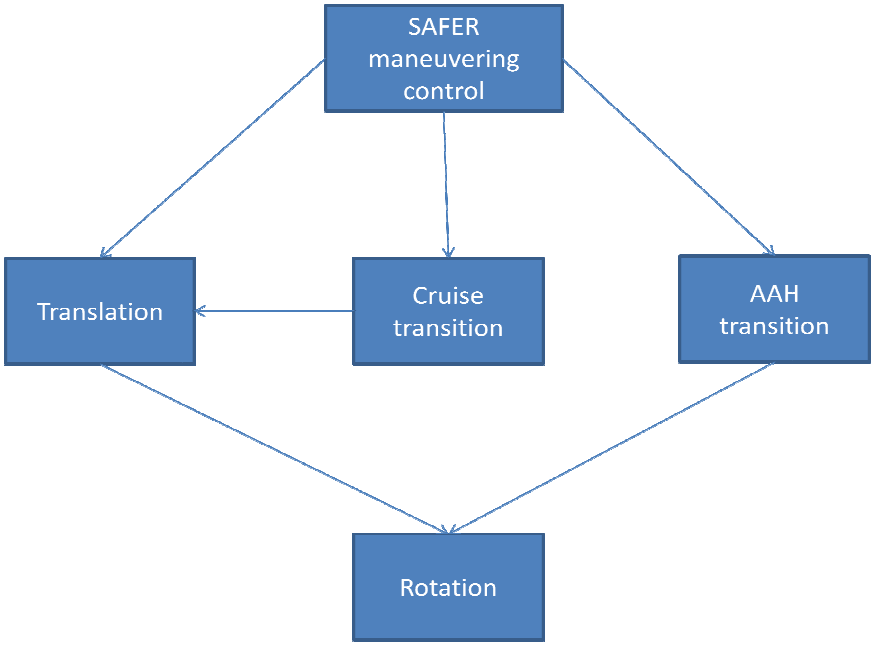

Figure 13: Uses relationship for SAFER product line

Step 5 - Decision model table: After describing the uses relationship, we generate the decision model table by extending the table of parameters of variation to add a column

for constraints, especially noting dependencies among parameters and a column for the mappings to modules. The two variabilities in our product line name AAH and cruise are independent of each other and hence do not have any constraints between each other. Table 5 describes the decision model table.

Variability	Name	Value Set	Constraints	Module Mapping
V1	AAH mode	Present, Absent	None	Present: 1&2&3 Absent: 1&2
V2	Cruise mode	Present, Absent	None	Present: 1&2&4 Absent: 1&2

Table 5: Decision Table for SAFER

Step 6- Dependency graph: After applying these five steps, to make consistent decisions regarding each new product *i*, the variabilities and constraints are modeled in the form of a dependency graph. Each variability is a node in the graph with a property set including its name, description, and valid parameter values. The nodes have two kinds of edges: outgoing edges and incoming edges, corresponding to constraints. Decision making is done by using different graph-walking algorithms to traverse through the graph such as failure-first optimization (FFO) or Least Options Optimization (LOO) [43]. Depending on the order in which the vaiabilities are selected, the graph will be accordingly pruned according to the constraints.

In the product line that we developed, the two variabilities, namely AAH and CC, are independent of each other since each handles a separate feature. The AAH handles the rotation feature of the device whereas CC handles the translation feature of the device. Hence there are no constraints between these variabilities. The dependency graph consists of 2 disconnected nodes without any edges between them. This means that the sequence in which the variabilities are selected does not affect the final result.

These 6 steps lead us to the development of the SAFER product line.

4.2 Results of FormulaEditor on SAFER product line

The Appendix shows the model files for the four products in the SAFER product line. This section explains the process of applying FormulaEditor to the four products in the product line and explains the utility of the FormulaEditor by evaluating the results.

Modeling Products of SAFER product line:

The appendix contains the model file for the original SAFER product provided by Ben Di Vito [2]. This is followed by the model files for the four products in the SAFER product line. We give a brief description of these four models below. Each of the four products are modeled to provide six-degrees-of-freedom, prioritized translation, and priority to rotation over translation. The Commonalities and variabilities are then verified with the four models.

Base SAFER: The model file for Base SAFER models the simple six-degree-of-freedom capability of SAFER. Since we require single translational acceleration to be active at a time, we introduce three variables for the effective acceleration along each translational axis. The idea of effective acceleration can be explained as follows. In the presence of a

positive X acceleration and a negative Y acceleration, the commonality C2 requires that X be given priority over Y. Thus the effective acceleration along X axis must be positive but the Y acceleration must be suppressed. Hence the effective Y acceleration must be zero.

Base SAFER Cruise: *Base SAFER Cruise* enhances the features of Base SAFER by adding the Cruise Control capability to it. The modules *tranCommand* and *cruiseTransition* are modeled to allow the feature of Cruise Control. To ensure that while in cruise mode, the translational accelerations are maintained, three variables were introduced to retain the previous accelerations in the X, Y and Z axes. The retained accelerations are equal to the effective accelerations in previous state.

AAH SAFER: AAH model is a modified version of the original *SAFER* model which introduces translational accelerations to the earlier version. The tranCommand module is introduced to incorporate this features of six-degrees-of-freedom, prioritized translation, and priority to rotation over translation.

AAH SAFER Cruise: *AAH SAFER* Cruise combines the features of AAH and Cruise. Essentially, the model for *AAH SAFER* Cruise is a grouping of the models for *AAH SAFER* and *Base SAFER Cruise*. The commonalities, C1, C2, and C3 as well as the variations of cruise mode and AAH are joined together in the model.

Verification of Commonalities: The commonality requirements C2 and C3 are to be satisfied by each product in the product line. C2 states that the translation commands are prioritized so that only one translational axis receives acceleration at a time, with the priority order being X, Y and then Z. C3 states that if both translation and rotation

commands are present simultaneously, rotation takes priority and translations will be suppressed.

Verification of common properties over all the products in the product line requires specification of these properties for each product. However, the application of FormulaEditor to our proposed SAFER product line models eliminated the extra work of re-specification of the properties. The use of property pattern for a commonality enabled reuse of this pattern throughout all the four models thereby reducing the specification and verification time.

We explain the verification of C2 and C3 using pattern files below.

Base-SAFER- Using the model file for Base-SAFER shown in Appendix, we conducted verification of the commonalities for this model. The commonality verification is explained below.

C2: Commonality C2 ensures prioritized acceleration along the translational axes, allowing only one axis at a time, with the priority being X, then Y and then Z. We verified the following property on the model.

```
(AG( ( ( ( tranGrip.noRotCmd ) & ( tranGrip.XAcc = NEG ) ) & ( tranGrip.YAcc = POS ) )
-> ( ( tranGrip.XAccEffect = NEG ) & ( tranGrip.YAccEffect = ZERO ) ) ))
```

C2 is specific to priority among translational axes. Hence, we verify this property when there is no rotational acceleration since the presence of rotational acceleration would suppress any translational acceleration. The property checks that in the absence of rotational acceleration, a negative acceleration along X axis and a positive acceleration along Y axis, priority is given to the X axis. Hence the effective X axis acceleration is

negative and the effective Y acceleration is suppressed to zero. We name this property as *no_rot_X_priority_Y*.

We used the same steps outlined in the previous chapter to create the property pattern i.e. 1) Use common patterns to generate un-instantiated properties 2) Instantiate required parameters with atoms 3) Move the partially instantiated property to the 'moved properties' section using the MOVE button, and 4) Save the moved property as a pattern in a new or existing pattern file. After specifying the property *no_rot_X_priority_Y*, it was saved as a pattern by using the move button. The pattern was saved in a file named *commonality_priority_patterns*. In the model panel, we then added the path to this pattern file so that the pattern is available for reuse. We verified similar properties for the other axes by re-using this property pattern and reinitializing the existing atoms with the atoms for the appropriate axes.

C3: Commonality C3 ensures that rotation takes priority over translation. We first verified the following property.

```
(A G ((( rotGrip.roll = POS) & ((tranGrip.XAcc = POS)|((tranGrip.YAcc = POS)|(tranGrip.ZAcc = POS))))  -> ((tranGrip.XAccEffect = ZERO)&((tranGrip.YAccEffect = ZERO)&(tranGrip.ZAccEffect = ZERO)))))
```

The property checks that in the presence of a rotational acceleration, any translational acceleration will be suppressed as rotation has priority over all the translations. This property is named as *rot_pos_priority_all_tran*. We saved this in the same pattern file *commonality_priority_patterns*. The reason for saving both patterns for C2 and C3 in the same file was for the convenience of the user. Both these patterns were related to commonalities. Hence by specifying this *commonality_priority_patterns* as the

common pattern file in the model panel, the patterns could be reused for checking whether the commonalities hold for the other products. Separate pattern files for each commonality could also be created. However this would require changing the common pattern file in the model panel to check each commonality.

