AGVTS: Automated Generation and Verification of Temporal Specifications for Aeronautics SCADE Models *

Hanfeng Wang[1], Zhibin Yang(✉)[1][0000−0002−9888−6975], Yong Zhou[1], Xilong Wang[1], Weilin Deng[1], and Wei Li[2]

[1] College of Computer Science and Technology,
Nanjing University of Aeronautics and Astronautics, Nanjing, China
yangzhibin168@163.com
[2] Aerospace Life-support Industries Ltd, Xiangyang, China

Abstract. SCADE is both a formal language and a model-based development environment, widely used to build and verify the models of safety-critical system (SCS). The SCADE Design Verifier (DV) provides SAT-based verification. However, DV cannot adequately express complex temporal specifications, and it may fail due to complexity problems such as floating numbers which are often used in the aeronautics domain. In addition, manually writing temporal specifications is not only time-consuming but also error-prone. To address these challenges, we propose an AGVTS method that can automate the task of generating temporal specifications and verifying aeronautics SCADE models. At first, we define a modular pattern language for precisely expressing Chinese natural language requirements. Then, we present a rule-based translation augmented with BERT, which translates restricted requirements into LTL and CTL. In addition, SCADE model verification is achieved by transforming it into nuXmv which supports both SMT-based and SAT-based verification. Finally, we illustrate a successful application of our methodology with an ejection seat control system, and convince our industrial partners of the usefulness of formal methods for industrial systems.

Keywords: SCADE · Temporal Specification · Pattern-based Language · BERT · nuXmv

1 Introduction

Safety-critical systems (SCS) [20] are the systems whose failure could result in loss of life, substantial economic loss, or damage to the environment. There are many well-known examples in different domains such as avionics, nuclear plants, transportation, and automotive. Formal verification is highly recommended by safety standards, e.g., DO-178C for the avionics domain, in order to ensure the

* Supported by the National Natural Science Foundation of China (62072233, U2241216), Aviation Science Fund of China (201919052002)

safety of this kind of systems [17]. SCADE is both a formal language [9] and a model-based development environment[3], widely used to build and verify the models of safety-critical system. SCADE provides three modeling styles, i.e., safety state machines, data flow and their combination.

Design Verifier (DV)[4], the formal verification module of SCADE, is a model checker based on a SAT-solver. However, DV cannot adequately express complex temporal specifications [24]. Temporal logics are popular methods for describing complex temporal properties, such as LTL [23], CTL [7] and TCTL [1], etc. There are several related works [12, 24] to enhance the verification capability of SCADE, which respectively transform SCADE models into UPPAAL and nuSmv to verify temporal properties. However, these works only consider safety state machine models. Additionally, DV may fail due to complexity problems such as floating numbers which are common in the aeronautics domain.

Moreover, it is always a challenge to manually translate natural language requirements into temporal logic formulae. As natural language is generally informal and ambiguous, this process is error-prone and time-consuming. Existing works on translating natural language requirements into temporal logics can be classified as several categories, such as rule-based [15, 29], deep learning [18], and Large Language Models (LLMs) [11]. However, these methods require either manual writing of formal specifications for atomic propositions, or utilizing plenty of patterns, or training with plenty of data. To the best of our knowledge, these works always consider the translation of English, but few consider Chinese, and there is little work focusing on the ejection seat control system domain.

To address the challenges above, we propose an AGVTS method, automatically generating and verifying temporal specifications for aeronautics SCADE models. The main contributions are summarized as follows:

(1) A modular pattern language (MPL) to precisely express Chinese natural language requirements. Benefiting from the modular structure, users can write requirements in a restricted and composite way with less patterns.
(2) A rule-based method augmented with BERT for automatically translating requirements expressed by MPL into LTL and CTL formulae.
(3) An automated transformation from SCADE to nuXmv that provides SMT-based and SAT-based model checking techniques to verify LTL and CTL properties with floating numbers. Compared with existing works, our transformation covers more modeling styles, such as data flow, safety state machines and their combination.
(4) We apply our method to an industrial ejection seat control system. It successfully translates 124 requirements and verifies the SCADE models of six modules of the system. The result convinces our industrial partners of the usefulness of formal methods to industrial systems.