For any model for another product with the same naming scheme as used by *Base-SAFER*, the above described patterns can be reused. The patterns will be displayed in the list of the available patterns along with other common built-in patterns. In the model panel, the pattern file *commonality_priority_patterns* was selected and used to specify the commonalities in each of the other three products. Their use is explained below.

Base-SAFER-Cruise- The introductions of the cruise mode in Base-SAFER-Cruise required that the commonalities by checked when the cruise mode is disabled and when it is enabled. The patterns created while verifying this property in Base-SAFER were re-used to specify these properties in *Base-SAFER-Cruise*. The patterns were reinstantiated by adding the variable which checks that the Cruise mode is disabled or enabled.

C2: The properties to verify C2 are as follows.

Cruise disabled:

(A G (((((tranGrip.noRotCmd) & (cruiseState.engage = cruiseOff))& (tranGrip.XAcc = NEG)) & (tranGrip.YAcc = POS)) -> ((tranGrip.XAccEffect = NEG) & (tranGrip.YAccEffect = ZERO))))

Cruise enabled:

(A G (((((tranGrip.noRotCmd) & (cruiseState.engage = cruiseOn)) & (tranGrip.XAcc = NEG)) & (tranGrip.YAcc = POS)) -> ((tranGrip.XAccEffect = NEG) & (tranGrip.YAccEffect = ZERO))))

The pattern created for this property in Base-SAFER eliminated the need to specify entire properties again. We reinitialized the first atom (tranGrip.noRotCmd) to include another atom which checks the cruise state. We added the additional condition named cruiseState.engage = cruiseOff and cruiseState.engage = cruiseOn to the property in Base-SAFER which checks that when the cruise control is inactive, C2 is satisfied.

C3: Similarly for C3, we reused the pattern created in Base-SAFER for C3 and reinitialized the first atom (rotGrip.roll = POS) to include the atom cruiseState.engage = cruiseOff and cruiseState.engage = cruiseOn in addition to rotGrip.roll = POS. The properties for C3 which we verified were

Cruise disabled:

(A G (((rotGrip.roll = POS) & (cruiseState.engage = cruiseOff)) & ((tranGrip.XAcc = POS)|((tranGrip.YAcc = POS)|(tranGrip.ZAcc = POS)))) -> ((tranGrip.XAccEffect = ZERO)&((tranGrip.YAccEffect = ZERO)&(tranGrip.ZAccEffect = ZERO)))))

Cruise enabled:

(A G (((rotGrip.roll = POS) & (cruiseState.engage = cruiseOn)) & ((tranGrip.XAcc = POS)|((tranGrip.YAcc = POS)|(tranGrip.ZAcc = POS)))) -> ((tranGrip.XAccEffect = ZERO)&((tranGrip.YAccEffect = ZERO)&(tranGrip.ZAccEffect = ZERO)))))

The use of the previously created patterns reduced the property verification time as it eliminated the re-specification of the similar properties.

AAH-SAFER- The patterns created in Base-SAFER were reused here also to verify the commonalities. The properties below differ from those in Base-SAFER only in a single atom. In this case, the property differed in the state for AAH. As in Base-SAFER-Cruise, here we introduced the conditions AAHState.toggle.engage = AAHOff and

AAHState.toggle.engage = AAHOn in AAH-SAFER to take into account that the property is satisfied when AAH is switched off and when it is active.

The properties for C2 and C3 are shown below.

C2:

Cruise disabled:

(AG(((((tranGrip.noRotCmd) & (AAHState.toggle.engage = AAHOff)) & (tranGrip.XAcc = NEG)) & (tranGrip.YAcc = POS)) -> ((tranGrip.XAccEffect = NEG) & (tranGrip.YAccEffect = ZERO))))

Cruise enabled:

(AG(((((tranGrip.noRotCmd) & (AAHState.toggle.engage = AAHOn)) & (tranGrip.XAcc = NEG)) & (tranGrip.YAcc = POS)) -> ((tranGrip.XAccEffect = NEG) & (tranGrip.YAccEffect = ZERO)))

C3:

Cruise disabled:

(AG (((rotGrip.roll = POS) & (AAHState.toggle.engage = AAHOff)) & ((tranGrip.XAcc = POS)|((tranGrip.YAcc = POS)|(tranGrip.ZAcc = POS)))) -> ((tranGrip.XAccEffect = ZERO)&((tranGrip.YAccEffect = ZERO)&(tranGrip.ZAccEffect = ZERO)))))

Cruise enabled:

(AG (((rotGrip.roll = POS) & (AAHState.toggle.engage = AAHOn)) & ((tranGrip.XAcc = POS)|((tranGrip.YAcc = POS)|(tranGrip.ZAcc = POS)))) -> ((tranGrip.XAccEffect = ZERO)&((tranGrip.YAccEffect = ZERO)&(tranGrip.ZAccEffect = ZERO)))))

AAH-SAFER-Cruise- For AAH-SAFER-Cruise, two conditions had to be added to the existing pattern from Base-SAFER: one for AAH mode and the other for cruise mode. The properties below show the two additions that were made. To ensure that the properties are satisfied when both cruise control and AAH are active and disabled, four properties had to be verified. Two atoms were added in each of the properties, one

corresponding to the Cruise mode and the other corresponding to the AAH mode. The properties are given as follows.

C2:

AAH and Cruise disabled

(AG((((((cruiseState.effectCruiseState = cruiseOff) & (AAHState.toggle.effectAAHState = AAHOff))
& (tranGrip.noRotCmd)) & (tranGrip.XAcc = NEG)) & (tranGrip.YAcc = POS))
-> ((tranGrip.XAccEffect = NEG) & (tranGrip.YAccEffect = ZERO))))

AAH disabled and Cruise enabled:

(AG((((((cruiseState.effectCruiseState = cruiseOn) & (AAHState.toggle.effectAAHState = AAHOff))
& (tranGrip.noRotCmd)) & (tranGrip.XAcc = NEG)) & (tranGrip.YAcc = POS))
-> ((tranGrip.XAccEffect = NEG) & (tranGrip.YAccEffect = ZERO))))

AAH enabled and Cruise disabled:

(AG((((((cruiseState.effectCruiseState = cruiseOff) & (AAHState.toggle.effectAAHState = AAHOn))
& (tranGrip.noRotCmd)) & (tranGrip.XAcc = NEG)) & (tranGrip.YAcc = POS))
-> ((tranGrip.XAccEffect = NEG) & (tranGrip.YAccEffect = ZERO))))

AAH and Cruise enabled:

(AG((((((cruiseState.effectCruiseState = cruiseOn) & (AAHState.toggle.effectAAHState = AAHOn))
& (tranGrip.noRotCmd)) & (tranGrip.XAcc = NEG)) & (tranGrip.YAcc = POS))
-> ((tranGrip.XAccEffect = NEG) & (tranGrip.YAccEffect = ZERO))))

C3:

AAH and Cruise disabled:

(A G (((cruiseState.effectCruiseState = cruiseOff) & (AAHState.toggle.effectAAHState = AAHOff)& (rotGrip.roll = POS) & ((tranGrip.XAcc = POS)|((tranGrip.YAcc = POS)|(tranGrip.ZAcc = POS))))
-> ((tranGrip.XAccEffect = ZERO)&((tranGrip.YAccEffect = ZERO)&(tranGrip.ZAccEffect = ZERO)))))

AAH enabled and Cruise disabled:

(A G (((cruiseState.effectCruiseState = cruiseOff) & (AAHState.toggle.effectAAHState = AAHOn)& (rotGrip.roll = POS) & ((tranGrip.XAcc = POS)|((tranGrip.YAcc = POS)|(tranGrip.ZAcc = POS)))) -> ((tranGrip.XAccEffect = ZERO)&((tranGrip.YAccEffect = ZERO)&(tranGrip.ZAccEffect = ZERO)))))

AAH disabled and Cruise enabled:

(A G (((cruiseState.effectCruiseState = cruiseOn) & (AAHState.toggle.effectAAHState = AAHOff)& (rotGrip.roll = POS) & ((tranGrip.XAcc = POS)|((tranGrip.YAcc = POS)|(tranGrip.ZAcc = POS)))) -> ((tranGrip.XAccEffect = ZERO)&((tranGrip.YAccEffect = ZERO)&(tranGrip.ZAccEffect = ZERO)))))

AAH and Cruise enabled:

(AG(((cruiseState.effectCruiseState = cruiseOn) & (AAHState.toggle.effectAAHState = AAHOn) & (rotGrip.roll = POS) & ((tranGrip.XAcc = POS)|((tranGrip.YAcc = POS)|(tranGrip.ZAcc = POS)))) -> ((tranGrip.XAccEffect = ZERO)&((tranGrip.YAccEffect = ZERO)&(tranGrip.ZAccEffect = ZERO)))))

Verification of Variabilities- Properties pertaining only to AAH are requirements on some but not all the products. Since the AAH feature is present in *AAH-SAFER* and *AAH-SAFER-Cruise*, these properties need to be verified only on these products. Similarly, the properties specific to the Cruise mode need to be verified only on *Base-SAFER-Cruise* and *AAH-SAFER-Cruise*. The need to verify similar properties on more than one product encouraged us to use property patterns to verify these variabilities.

- *Properties pertaining to cruise variability:* For example, one of the properties pertaining to cruise mode states that acceleration along any of the translational axes (X, Y, or Z) while the cruise mode is active, deactivates the cruise mode. We named this

property '*Cruise_on_to_off*' The verification of this property in Base-SAFER-Cruise and AAH-SAFER-Cruise is shown below.

✓ *Base-SAFER-Cruise*- The property specification in CMU-SMV for *Cruise_on_to_off* was written as

```
( AG ( ( ( cruiseState.engage = cruiseOn ) & ( !tranGrip.noTranCmd ) )
 -> ( AX ( cruiseState.engage = cruiseOff ) ) ) )
```

This property was saved as a pattern in a file named *cruise_variabilities_pattern* for reuse in *AAH-SAFER-Cruise*.

✓ *AAH-SAFER-Cruise*- AAH-SAFER-Cruise was modeled with the motive of separating the implementation of AAH mode and Cruise mode. As a result, the verification of the properties related to only AAH mode or Cruise mode did not require addition of any constraints to them. The pattern created for *Cruise_on_to_off* in Base-SAFER-Cruise was reused without any reinstantiation or modification. For reusing the pattern, we added the path to the pattern file *cruise_variabilities_pattern* in the model panel.

Similarly, another property pertaining to the cruise mode which is a variation of *Cruise_on_to_off* states that, if the cruise state is active, then it continues to be in the active state in the absence of a translational command. To verify this property, we reused the pattern created for *Cruise_on_to_off* and re-instantiated two atoms (!tranGrip.noTranCmd and cruiseState.engage = cruiseOff). The property remained the same for AAH-SAFER and AAH-SAFER-Cruise and the property is as follows.

```
(AG(((cruiseState.engage = cruiseOn)&(tranGrip.noTranCmd ) )
->(A X( cruiseState.engage = cruiseOn))))
```

Properties pertaining to AAH variability- The AAH feature for the SAFER product line was modeled exactly as it was done in the original SAFER model provided by Ben Di Vito. By following the same nomenclature of the original SAFER model for the SAFER product line, the patterns that we created for the original SAFER model during its evaluation were entirely used without any modification for verification of the AAH properties in AAH-SAFER and AAH-SAFER-Cruise. This significantly reduced the verification time as no new patterns nor properties had to be specified for the AAH features.

Analyses of SAFER Product line Verification Results- We evaluate the results of using FormulaEditor based on two aspects: reuse and change.

1. **Reuse**: The evaluation of the reusability aspect of FormulaEditor as applied to our proposed SAFER product line demonstrates the ease of specification and verification of properties due to reuse of property patterns. With a standard nomenclature for the product line models, property patterns created for property verification in one product could be reused for other products in the product line. As compared to property verification in the absence of FormulaEditor, FormulaEditor reduced the time and effort needed for specification of properties in product lines. The simple product line we proposed included only four products with two variabilities. Adding variabilities would likely increase the number of properties to be verified. In such situations, the usefulness of FormulaEditor would be experienced to a greater extent.

2. **Change**: A change in a product line can result from evolution of properties of the existing products in the product line or from evolution of the products themselves.

Property Evolution

Property evolution in product line involves:

- *Addition of new properties to the product line*: With time, new requirements may be identified for the product line. In product lines, such new requirements are many times very similar to the existing requirements of the product line. Property specification for these new requirements can utilize the features of property patterns and dynamic atom selection described in Chapter 3 to maximize reusability. If the additional requirements are similar to existing requirements, the property patterns for existing properties can be reused for specification of new properties. The existing property patterns can also be composed to form new patterns. The feature of dynamic atom selection allows efficient initialization of these patterns. Property patterns also allow the creation of patterns for these new requirements for a single product in the product line with reuse for the other products in the product line.
- *Deletion of existing properties from the product line*: With the passage of time, existing properties in the product line may become obsolete and may have to be discarded. Such a situation can occur for different reasons. One reason is the modification of the requirements of a particular product in the product line. In time, if it is decided that a particular feature is not required for a product in the product line, the model for that product can be changed and some of the earlier properties may become obsolete. FormulaEditor allows easy deletion of properties which do not have side-effects. Obsolete properties which do not have dependencies, or in other words, those which do not have any other properties depending on them, can be deleted from the product line. As FormulaEditor maps the properties to their underlying requirements, deletion of such properties is facilitated. However, deletion of such properties for the entire product line

would require manually removing the property from each of the property tables in the product line. This process can become labor-intensive in a large product line context.

- *Transformation of commonality to variability*: It is common that with time, a commonality is transformed into a variability. As the product line evolves, certain requirements which were common to all the products previously may no longer be a commonality for the future products. Such requirements may become variabilities for the new products. In such situations, the existence of property patterns reduces the complexity of the transformation of commonality to variability. We described in the previous sections that the efficient method of verification of commonalities is to use property patterns. In case of the new products where these previously common properties now become variabilities, the existing property patterns for the previously common properties can be reused to specify variabilities for the new products. The only factor that needs to be taken into consideration is the dependence of these common properties on other properties. If the previously common properties depend on other properties, then there can be two options to verify the variability for the new product. The first method is to verify both the properties and model them as variabilities. The second method is to create a new pattern for the new variability by composing the two old properties into a single property. In either of the cases, FormulaEditor allows efficient method for property specification by providing the features of property patterns and dynamic atom selection.
- *Transformation of variability to commonality*: As the product line evolves, a requirement which is a variability for the existing products may become a commonality for future products in the product line. In such situations, the requirement needs to be verified for each of the new products in the product line. If the variability was present in

many products, then we described in the previous chapters how FormulaEditor can reduce the specification and verification time by the use of property patterns and pattern files. These property patterns can be reused for verifying the commonality for the new products. If the variability was present in only a single product and resulted in very few, say one or two properties then we can understand that the property patterns would not result in reduced verification times. However, the use of property patterns to create the commonality and its reuse for the future products will reduce the specification and verification time in the future.

Product Evolution

A product line can also change either by the addition of a new product to the product line or by deleting/removing a product from the product line. Modification of the requirements of an existing product in the product line can be considered as the combination of removal of a product followed by the addition of a new product in the product line.

Addition of a product: Addition of a new product to the product line involves introduction of new variabilities to the product line or the use of combination of existing variabilities. Property patterns could be created for the variabilities that are predicted to be satisfied by other products in the product line. Our practical experience during the verification of SAFER product line demonstrated the use of patterns. The SAFER product line was initially designed to have three products namely, Base-SAFER, Base-SAFER-Cruise and AAH-SAFER. The addition of AAH-SAFER-Cruise to the product-line involved the verification of properties related to both AAH mode and Cruise mode.

The pattern files created for these properties were directly used to verify the properties without any additional specification effort being required.

Deletion of a product: Removing a product from a product line may result in the removal of certain variabilities which are only specific to this product. FormulaEditor separately verifies each of the products in the product line and at the same time reuses the patterns to reduce overhead. Due to this separation of verification, the removal of a product from the product line does not affect the other products as the variabilities in other products are verified using separate patterns. When a product is removed, just the pattern file associated with the variabilities in this product needs to be removed. This separation of verification and use of pattern files eases the task of tracking the variabilities for each product. For example, if we remove the cruise feature from the product line which results in the removal of Base-SAFER-Cruise and AAH-SAFER-Cruise from the product line, only the pattern file associated with the properties for Cruise mode needs to be deleted and the other products will continue to function as before.

CHAPTER 5. CONCLUSION AND FUTURE WORK

This work describes improvements to FormulaEditor, a tool-supported technique that facilitates the reuse of property specifications for model checking the members of a software product line. Reuse of property specifications avoids the overhead for specification of properties for every member of the product line. The previous version of the tool mapped the properties to 1) the underlying product line requirements, 2) the Cadence SMV models for the products, and 3) the verification results. The tool enables reuse of shared product line properties, as well as of product line-specific patterns of properties, while carefully preserving any distinctions among the product line members. It also manages the changes and re-verification needed as the product line evolves.