[3] Ansys SCADE Suite https://www.ansys.com/products/embedded-software/ansys-scade-suite
[4] DV is based on Prover Technology proof engines (www.prover.com)

2 Global View of the AGVTS Method

Fig.1 gives an overview of the AGVTS method. AGVTS has three modules shown as follows.

- **Modular Pattern Language (MPL)**: Define a pattern language organized in a modular structure. The syntax and semantics of each pattern guide users to write requirements in a restricted and composite manner.
- **Rule-based Requirements Translator Augmented with BERT**: Parse requirements written in MPL and build pattern structure trees for them. Then generate LTL and CTL formulae through traversing the tree.
- **SCADE2nuXmv Model Transformation**: Transform SCADE models into nuXmv. Subsequently, verify the nuXmv models, and show the verification results and the traceability between requirements and counterexamples.

In the following sections, we will introduce the three modules in detail.

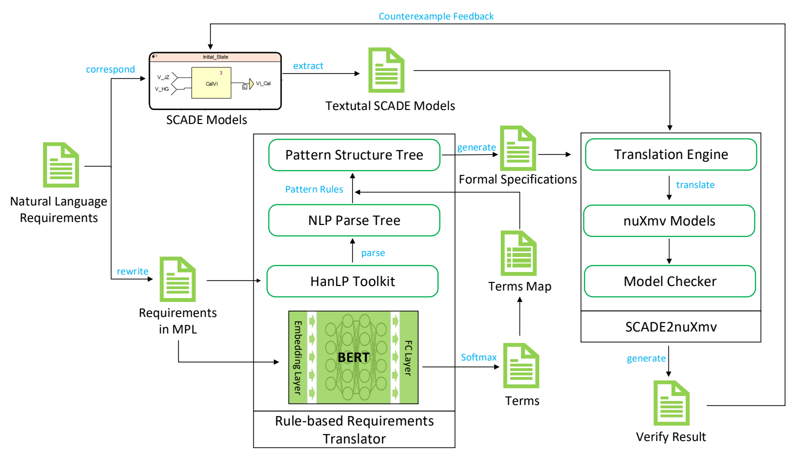

Fig. 1. The Global View of the AGVTS Method

3 Modular Pattern-based Language

The modular pattern-based language (MPL) focuses on ejection seat control system which usually contains complex computation in the requirements. MPL has three segments: *Atomic Proposition* (*AP*), *Relation*, and *Scope*. Each segment has several patterns. This feature allows users to write requirements in a composite manner with fewer patterns. Appendix 1, 2 and 3 show the formal syntax and semantics of the patterns. As shown in Fig.2, we first write three atomic statements in *AP* patterns. Then the *Relation* patterns are utilized to connect statements for constructing compound statements and *Scope* patterns are added to declare the effective extent of different statements. Finally we obtain the complete requirement. Please note that the *Relation* and *Scope* patterns can be nested in any way.

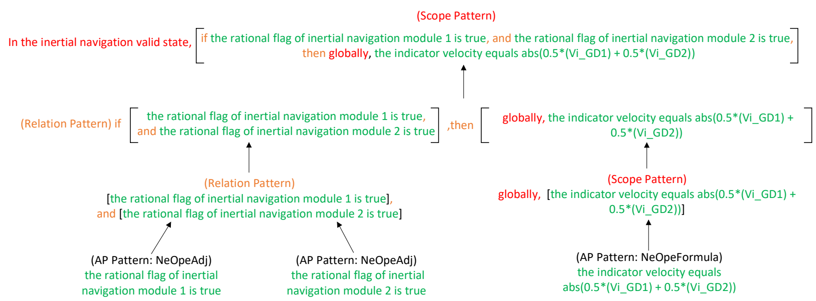

Fig. 2. Writing Requirements in a Composite Manner

***AP* Patterns** *AP* patterns form the basis of MPL, serving to specify an atomic statement on an event or a state of the system. As shown in Appendix 1, MPL has nine *AP* patterns. Each pattern consists of several *Ingredient* tags we define, such as *Ne* for variables and constants, *Ope* for operators, and *Formula* for complex computation. We have defined seven *Ingredient* tags in total. For Instance, in Fig.2, "the indicator velocity equals $abs(0.5 * (Vi\_GD1) + 0.5 * (Vi\_GD1))$" is written in "*Ne Ope Formula*" pattern, in which "rational flag of inertial navigation module 1", "equals" and "$abs(0.5 * (Vi\_GD1) + 0.5 * (Vi\_GD1))$" are labeled as *Ne*, *Ope*, and *Formula* respectively.