This work extends the previous version of the tool to allow verification of the members of the product line that are modeled in CMU-SMV in addition to Cadence-SMV. The work supports formal verification of product lines for legacy systems written in the CMU-SMV language. The improvements are tested on a proposed product line based on the original SAFER case study. Possible variations for the SAFER product line are suggested and the advantages of the improved FormulaEditor are tested on this product line.

Future work can include the extension of the tool to support additional model checkers such as the more recent NuSMV or SPIN. Also, the technique can be made more flexible by: 1) making the atom-extraction rules easier to modify so that users can change them at the time of specification; 2) investigating automatic instantiation of property patterns; 3) allowing properties specified elsewhere to be managed more easily by extending the property reuse management capability to allow clean interfaces with

other tools; and 4) allowing automatic generation of partially initialized patterns from given set of properties.

BIBLIOGRAPHY

[1]. Atkinson, C. et. al. Component-Based Product line Engineering with UML. Addison-Wesley, 2002.

[2]. Ben L. Di Vito. High-automation proofs for properties of requirements models. *Software Tools for Technology Transfer*, September 2000.

[3]. Bennet, K., Legacy Systems, *IEEE Software*, Jan 1995, 19-73.

[4]. Blazy, S., Gervais, F., Laleau, R. Reuse of Specification Patterns with the B Method, In *Proc. of ZB 2003: Formal Specification and Development in Z and B: Third International Conference of B and Z Users*, Turku, Finland, LNCS 2651, Springer Verlag, 2003, 40-57.

[5]. Bryant, R. E. 1986. Graph-based algorithms for Boolean function manipulation. IEEE Trans. Comput. C-35, 8, 677-691.

[6]. Burch, J. R, Clarke, E. M., McMillan, K. L., and Dill, D. L. 1990. Sequential circuit verification using symbolic model checking. In Proceedings of the 27th Design Automation Conference. IEEE Computer Society Press, Los Alamitos, California, 46-51.

[7]. Burch, J. R., Clarke, E. M., and Long, D. E. 1991. Representing circuits more efficiently in symbolic model checking. In proceedings of the 28th Design Automation Conference. IEEE Computer Society Press, Los Alamitos, California, 403-407.

[8]. Burch, J. R., Clarke, E. M., McMillan, K. L., Dill, D. L., Hwang, L. J. Symbolic Model checking: 10^{20} States and Beyond. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, June 1990.

[9]. Cadence SMV, Ken McMillan's homepage. http://www.kenmcmil.com/smv.html

[10]. Cadence SMV. http://www.ita.cs.ru.nl/publications/papers/biniam/smv/

[11]. Childs, A. et. Al. CALM and Cadena: Metamodeling for Component-Based Product line Development. *IEEE Computer, 39*, 2 (FEB. 2006), 42-50.

[12]. Clarke, E. M., and Emerson, E.A. 1981. Synthesis of synchronization skeletons for branching time temporal logic. In Logic of Programs: Workshop, New York. Springer-Verlag, New York.

[13]. Clarke, E.M, Grumberg, O., and Peled D.A. *Model checking*. The MIT Press, 2000.

[14]. Clarke, E.M., Emerson, E.A., and Sistla, A. P. 1983. Automatic verification of finite state concurrent systems using temporal logic specifications. In Proceedings of 10th Annual ACM Symposium on Principles of Programming Languages (Austin, Tx. Jan). ACM, New York, 117-126.

[15]. Clarke, E.M., Emerson, E.A., and Sistla, A. P. 1986. Automatic verification of finite state concurrent systems using temporal logic specifications. ACM Trans. Program. Lang. Syst. 8, 2 (April), 244-263.

[16]. Cleaveland, R. 1990. Tableau-based model checking in propositional mucalculus. Acta Inf. 27, 8 (Sept), 725-747.

[17]. Clements, P., and Northrop, L. Software Product lines. Boston: Addison-Wesley, 2002.

[18]. Doerr, B., and Sharp, D. 2000. Freeing Product line Architectures from Execution Dependencies. P. Donohoe ed., Proceedings Software Product line Conference, Kluwer Academic Publishers.

[19]. Dwyer, M.B., Avrunin, G. S., and Corbett, J.C. Patterns of property specifications for finite-state verification. In *Proc. of ICSE'99* (LA, USA, May 16-22, 1999). ACM Press, 1999, 411-420.

[20]. Extra Vehicular Activity. http://msis.jsc.nasa.gov/sections/section14.htm.

[21]. Farkash, et.al. Reuse-aware Property Specification. Property-Based System Design (PROSYD) deliverable 1. 1/2.

[22]. Formal methods specification and analysis guidebook for the verification of software and computer systems, Appendix C.

[23]. Gomaa, H. Designing Software Product lines with UML: From Use Cases to Pattern-Based Software Architectures. Addison-Wesley, 2005.

[24]. Havelund, K., Lowry, M., and Penix, J. Formal Analysis of a Space-Craft Controller using SPIN. *IEEE Trans. on Software Engineering, 27*, 8 (Aug. 2001), 749-765.

[25]. Heie, A. 2002. Global Software Product lines and Infinite Diversity. http://www.sei.cmu.edu/SPLC2/keynote_slides/keynote_1.htm

[26]. Holzmann, G.J., The Spin Model checker, *IEEE Trans. on Software Engineering, 23*, 1997, 279-295.

[27]. Huth, M., and Ryan, M. *Logic in Computer Science: Modeling and reasoning about systems*, 2nd ed., Cambridge University Press, 2004. [Online]. Available: http://pubs.doc.ic.ac.uk/logic-computer-science-second/

[28]. Kaivola, R. Formal Verification of Pentium Pentium® 4 Components with Symbolic Simulation and Inductive Invariants. In *Proc. of CAV 2005* (Edinburgh, UK, July 6-10, 2005). Springer 2005, 170-184.

[29]. Li, H., Krishnamurthi, S., and Fisler, K. Modular Verification of Open Features Using Three-Valued Model checking. *Automated SW Eng., 12,* 3 (July 2005), 349-382.

[30]. Lichtenstein, O., and Pnueli, A. 1985. Checking that finite state concurrent programs satisfy their linear specification. In Proceedings of the 12th Annual ACM Symposium on Principles of Programming Languages (New Orleans, LA. Jan). ACM, New York, 97-107.

[31]. Liu, J., Dehlinger, J., and Lutz, R. R. Safety Analysis of Software Product lines Using State-Based Modeling. *Journal of Systems and Software* 80(11):1879 - 1892, 2007.

[32]. Liu, J., Hauptman, M., Lutz, R. R, Geppert, B., Rößler, F., and Weiss, F. (2007). A Tool-supported Technique for Specification & Management of Model checking Properties for Software Product lines. *Technical Report* 08-05, Computer Science, Iowa State University.

[33]. McMillan, K. L. Symbolic model checking- an approach to the state explosion problem. PhD thesis, SCS, Carnegie Mellon University, 1992.

[34]. Model Checking Group at CMU. The SMV System. http://www.cs.cmu.edu/~modelcheck/smv.html

[35]. Pohl, K., Bockle, G., van der Linden, F. J. Software Product line Engineering: Foundations, Principles and Techniques. Springer, 2005.

[36]. Quielle, J., and Sifakis, J. 1981. Specification and verification of concurrent systems in CESAR. In Proceedings of the 5th International Symposium in Programming.

[37]. Robby, Dywer, M.B., and Hatcliff, J. Bogor: A Flexible Framework for Creating Software Model Checkers. In *Proc. Of Testing: Academia & Industry Conf. – Practice And Research Techniques (TAIC PART)* (Windsor, United Kingdom, Aug. 29-31, 2006), 3-22.

[38]. Schmid, K. and Verlage, M. 2002. The Economic Impact of Product line Adoption and Evolution. IEEE Software, 19(4): 50-57.

[39]. Sistla, A. P., and Clarke, E. 1986. Complexity of Propositional temporal logics. J. AC< 32, 3 (July), 733-749.

[40]. T. Kishi and N. Noda. Formal verification and software product lines. *Communication of the ACM* 49(12): 73-77, Dec. 2006.

[41]. Toft, P., Coleman, D. and Ohta, J. 2000. A Cooperative Model for Cross-Divisional Product Development for a Software Product line. P. Donohoe ed., Proceedings Software Product line Conference, Kluwer Academic Publishers.

[42]. Weiss, D.M, and Lai, C. T. R. Software Product line Engineering: A Family-Based Software Development Process. Addison-Wesley,1999.

[43]. Weiss, Li, Slye, Sun. Decision-Model-Based Code Generation for SPLE, International Software Product line Conference, 2008.

[44]. Wijinstra, J. 2002, Critical Factors for a Successful Platform-Based Product Family Approach. G. Chastek ed., Proceedings Software Product line Engineering Conference, Springer LNCS 2379.

APPENDIX

The appendix contains the following supplemental material:

1. CMU-SMV model for the original SAFER product
2. Property Set for the original SAFER product
3. CMU-SMV model for Base-SAFER
4. CMU-SMV model for Base-SAFER-Cruise
5. CMU-SMV model for AAH-SAFER
6. CMU-SMV model for AAH-SAFER-Cruise

CMU-SMV Model for the original SAFER product

This is the CMU-SMV model for the original SAFER product provided by Ben Di Vito [2]

```
MODULE main
VAR
  switches     : HCMSwitches;
  rotGrip      : rotCommand;
  AAHState     : AAHTransition(switches, rotGrip);
  allAxesOff : boolean;
 ASSIGN
  allAxesOff := AAHState.toggle.allAxesOff;
DEFINE
  maxTicks := AAHState.maxTicks;

MODULE buttonState(switches, active, timeout)
VAR
  engage: {AAHOff, AAHStarted, AAHOn,
         pressedOnce, AAHClosing, pressedTwice};
ASSIGN
  init(engage) := AAHOff;
  next(engage) := case
                 switches.AAH = buttonDown: downTransition;
```

```
            switches.AAH = buttonUp:   upTransition;
          esac;
DEFINE
 downTransition :=
    case engage = AAHOff:      AAHStarted;
       engage = AAHStarted:   AAHStarted;
       engage = AAHOn:       pressedOnce;
       engage = pressedOnce:  pressedOnce;
       engage = AAHClosing:   pressedTwice;
       engage = pressedTwice: pressedTwice;
    esac;

 upTransition :=
    case engage = AAHOff:      AAHOff;
       engage = AAHStarted:   AAHOn;
            engage = AAHOn:       stateA;
            engage = pressedOnce:  stateB;
            engage = AAHClosing:   stateB;
       engage = pressedTwice: AAHOff;
    esac;

 stateA := case allAxesOff: AAHOff; 1: AAHOn; esac;
 stateB := case timeout <= 0: AAHOn;
           1:          AAHClosing;
       esac;

allAxesOff := !(active.roll | active.pitch | active.yaw);


MODULE AAHTransition(switches, rotCmd)
VAR
 activeAxes: rotPredicate;
 ignoreHCM:  rotPredicate;
 toggle:     buttonState(switches, activeAxes, timeout);
 timeout:    0..100;
ASSIGN
 init(timeout) := 0;
 next(activeAxes.roll) := starting |
                  (!(next(toggle.engage) = AAHOff) &
                   activeAxes.roll &
                   (rotCmd.roll = ZERO | ignoreHCM.roll));
 next(activeAxes.pitch) := starting |
                  (!(next(toggle.engage) = AAHOff) &
                   activeAxes.pitch &
                   (rotCmd.pitch = ZERO | ignoreHCM.pitch));
 next(activeAxes.yaw) := starting |
                  (!(next(toggle.engage) = AAHOff) &
                   activeAxes.yaw &
```

```
                    (rotCmd.yaw = ZERO | ignoreHCM.yaw));

  next(ignoreHCM.roll) :=
     case starting: !(rotCmd.roll = ZERO);  1: ignoreHCM.roll; esac;
  next(ignoreHCM.pitch) :=
     case starting: !(rotCmd.pitch = ZERO); 1: ignoreHCM.pitch; esac;
  next(ignoreHCM.yaw) :=
     case starting: !(rotCmd.yaw = ZERO);   1: ignoreHCM.yaw; esac;

  next(timeout) :=
   case toggle.engage = AAHOn &
         next(toggle.engage) = pressedOnce : maxTicks;
       timeout > 0                    : timeout - 1;
       1                          : 0;
   esac;
DEFINE
  maxTicks := 100;

  starting := toggle.engage = AAHOff & next(toggle.engage) = AAHStarted;


MODULE HCMSwitches
VAR
  MODE: {ROT, TRAN};
  AAH:  {buttonUp, buttonDown};

MODULE rotCommand
VAR
  roll:  {NEG, ZERO, POS};
  pitch: {NEG, ZERO, POS};
  yaw:   {NEG, ZERO, POS};

MODULE rotPredicate
VAR
  roll:  boolean;
  pitch: boolean;
  yaw:   boolean;
ASSIGN
  init(roll)  := 0;
  init(pitch) := 0;
  init(yaw)   := 0;
```

Property set for Original SAFER model

This is the CTL property set for the original SAFER model provided by Ben Di Vito [2]

P1:
```
DEFINE
        AAH_stays_off :=
        AG (AAH_state.toggle.engage = AAH_off &
        switches.AAH = button_up ->
        AX AAH_state.toggle.engage = AAH_off);
```

P2:
```
DEFINE
        AAH_stays_on  :=
         AG ( ! all_axes_off &
         AAH_state.toggle.engage = AAH_on &
        switches.AAH = button_up                ->
        AX AAH_state.toggle.engage = AAH_on);
```

P3:
```
DEFINE
  pressed_down :=
    AG (  AAH_state.toggle.engage = AAH_started &
            switches.AAH = button_down       ->
          AX AAH_state.toggle.engage = AAH_started);
```

P4:
```
DEFINE
        starting_axes_on :=
          AG (AAH_state.toggle.engage = AAH_off &
            (AX AAH_state.toggle.engage = AAH_started)  ->
            AX (AAH_state.active_axes.roll & AAH_state.active_axes.pitch
             & AAH_state.active_axes.yaw));
```

P5:
```
DEFINE
        not_axes_off :=
          AG (AAH_state.toggle.engage = AAH_on &
            (AX AAH_state.toggle.engage = AAH_on)         ->
            ! all_axes_off);
```

P6:
```
DEFINE
  ignore_starting :=
          AG (! (AAH_state.toggle.engage = AAH_off &
        (AX AAH_state.toggle.engage = AAH_started)) &
        AAH_state.ignore_HCM.roll               ->
          AX AAH_state.ignore_HCM.roll);
```

P7:
```
DEFINE
  timeout_pressed_once :=
    AG (AAH_state.toggle.engage = AAH_on &
```

```
                (AX AAH_state.toggle.engage = pressed_once)          ->
                AX AAH_state.timeout = max_ticks);
```

P8:
```
DEFINE
 timeout_return :=
          AG ((AAH_state.toggle.engage = pressed_once |
           AAH_state.toggle.engage = AAH_closing) &
          AX AAH_state.toggle.engage = AAH_on  ->
            AAH_state.timeout <= 1);
```

P9:
```
DEFINE
          on_to_off_direct :=
           AG (AAH_state.toggle.engage = AAH_on &
              (AX AAH_state.toggle.engage = AAH_off)       ->
             all_axes_off);
```

P10:
```
DEFINE
axes_off_AAH := AG (AAH_state.toggle.engage = AAH_off -> all_axes_off);
```

P11:
```
DEFINE
 closing_before_timeout :=
          AG (AAH_state.toggle.engage = AAH_closing    -> AAH_state.timeout > 0);
```

P12:
```
DEFINE
 inactive_during_off :=
          AG (1 -> ! E [AAH_state.toggle.engage = AAH_off U
            !all_axes_off & AAH_state.toggle.engage = AAH_off]);
```

P13:
```
DEFINE
          rot_axis_stays_off_roll :=
  AG (AAH_state.toggle.engage = AAH_on & !AAH_state.active_axes.roll
            -> ! E [!AAH_state.toggle.engage = AAH_started U
           AAH_state.active_axes.roll &
           !AAH_state.toggle.engage = AAH_started]);
```

P14:
```
DEFINE
 ignore_HCM_stays_on_roll :=
          AG (AAH_state.toggle.engage = AAH_started &
              AAH_state.ignore_HCM.roll
            -> ! E [!AAH_state.toggle.engage = AAH_off U
           !AAH_state.ignore_HCM.roll &
           !AAH_state.toggle.engage = AAH_off]);
```

P15:
```
DEFINE
 ignore_HCM_stays_off_roll :=
  AG (AAH_state.toggle.engage = AAH_started &
            !AAH_state.ignore_HCM.roll
           -> ! E [!AAH_state.toggle.engage = AAH_off U
```

```
                AAH_state.ignore_HCM.roll &
                !AAH_state.toggle.engage = AAH_off]);
```

P16:
```
DEFINE
 ignore_stays_active_roll :=
   AG (AAH_state.toggle.engage = AAH_started &
                AAH_state.active_axes.roll & AAH_state.ignore_HCM.roll
     -> ! E [!AAH_state.toggle.engage = AAH_off U
     !(AAH_state.active_axes.roll & AAH_state.ignore_HCM.roll) &
     !AAH_state.toggle.engage = AAH_off]);
```

P17:
```
DEFINE
          closing_within_timeout :=
          AG (1 -> ! E [AAH_state.toggle.engage = AAH_closing U
            AAH_state.timeout = 0 &
            AAH_state.toggle.engage = AAH_closing]);
```