***Relation* Patterns** *Relation* patterns describe the temporal or logical relations between atomic or compound statements, which are characterized by different keywords. We have defined six *Relation* patterns: *Simple*, *Response*, *Condition*, *Precedence*, *Conjunction* and *Disjunction*, as shown in Table 1 where ϕ_i $(1 \leq i \leq n)$ denotes a statement. The *Simple* pattern represents the atomic statements in the requirement, which are specified by the *AP* patterns. The *Response* and *Precedence* patterns are introduced from [2]. *Conjunction* and *Disjunction* patterns express the complex nesting relations of a series of statements, as the parentheses are seldom used in Chinese requirements. To support the nesting of *Relation* pattern, we define a priority for each of them.

For example, in Fig.2, we use *Conjunction* pattern to connect two atomic statements. The compound statement "the rational flag of inertial navigation module 1 is true and the rational flag of inertial navigation module 2 is true" indicates that the rational flags of both inertial navigation modules must be true simultaneously. Moreover, the statement "ϕ_1 weak_and ϕ_2, or ϕ_3 weak_and ϕ_4" can express "$(\phi_1 \wedge \phi_2) \vee (\phi_3 \wedge \phi_4)$, based on different priorities.

Table 1. Relation Patterns

Pattern Name	Natural Language Format	Meaning	Priority
Simple	ϕ_1	the atomic statement	6
Response	if ϕ_1 holds, then in response ϕ_2 holds	if ϕ_1 holds, then ϕ_2 must holds in the next cycle	1
Condition	if ϕ_1, then ϕ_2	if ϕ_1 holds, then ϕ_2 must holds in the same cycle	0
Precedence	ϕ_2 precedes ϕ_1	if ϕ_1 holds in the future, ϕ_2 must hold at least one time before ϕ_1 holds	1
Conjunction	ϕ_1 and ϕ_2 and $\cdots$ and ϕ_n	all of ϕ_i $(1 \leq i \leq n)$ must hold in the specified cycle	2
	ϕ_1 weak_and ϕ_2 weak_and $\cdots$ weak_and ϕ_n		4
Disjunction	ϕ_1 or ϕ_2 or $\cdots$ or ϕ_n	at least one of ϕ_i $(1 \leq i \leq n)$ holds in the specified cycle	3
	ϕ_1 weak_or ϕ_2 weak_or $\cdots$ weak_or ϕ_n		5

***Scope* Patterns** *Scope* patterns describe the effective extent of an atomic or compound statement. As shown in Appendix 3, we have defined 10 *Scope* patterns, such as "Globally", "In x state", and "Every n cycles", where x denotes a state name and n denotes a positive integer. As shown in Fig.2, the scope "In the inertial navigation valid state" expresses that the property stated subsequently must hold when the system is in the "inertial navigation valid" state.

4 Rule-based Translation Augmented with BERT

In this section, we will introduce the rule-based translation augmented with BERT method which translates requirements into LTL and CTL specifications. As shown in Fig.3, to illustrate the workflow of the method, we consider the following requirement:

Example 1. Globally, if the state of the system is calculating angle by two inertial navigation modules, then the input angle always equals 0.5 times the sum of the angles in inertial modules 1 and 2.

In the following paragraphs, we will represent the details of each step.

Rewriting & Pre-processing Natural language are usually ambiguous, so we rewrite the requirement in MPL first. Additionally, the complex calculations expressed in natural language should be replaced with corresponding formal expression to simplify the translation. For example, the statement "0.5 times the sum of the angles in inertial modules 1 and 2" in *Example 1* is replaced by "$(0.5 * (angle\_GD1 + angle\_GD2))$". The *Example 1* is rewritten as "Globally, if the state equals calculate angle by two inertial navigation modules state, then globally, the input angle equals $(0.5 * (angle\_GD1 + angle\_GD2))$".

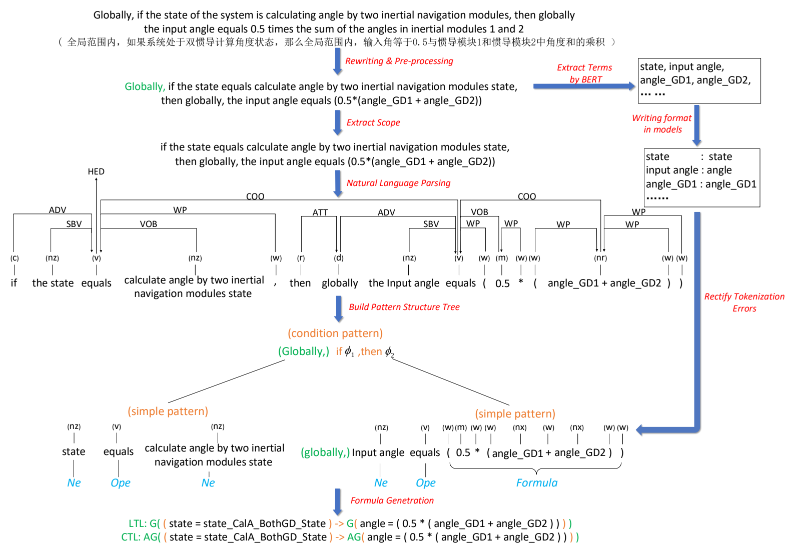

Fig. 3. Overview of the Rule-Based Translation Augmented with BERT

Extract *Scopes* & Natural Language Parsing The accuracy of a NLP parser decreases as the length of the requirement increases, so the requirement should be as concise as possible. Since the expressions of *Scope* patterns in MPL are fixed, we extract them with regular expression before parsing the rewritten requirement. For example, we extract "*Globally,*" from *Example 1*. Then HanLP[5], a Chinese NLP toolkit, is leveraged to parse the requirement, including tokenization, POS tagging, and dependency relations analysis.

Extracting Terms by BERT Compared with English, Chinese natural language lacks separators to distinguish words. Therefore, tokenization is the initial task when parsing Chinese with NLP techniques. However, incorrect tokenization may occur. Clearly, an incorrect tokenization will finally lead to a wrong parsing result. For example, "$angle\_GD1 + angle\_GD2$" in *Example 1* is recognized as one token.

To solve this problem, we implement a *Term Extractor* to extract terms from requirements, which assists in improving the accuracy of word segmentation and correcting its results. Term extraction essentially classifies each token in the requirement into three categories: the beginning of the term, the body of the term, and non-term. Therefore, we build a deep learning model, as shown in Fig.4, to complete this task. This model first utilizes BERT [13], a pre-trained language model, to extract text features from the requirements. Then a fully connected layer (FC Layer) is utilized to calculate the probability of each token

[5] https://github.com/hankcs/HanLP/

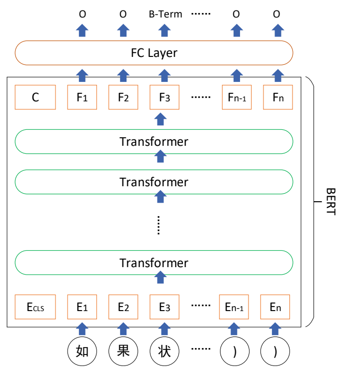

Fig. 4. The Structure of the BERT-Based Model

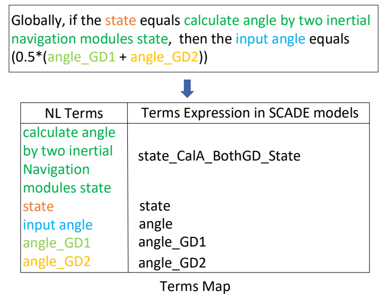

Fig. 5. Examples of a Terms Map

belonging to different categories based on the text features. Finally, each token is labeled as the category with the highest probability.