P18:
```
DEFINE
 on_to_off_path :=
          AG (AAH_state.toggle.engage = AAH_on
          -> ! E [!AAH_state.toggle.engage = AAH_off U
          AAH_state.toggle.engage = AAH_started &
          !AAH_state.toggle.engage = AAH_off]);
```

P19:
```
DEFINE
 closing_to_on_path :=
   AG (AAH_state.toggle.engage = AAH_closing
            -> ! E [!AAH_state.toggle.engage = AAH_on U
          AAH_state.toggle.engage = pressed_once &
          !AAH_state.toggle.engage = AAH_on]);
```

P20:
```
DEFINE
 off_to_closing_path :=
          AG (AAH_state.toggle.engage = AAH_off
           -> ! E [!AAH_state.toggle.engage = AAH_closing U
           AAH_state.toggle.engage = pressed_twice &
           !AAH_state.toggle.engage = AAH_closing]);
```

P21:
```
DEFINE
         AAH_started_exit :=
          AG (AAH_state.toggle.engage = AAH_started
            -> ! E [!switches.AAH = button_up U
           !(AX AAH_state.toggle.engage = AAH_started) &
           !switches.AAH = button_up]);
```

P22:
```
DEFINE
 pressed_once_exit :=
           AG (AAH_state.toggle.engage = pressed_once
```

```
                -> ! E [!switches.AAH = button_up U
                !(AX AAH_state.toggle.engage = pressed_once) &
                !switches.AAH = button_up]);
```

P23:
```
DEFINE
 pressed_twice_exit :=
  AG (AAH_state.toggle.engage = pressed_twice
           -> ! E [!switches.AAH = button_up U
          !(AX AAH_state.toggle.engage = pressed_twice) &
          !switches.AAH = button_up]);
```

P24:
```
DEFINE
 no_rot_no_ignore_roll :=
          AG (AAH_state.toggle.engage = AAH_off &
              (AX AAH_state.toggle.engage = AAH_started) &
              rot_grip.roll = ZERO
            -> ! E [!(AX AAH_state.toggle.engage = AAH_off) U
           (AX AAH_state.ignore_HCM.roll) &
           !(AX AAH_state.toggle.engage = AAH_off)]);
```

P25:
```
DEFINE
 rot_cmd_ignore_roll :=
          AG (AAH_state.toggle.engage = AAH_off &
             (AX AAH_state.toggle.engage = AAH_started) &
             !(rot_grip.roll = ZERO)
            -> ! E [!(AX AAH_state.toggle.engage = AAH_off) U
           !(AX AAH_state.ignore_HCM.roll) &
           !(AX AAH_state.toggle.engage = AAH_off)]);
```

P26:
```
DEFINE
 ignore_stays_on_starting_roll :=
  AG (AAH_state.toggle.engage = AAH_started &
      (AX AAH_state.ignore_HCM.roll)
           -> ! E [!(AAH_state.toggle.engage = AAH_off &
          (AX AAH_state.toggle.engage = AAH_started)) U
          !(AX AAH_state.ignore_HCM.roll) &
          !(AAH_state.toggle.engage = AAH_off &
           (AX AAH_state.toggle.engage = AAH_started))]);
```

P27:
```
DEFINE
        ignore_stays_off_starting_roll :=
          AG (AAH_state.toggle.engage = AAH_started &
           !(AX AAH_state.ignore_HCM.roll)
          -> ! E [!(AAH_state.toggle.engage = AAH_off &
           (AX AAH_state.toggle.engage = AAH_started)) U
          (AX AAH_state.ignore_HCM.roll) &
          !(AAH_state.toggle.engage = AAH_off &
           (AX AAH_state.toggle.engage = AAH_started))]);
```

P28: DEFINE

```
rot_cmd_inactive_roll :=
 AG (!(AAH_state.toggle.engage = AAH_off) &
  AAH_state.active_axes.roll &
   !AAH_state.ignore_HCM.roll &
    !(rot_grip.roll = ZERO)
  -> ! E [!(AX AAH_state.toggle.engage = AAH_off) U
  (AX AAH_state.active_axes.roll) &
  !(AX AAH_state.toggle.engage = AAH_off)]);
```

P29: DEFINE

```
active_until_rot_cmd_roll :=
 AG (!(AAH_state.toggle.engage = AAH_off) &
  AAH_state.active_axes.roll &
 !AAH_state.ignore_HCM.roll
-> ! E [!((AX AAH_state.toggle.engage = AAH_off) |
  !(rot_grip.roll = ZERO)) U
  !(AX AAH_state.active_axes.roll) &
  !((AX AAH_state.toggle.engage = AAH_off) |
   !(rot_grip.roll = ZERO))]);
```

P30: DEFINE

```
closing_path_duration :=
 AG (AAH_state.toggle.engage = AAH_on &
 (AX AAH_state.toggle.engage = pressed_once)
-> ! E [((AX AAH_state.toggle.engage = pressed_once) |
  (AX AAH_state.toggle.engage = AAH_closing)) U
  !((AX AAH_state.toggle.engage = AAH_closing)
   -> !(AX AAH_state.timeout = 0)) &
  ((AX AAH_state.toggle.engage = pressed_once) |
   (AX AAH_state.toggle.engage = AAH_closing))]);
```

CMU-SMV Model for Base-SAFER

```
MODULE HCMSwitches
VAR
 MODE: {ROT, TRAN};
 AAH:  {buttonUp, buttonDown};

MODULE rotCommand
VAR
 roll:  {NEG, ZERO, POS};
 pitch: {NEG, ZERO, POS};
 yaw:   {NEG, ZERO, POS};

MODULE tranCommand(rotGrip)
VAR
```

```
  XAcc:  {NEG, ZERO, POS};
  YAcc:  {NEG, ZERO, POS};
  ZAcc:  {NEG, ZERO, POS};
  XAccEffect:  {NEG, ZERO, POS};
  YAccEffect:  {NEG, ZERO, POS};
  ZAccEffect:  {NEG, ZERO, POS};
  noRotCmd : boolean;
ASSIGN
  XAccEffect :=
          case      !(noRotCmd) : ZERO;
                    noRotCmd : XAcc;
                    1                   : ZERO;
          esac;
  YAccEffect :=
          case      !(noRotCmd) : ZERO;
                    !(XAcc = ZERO) : ZERO;
                    1                   : YAcc;
          esac;
  ZAccEffect :=
          case  !(noRotCmd) : ZERO;
              !(XAcc = ZERO) | !(YAcc = ZERO) : ZERO;
              1                                 : ZAcc;
          esac;
  noRotCmd := (rotGrip.roll = ZERO) & (rotGrip.pitch = ZERO) & (rotGrip.yaw = ZERO);

MODULE main
VAR
  switches          : HCMSwitches;
  rotGrip           : rotCommand;
  tranGrip          : tranCommand(rotGrip);
```