The extracted terms serve two purposes. One is to expand the tokenization lexicon of HanLP, which reduces the probability of incorrect tokenization. The other is to rectify the potential errors in the parsing result of HanLP. In addition, we construct a *terms map* in order to match the extracted terms with their corresponding variables or constants in the SCADE model, as shown in Fig.5.

Build Pattern Structure Tree & Rectify NLP Errors After getting the NLP parsing result, we recursively construct a pattern structure tree for each statement in the requirement. The whole pattern structure tree describes the nesting relations among the *Relation* patterns present in the requirement. Each node of the tree corresponds to a statement in the requirement, including the *Relation* pattern of the statement, the keywords of the *Relation* pattern, and the *Scope* pattern of the statement. Please note that each leaf node of this tree corresponds to an atomic statement in the requirement.

Each *Relation* pattern recorded by the tree node represents the one with the highest priority in its corresponding statement. To determine the *Relation* pattern with the highest priority, we utilize tokenization and lexical matching to find the keywords of *Relation* patterns. Then we compare the priorities of the patterns they belong to, as shown in Table 2.

In cases where no keywords are identified, the pattern of the statement is the *Simple* pattern. Then we identify the *AP* pattern of the statement by determining the *Ingredient* tag of each token and combining the tags in order. The task is performed by tokenization, lexical matching and POS tagging. Additionally, we utilize domain lexicons which include verbs, adjectives, operators and functions in the requirement to filter useless tokens. The extracted terms are leveraged to enhance the NLP result in this process. That is, for tokens categorized as "noun", we substitute the original token with the term that matches the longest consecutive characters in the requirement.

As shown in Fig.3, the *Condition* pattern recorded by the root node of the pattern structure tree has the highest priority in *Example 1*. Both of its children are atomic statements, so the corresponding nodes of these statements record their AP and *Scope* patterns. In this process, the incorrect token "$angle\_GD1 + angle\_GD2$" is rectified to "$angle\_GD1$", "$angle\_DD2$" and the operator "+".

Formula Generation The last step is generating formal specifications by composing formal expressions of each node on the pattern structure tree in post order. The mapping of our *Relation* patterns to LTL and CTL are shown in Table 2, and the mapping rules of AP patterns and *Scope* patterns are illustrated in Appendix 1, 2 and 3. These rules strictly follow the formal semantics of MPL.

Table 2. Mapping Rules of *Relation* Patterns (Except for *Simple* Pattern)

Pattern Name	LTL Formula	CTL Formula
Response	$\phi_1 \rightarrow X\,\phi_2$	$\phi_1 \rightarrow AX\,\phi_2$
Condition	$\phi_1 \rightarrow \phi_2$	$\phi_1 \rightarrow \phi_2$
Precedence	$F\,\phi_1 \rightarrow (!\,\phi_1\,U\,(\phi_2 \wedge !\,\phi_1))$	$AF\,\phi_1 \rightarrow A(!\,\phi_1\,U\,(\phi_2 \wedge !\,\phi_1))$
Conjunction	$\phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n$	$\phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n$
Disjunction	$\phi_1 \vee \phi_2 \vee \cdots \vee \phi_n$	$\phi_1 \vee \phi_2 \vee \cdots \vee \phi_n$

Based on the mapping rules above, as shown in Fig.3, we first translate the leaf nodes on the pattern structure tree into AP formulae. For instance, "state equals calculate angle by two inertial navigation modules state" is translated to "$state = state\_CalA\_BothGD\_State$". Secondly, the generated formulae are connected by "$\rightarrow$" which is translated from the keywords recorded by the root node. During the translation, the *Scope* patterns ,"globally", are translated into corresponding temporal operators and added to the corresponding formulae. The LTL and CTL formulae are shown as follows:

$$LTL : G((state = state\_CalA\_BothGD\_State) \rightarrow \\ G(angle = (0.5 * (angle\_GD1 + angle\_GD2)))) \tag{1}$$

$$CTL : AG((state = state\_CalA\_BothGD\_State) \rightarrow \\ AG(angle = (0.5 * (angle\_GD1 + angle\_GD2)))) \tag{2}$$

5 SCADE2nuXmv Model Transformation

In this section, we will introduce the automated transformation from SCADE to nuXmv. Fig.6 shows the overview of SCADE2nuXmv. We first extract the textual representation of the SCADE model through SCADE IDE. Subsequently, use

ANTLR, a toolkit for lexical analysis and syntax analysis, to build a syntax tree for it. Then generate target nuXmv models based on the syntax tree. Finally, we employ the model checker of nuXmv to verify the generated model. The traceability between the execution trace of counterexamples, and corresponding formulae and requirements is also generated.

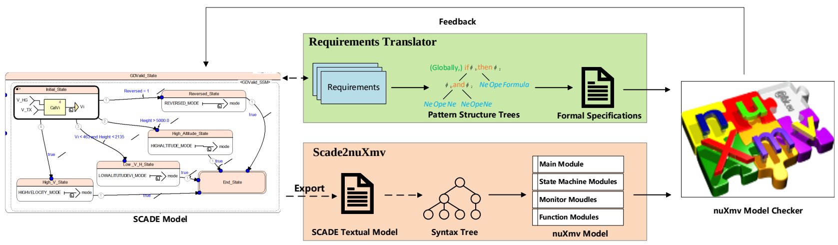

Fig. 6. SCADE Models Verification Achieved by SCADE2nuXmv

The reasons why we choose nuXmv are the ability to express hierarchical models, support for both SMT-based and SAT-based verification, and the verification of both infinite-state and finite-state systems. For instance, the middle and right columns in Fig.7 construct a nuXmv model that contains hierarchical state machine. The SM_EJ_Core is the top state machine of the model. It declares the sub-state machine $SM\_GDValid\_State$ module, which selects strategy according to the different variables (e.g., Vi), in its VAR part.

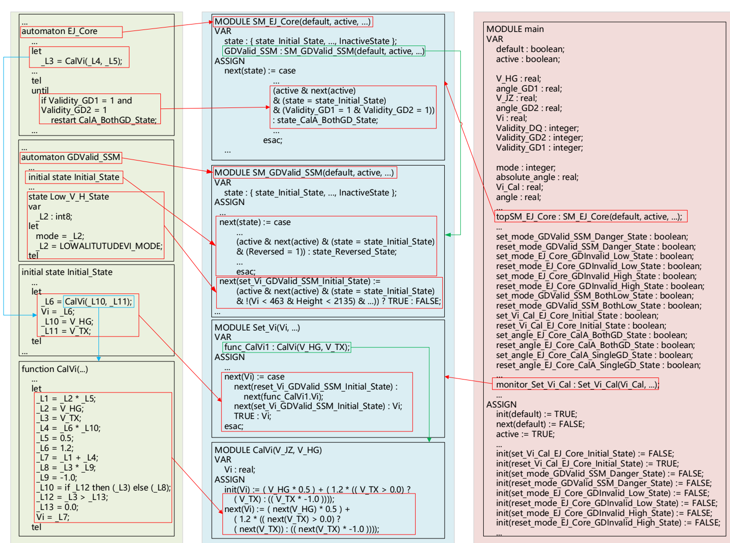

Fig. 7. The Translation from SCADE to nuXmv

The nuXmv models generated by our method contains four module types: *Monitor*, *State Machine*, *Function* and *Main*. The *Monitor* module implements the monitor mechanism. The *State Machine* module and *Function* module respectively represent safety state machine and data flow in SCADE models. The *Main* module is the entry of target models. In the following, we will introduce these modules in detail.