CMU-SMV Model for Base-SAFER-Cruise

```
--
--Model file for Base-SAFER-Cruise
--

MODULE HCMSwitches

VAR
  MODE: {ROT, TRAN};
  cruise:  {buttonUp, buttonDown};


MODULE rotCommand

VAR
  roll:  {NEG, ZERO, POS};
```

```
  pitch: {NEG, ZERO, POS};
  yaw:   {NEG, ZERO, POS};


MODULE tranCommand(rotGrip, cruiseState)

VAR
  XAcc:  {NEG, ZERO, POS};
  YAcc:  {NEG, ZERO, POS};
  ZAcc:  {NEG, ZERO, POS};
  XAccEffect:  {NEG, ZERO, POS};
  YAccEffect:  {NEG, ZERO, POS};
  ZAccEffect:  {NEG, ZERO, POS};
  predictXAcc: {NEG, ZERO, POS};
  predictYAcc: {NEG, ZERO, POS};
  predictZAcc: {NEG, ZERO, POS};
  noRotCmd: boolean;
  noTranCmd: boolean;
  noCurStateTranCmd: boolean;

ASSIGN
  init(predictXAcc):= ZERO;
  init(predictYAcc):= ZERO;
  init(predictZAcc):= ZERO;
  next(predictXAcc):= XAccEffect;
  next(predictYAcc):= YAccEffect;
  next(predictZAcc):= ZAccEffect;

  XAccEffect :=
        case    (cruiseState.engage = cruiseOn) & noCurStateTranCmd & noRotCmd: predictXAcc;
                (cruiseState.engage = cruiseOn) & !(noRotCmd): ZERO;
                (cruiseState.engage = cruiseOn) & noRotCmd & !(noCurStateTranCmd): XAcc;
                !(cruiseState.engage= cruiseOn) & !(noRotCmd) : ZERO;
                !(cruiseState.engage= cruiseOn) & noRotCmd : XAcc;
                1                       : ZERO;
        esac;

  YAccEffect :=
        case    (cruiseState.engage = cruiseOn) & noCurStateTranCmd & noRotCmd: predictYAcc;
                (cruiseState.engage = cruiseOn) & !(noRotCmd): ZERO;
                (cruiseState.engage = cruiseOn) & noRotCmd &!(XAcc = ZERO): ZERO;
                (cruiseState.engage = cruiseOn) & noRotCmd & !(noCurStateTranCmd) & (XAcc = ZERO):
YAcc;
                !(cruiseState.engage= cruiseOn) & !(noRotCmd) : ZERO;
                !(cruiseState.engage= cruiseOn) & !(XAcc = ZERO) : ZERO;
                1                       : YAcc;
        esac;
```

```
 ZAccEffect :=
        case    (cruiseState.engage = cruiseOn) & noCurStateTranCmd & noRotCmd: predictZAcc;
                (cruiseState.engage = cruiseOn) & !(noRotCmd): ZERO;
                (cruiseState.engage = cruiseOn) & noRotCmd & (!(XAcc = ZERO) |!(YAcc = ZERO) ): ZERO;
                (cruiseState.engage = cruiseOn) & noRotCmd & !(noCurStateTranCmd) & (XAcc = ZERO)
& (YAcc = ZERO): ZAcc;
                !(cruiseState.engage= cruiseOn) & !(noRotCmd) : ZERO;
                !(cruiseState.engage= cruiseOn) & !(XAcc = ZERO) | !(YAcc = ZERO) : ZERO;
                1                                       : ZAcc;
        esac;

 noRotCmd       := (rotGrip.roll = ZERO) & (rotGrip.pitch = ZERO) & (rotGrip.yaw = ZERO);
 noTranCmd      := (XAccEffect = ZERO) & (YAccEffect = ZERO) & (ZAccEffect = ZERO);
 noCurStateTranCmd := (XAcc = ZERO) & (YAcc = ZERO) & (ZAcc = ZERO);


MODULE main

VAR
 switches       : HCMSwitches;
 rotGrip        : rotCommand;
 tranGrip       : tranCommand(rotGrip, cruiseState);
 cruiseState    : cruiseTransition(switches, rotGrip, tranGrip);


MODULE cruiseTransition(switches, rotGrip, tranGrip)

VAR
 engage: {cruiseOff, cruiseStarted, cruiseOn};
 --anyDirAcc: boolean;

ASSIGN
 init(engage)  := cruiseOff;
 next(engage)  := case
                  switches.cruise = buttonDown          : downTransition;
                  switches.cruise = buttonUp            : upTransition;
                  (engage = cruiseOn) & !(tranGrip.noTranCmd) : cruiseOff;
                 esac;

 DEFINE

 downTransition := case
                    engage = cruiseOff                  : cruiseStarted;
              engage = cruiseStarted                    : cruiseStarted;
              (engage = cruiseOn)& !(tranGrip.noTranCmd): cruiseOff;
                    1                                           : engage;
                  esac;
```

```
 upTransition := case
                engage = cruiseOff                                  :cruiseOff;
                      engage = cruiseStarted                        :cruiseOn;
                   (engage = cruiseOn) & !(tranGrip.noTranCmd):cruiseOff;
                      1                                             : engage;
                   esac;
```

CMU-SMV Model for AAH-SAFER

```
MODULE main
VAR
  switches     : HCMSwitches;
  rotGrip      : rotCommand;
  AAHState     : AAHTransition(switches, rotGrip);
  allAxesOff : boolean;
ASSIGN
 allAxesOff := AAHState.toggle.allAxesOff;
DEFINE
  maxTicks := AAHState.maxTicks;

MODULE buttonState(switches, active, timeout)
VAR
  engage: {AAHOff, AAHStarted, AAHOn,
         pressedOnce, AAHClosing, pressedTwice};
ASSIGN
  init(engage) := AAHOff;
  next(engage) := case
                 switches.AAH = buttonDown: downTransition;
                 switches.AAH = buttonUp:   upTransition;
                esac;
DEFINE
   downTransition :=
      case engage = AAHOff:       AAHStarted;
           engage = AAHStarted:   AAHStarted;
           engage = AAHOn:        pressedOnce;
           engage = pressedOnce:  pressedOnce;
           engage = AAHClosing:   pressedTwice;
           engage = pressedTwice: pressedTwice;
      esac;
  upTransition :=
      case engage = AAHOff:       AAHOff;
           engage = AAHStarted:   AAHOn;
                engage = AAHOn:       stateA;
                engage = pressedOnce:  stateB;
                engage = AAHClosing:   stateB;
```

```
        engage = pressedTwice: AAHOff;
    esac;
 stateA := case allAxesOff: AAHOff; 1: AAHOn; esac;
 stateB := case timeout <= 0: AAHOn;
           1:          AAHClosing;
        esac;
allAxesOff := !(active.roll | active.pitch | active.yaw);

MODULE AAHTransition(switches, rotCmd)
VAR
 activeAxes: rotPredicate;
 ignoreHCM:  rotPredicate;
 toggle:     buttonState(switches, activeAxes, timeout);
 timeout:    0..100;

ASSIGN
 init(timeout) := 0;

 next(activeAxes.roll) := starting |
                    (!(next(toggle.engage) = AAHOff) &
                     activeAxes.roll &
                     (rotCmd.roll = ZERO | ignoreHCM.roll));
 next(activeAxes.pitch) := starting |
                    (!(next(toggle.engage) = AAHOff) &
                     activeAxes.pitch &
                     (rotCmd.pitch = ZERO | ignoreHCM.pitch));
 next(activeAxes.yaw) := starting |
                    (!(next(toggle.engage) = AAHOff) &
                     activeAxes.yaw &
                     (rotCmd.yaw = ZERO | ignoreHCM.yaw));

 next(ignoreHCM.roll) :=
   case starting: !(rotCmd.roll = ZERO);  1: ignoreHCM.roll; esac;
 next(ignoreHCM.pitch) :=
   case starting: !(rotCmd.pitch = ZERO); 1: ignoreHCM.pitch; esac;
 next(ignoreHCM.yaw) :=
   case starting: !(rotCmd.yaw = ZERO);   1: ignoreHCM.yaw; esac;

 next(timeout) :=
  case toggle.engage = AAHOn &
       next(toggle.engage) = pressedOnce : maxTicks;
      timeout > 0                        : timeout - 1;
      1                             : 0;
  esac;
DEFINE
 maxTicks := 100;
 starting := toggle.engage = AAHOff & next(toggle.engage) = AAHStarted;
```

```
MODULE HCMSwitches
VAR
  MODE: {ROT, TRAN};
  AAH:  {buttonUp, buttonDown};

MODULE rotCommand
VAR
  roll:  {NEG, ZERO, POS};
  pitch: {NEG, ZERO, POS};
  yaw:   {NEG, ZERO, POS};

MODULE tranCommand(rotGrip)
VAR
  XAcc:  {NEG, ZERO, POS};
  YAcc:  {NEG, ZERO, POS};
  ZAcc:  {NEG, ZERO, POS};
  XAccEffect:  {NEG, ZERO, POS};
  YAccEffect:  {NEG, ZERO, POS};
  ZAccEffect:  {NEG, ZERO, POS};
  noRotCmd: boolean;
  noTranCmd: boolean;
ASSIGN
  XAccEffect :=
          case      !(noRotCmd) : ZERO;
                    !(XAcc = ZERO) & noRotCmd : XAcc;
                    1                   : ZERO;
          esac;

  YAccEffect :=
          case      !(noRotCmd) : ZERO;
                    !(XAcc = ZERO) : ZERO;
                    1                   : YAcc;
          esac;
  ZAccEffect :=
          case  !(noRotCmd) : ZERO;
              !(XAcc = ZERO) | !(YAcc = ZERO) : ZERO;
              1                                 : ZAcc;
          esac;
  noRotCmd      := (rotGrip.roll = ZERO) & (rotGrip.pitch = ZERO) & (rotGrip.yaw = ZERO);
  noTranCmd     := (XAccEffect = ZERO) & (YAccEffect = ZERO) & (ZAccEffect = ZERO);

MODULE rotPredicate
VAR
  roll:  boolean;
  pitch: boolean;
  yaw:   boolean;
ASSIGN
  init(roll)  := 0;
```

```
 init(pitch) := 0;
 init(yaw)   := 0;
```

CMU-SMV Model for AAH-SAFER-Cruise

```
MODULE main
VAR
 AAHSwitches    : AAHSwitches;
 cruiseSwitches  : cruiseSwitches;
 rotGrip          : rotCommand;
 tranGrip         : tranCommand(rotGrip, cruiseState);
 AAHState    : AAHTransition(AAHSwitches, rotGrip);
 cruiseState  : cruiseTransition(cruiseSwitches, rotGrip, tranGrip);
 allAxesOff : boolean;

ASSIGN
 allAxesOff := AAHState.toggle.allAxesOff;

DEFINE
 maxTicks := AAHState.maxTicks;

MODULE buttonState(switches, active, timeout)
VAR
 engage: {AAHOff, AAHStarted, AAHOn,
       pressedOnce, AAHClosing, pressedTwice};
 effectAAHState : {AAHOn, AAHOff};

ASSIGN
 init(engage) := AAHOff;
 next(engage) := case
             switches.AAH = buttonDown: downTransition;
             switches.AAH = buttonUp:   upTransition;
            esac;
 effectAAHState := case
                    engage = AAHStarted |engage = AAHOn |engage = pressedOnce |engage =
AAHClosing |engage = pressedTwice : AAHOn;
                    engage = AAHOff : AAHOff;
                  esac;
DEFINE
 downTransition :=
     case engage = AAHOff:      AAHStarted;
        engage = AAHStarted:   AAHStarted;
        engage = AAHOn:       pressedOnce;
        engage = pressedOnce:  pressedOnce;
        engage = AAHClosing:   pressedTwice;
        engage = pressedTwice: pressedTwice;
     esac;
```

```
 upTransition :=
    case engage = AAHOff:      AAHOff;
       engage = AAHStarted:   AAHOn;
            engage = AAHOn:       stateA;
            engage = pressedOnce:  stateB;
            engage = AAHClosing:   stateB;
       engage = pressedTwice: AAHOff;
    esac;

 stateA := case allAxesOff: AAHOff; 1: AAHOn; esac;
 stateB := case timeout <= 0: AAHOn;
           1:          AAHClosing;
       esac;

allAxesOff := !(active.roll | active.pitch | active.yaw);

MODULE AAHTransition(switches, rotCmd)
VAR
 activeAxes: rotPredicate;
 ignoreHCM:  rotPredicate;
 toggle:     buttonState(switches, activeAxes, timeout);
 timeout:    0..100;

ASSIGN
 init(timeout) := 0;
 next(activeAxes.roll) := starting |
                    (!(next(toggle.engage) = AAHOff) &
                     activeAxes.roll &
                     (rotCmd.roll = ZERO | ignoreHCM.roll));
 next(activeAxes.pitch) := starting |
                    (!(next(toggle.engage) = AAHOff) &
                     activeAxes.pitch &
                     (rotCmd.pitch = ZERO | ignoreHCM.pitch));
 next(activeAxes.yaw) := starting |
                    (!(next(toggle.engage) = AAHOff) &
                     activeAxes.yaw &
                     (rotCmd.yaw = ZERO | ignoreHCM.yaw));

 next(ignoreHCM.roll) :=
   case starting: !(rotCmd.roll = ZERO);  1: ignoreHCM.roll; esac;
 next(ignoreHCM.pitch) :=
   case starting: !(rotCmd.pitch = ZERO); 1: ignoreHCM.pitch; esac;
 next(ignoreHCM.yaw) :=
   case starting: !(rotCmd.yaw = ZERO);   1: ignoreHCM.yaw; esac;

 next(timeout) :=
  case toggle.engage = AAHOn &
       next(toggle.engage) = pressedOnce : maxTicks;
```

```
      timeout > 0                    : timeout - 1;
      1                          : 0;
   esac;

DEFINE
  maxTicks := 100;
  starting := toggle.engage = AAHOff & next(toggle.engage) = AAHStarted;

MODULE AAHSwitches
VAR
  MODE: {ROT, TRAN};
  AAH:  {buttonUp, buttonDown};

MODULE rotCommand
VAR
  roll:  {NEG, ZERO, POS};
  pitch: {NEG, ZERO, POS};
  yaw:   {NEG, ZERO, POS};



MODULE rotPredicate
VAR
  roll:  boolean;
  pitch: boolean;
  yaw:   boolean;

ASSIGN
  init(roll)  := 0;
  init(pitch) := 0;
  init(yaw)   := 0;



MODULE cruiseTransition(cruiseSwitches, rotGrip, tranGrip)

VAR
  engage: {cruiseOff, cruiseStarted, cruiseOn};
  effectCruiseState : {cruiseOff, cruiseOn};

ASSIGN
  init(engage)  := cruiseOff;
  next(engage)  := case
                     cruiseSwitches.cruise = buttonDown   : downTransition;
                     cruiseSwitches.cruise = buttonUp       : upTransition;
                     (engage = cruiseOn) & !(tranGrip.noTranCmd) : cruiseOff;
                   esac;
  effectCruiseState := case
```

```
                    engage = cruiseStarted | engage =cruiseOn : cruiseOn;
                    engage = cruiseOff: cruiseOff;
                    esac;

DEFINE

 downTransition := case
                     engage = cruiseOff                    : cruiseStarted;
              engage = cruiseStarted                       : cruiseStarted;
              engage = cruiseOn    & !(tranGrip.noTranCmd): cruiseOn;
                     1                                     : engage;
                   esac;

 upTransition := case
              engage = cruiseOff                           :cruiseOff;
                     engage = cruiseStarted                :cruiseOn;
                engage = cruiseOn          & !(tranGrip.noTranCmd):cruiseOn;
                     1                                     : engage;
                  esac;




MODULE tranCommand(rotGrip, cruiseState)
VAR
 XAcc:  {NEG, ZERO, POS};
 YAcc:  {NEG, ZERO, POS};
 ZAcc:  {NEG, ZERO, POS};
 XAccEffect:  {NEG, ZERO, POS};
 YAccEffect:  {NEG, ZERO, POS};
 ZAccEffect:  {NEG, ZERO, POS};
 predictXAcc: {NEG, ZERO, POS};
 predictYAcc: {NEG, ZERO, POS};
 predictZAcc: {NEG, ZERO, POS};
 noTranCmd : boolean;
 noRotCmd : boolean;
 noCurStateTranCmd: boolean;
ASSIGN
 init(predictXAcc):= ZERO;
 init(predictYAcc):= ZERO;
 init(predictZAcc):= ZERO;
 next(predictXAcc):= XAccEffect;
 next(predictYAcc):= YAccEffect;
 next(predictZAcc):= ZAccEffect;

  XAccEffect :=
         case     (cruiseState.engage = cruiseOn) & noCurStateTranCmd & noRotCmd: predictXAcc;
```

```
                (cruiseState.engage = cruiseOn) & !(noRotCmd): ZERO;
                (cruiseState.engage = cruiseOn) & noRotCmd & !(noCurStateTranCmd): XAcc;
                !(cruiseState.engage= cruiseOn) & !(noRotCmd) : ZERO;
                !(cruiseState.engage= cruiseOn) & noRotCmd : XAcc;
                1                       : ZERO;
        esac;

 YAccEffect :=
        case    (cruiseState.engage = cruiseOn) & noCurStateTranCmd & noRotCmd: predictYAcc;
                (cruiseState.engage = cruiseOn) & !(noRotCmd): ZERO;
                (cruiseState.engage = cruiseOn) & noRotCmd &!(XAcc = ZERO): ZERO;
                (cruiseState.engage = cruiseOn)&noRotCmd&!(noCurStateTranCmd)&(XAcc=ZERO):
YAcc;
                !(cruiseState.engage= cruiseOn) & !(noRotCmd) : ZERO;
                !(cruiseState.engage= cruiseOn) & !(XAcc = ZERO) : ZERO;
                1                       : YAcc;
        esac;
 ZAccEffect :=
        case    (cruiseState.engage = cruiseOn) & noCurStateTranCmd & noRotCmd: predictZAcc;
                (cruiseState.engage = cruiseOn) & !(noRotCmd): ZERO;
                (cruiseState.engage = cruiseOn) & noRotCmd & (!(XAcc = ZERO) | !(YAcc = ZERO) ):
ZERO;
                (cruiseState.engage = cruiseOn) & noRotCmd & !(noCurStateTranCmd) & (XAcc = ZERO)
& (YAcc = ZERO): ZAcc;
                !(cruiseState.engage= cruiseOn) & !(noRotCmd) : ZERO;
            !(cruiseState.engage= cruiseOn)     & !(XAcc = ZERO) | !(YAcc = ZERO) : ZERO;
            1                                   : ZAcc;                     : ZAcc;
        esac;

 noRotCmd       := (rotGrip.roll = ZERO) & (rotGrip.pitch = ZERO) & (rotGrip.yaw = ZERO);
 noTranCmd      := (XAccEffect = ZERO) & (YAccEffect = ZERO) & (ZAccEffect = ZERO);
 noCurStateTranCmd := (XAcc = ZERO) & (YAcc = ZERO) & (ZAcc = ZERO);


MODULE cruiseSwitches
VAR
 cruise:  {buttonUp, buttonDown};
```