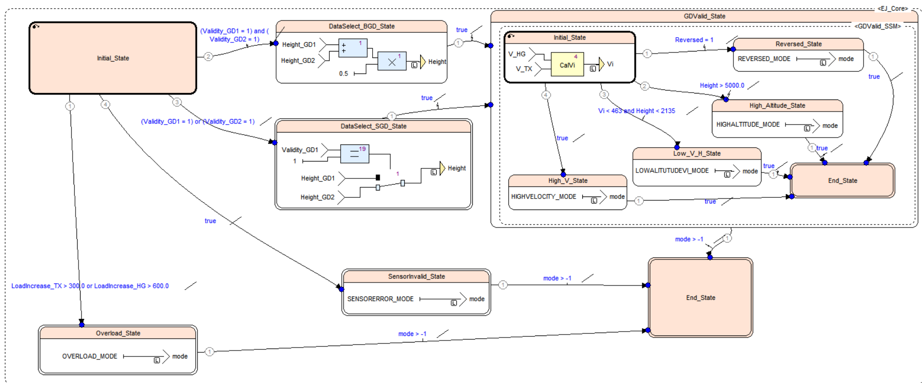

Fig. 8. The SCADE Model Example

Monitor Module The language supported by nuXmv forbids multiple assignments to a variable [4], which is common in safety state machine, so we created *Monitor* modules inspired by [8] to manipulate the assignment of output and local variables. Each *Monitor* module incorporates two "case" expressions with multiple execution branches, to assign the value of the monitored variable. One initializes the monitored variable, while the other updates it after the first cycle.

Each "case" expression branch corresponds to an assignment expression of the monitored variable in the original SCADE model. To determine the branch to execute, we create *monitor variables* similar to the ones in [8]. Each *monitored variable* triggers a branch when it becomes true. To distinguish between assignment expressions related to state transitions, we employ two types of monitor variables, $reset\_x\_s$ and set_x_s, where x denotes the monitored variable and s denotes the state assigning the value of x. When the state machine transitions to s, $reset\_x\_s$ turns TRUE. If none of the transition conditions of state s are met, set_x_s becomes TRUE. We further replace variables prefixed with "_L", which represent connection lines in the SCADE model, with their equivalent variables to reduce the state space. For example, the "Set_Vi" in Fig.7 is an example of such a module. The "case" expression assigns the value of "Vi" according to the monitor variables of each branch.

State Machine Module Each *State Machine* module corresponds to one state machine in the original SCADE model. It records the state transition, assigns monitor variables and instantiates submachines. Such modules also implement the hierarchical structure in safety state machine utilizing *active* and *default* variables, which respectively denote the activation of the state machine and whether the state machine transitions from inactive to active in the cycle. For example, the "*SM_ GDValid_ SSM*" module in Fig.7 is a *State Machine* module, representing the state machine "*GDValid_ SSM*" in Fig.8. The first "case" expression depicts the state transition, while subsequent one assigns monitor variables related to it.

Function Module A *Function* module corresponds to a data flow operator in the original SCADE model. Each output variable of it is defined by two expressions in the corresponding *Function* module. One, the same as the expression in the SCADE model, assigns its initial value. The other, similar to the expression in the SCADE model but with each variable enclosed by a *next* operator, assigns the value of the variable after the first cycle. After the translation, we further replace the temporary variables "_Li" (where i is a positive integer), that represent connection lines in the SCADE model, with their equivalent variables to reduce the state space. For example, the "*CalVi*" module in Fig.7 is a *Function* module. The temporary variable "_Li" is replaced with its equivalent variable, e.g., "_L2" is replaced with "V_HG".

Main Module Each nuXmv model has one *Main* module which instantiates the interface of the SCADE model, the local variables, all monitor variables and modules, and the top state machine. It also assigns the initial value of all monitoring variables. Additionally, a variable equalling the state of the top state machine is defined in the *Main* module, allowing users to verify specifications related to a specific state. This variable only exists when the original SCADE model contains safety state machines.

As mentioned in section 1, SCADE provides three modeling styles, i.e., safety state machines, data flow and their combination. When the SCADE model is data flow style, its nuXmv model contains *Main Module* and *Function Module*. On the contrary, when the SCADE model is safety state machine style, we use *State Machine* and *Main* modules to construct the target nuXmv models. In addition, we use all the above four types of modules to construct the target nuXmv model for the SCADE model that combines data flow and safety state machine. The translation algorithms of these modules are shown in Appendix 4.

6 Industrial Case Studies and Evaluation

6.1 Ejection Seat Control System

Ejection seats must eject pilots out of the cockpit when the pilots pull the switch and open the parachute at an appropriate time to safely send pilots back to the ground. When the seat is ejected from the cockpit, the control system in the seat chooses different control strategies as the environment varies. During this process, it avoids rotation, excessive loads and wrong movement direction. When the seat is in a non-upright position, the control system adjusts the attitude of the seat.

As a typical safety-critical software, SCADE is suitable to model this control system. Moreover, the control system controls all subsystems of the ejection seat to faithfully implement the chosen strategy. Each step in the strategy has a strict execution order. This feature makes temporal logic suitable for describing the specifications of its requirements.

6.2 Implementation and Experiments

We have developed a tool chain to implement the AGVTS method in this paper. To verify the effectiveness of AGVTS, we utilize the tool chain to verify six modules in the ejection seat control system against a set of 124 requirements.

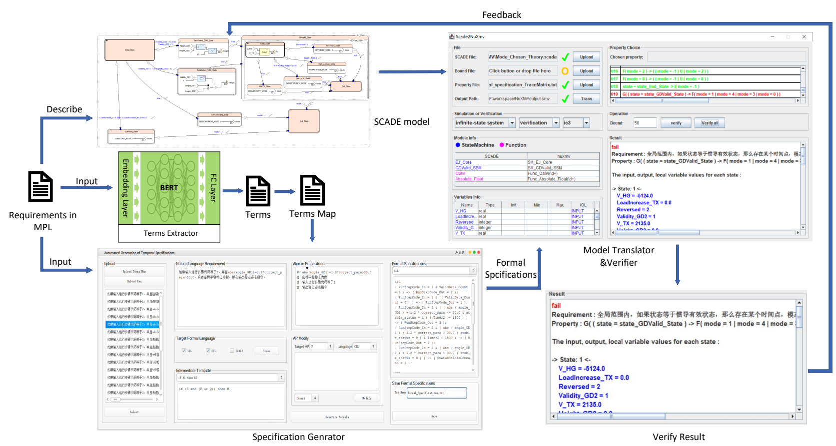

Fig. 9. Implementation of the Approach

As shown in Fig.9, the tool chain first uses an *BERT-based terms extractor*, which is fine-tuned on the requirements of the ejection seat control system, to extract terms from the requirements written in MPL. Industrial engineers subsequently confirm and rectify the errors in the extracted terms. Then they build a *terms map* based on the terms. Thirdly, the *specification generator* reads the *terms map* and converts the requirements written in MPL into LTL and CTL formulae. Finally a *model transformer & verifier* transforms the SCADE model into nuXmv and verifies the generated formulae against the nuXmv model. Additionally, it generates the traceability between the execution trace of counterexamples, and corresponding formulae and requirements to help rectify the original model.

To ensure the effectiveness of the tool chain, we invite several formal experts to confirm whether the generated LTL and CTL formulas accurately capture the intent of the requirements. Then the experts check the generated nuXmv models to determine whether these models faithfully implement the functions of the original models.

6.3 Evaluation

The major objective of this subsection is to evaluate the effectiveness of our method. We will explain the evaluation from three perspectives: Terms Extraction, Requirements Translation and Model Verification.

Table 3. The Statistics of Our BERT-Based Model

Data Set	Precision/%	Recall/%	F_1/%
Fine-tuned Test Set	93.94	95.88	94.90
Requirements of six modules	86.81	90.80	88.76

Terms Extraction As shown in Table 3, the BERT-based terms extractor achieves a precision of 93.94% and a recall of 95.88% on the fine-tuned test set. The 124 requirements used in our experiment contain 87 terms. The BERT-based terms extractor extracts 91 terms, 79 of which are correct. Error-extracted terms can be divided into two categories. One is common nouns (e.g., "cycle") that do not belong to ejection seat control system. The other is the segment of larger terms, for example "Inertial Navigation Module" is a wrong term split from "Valid Inertial Navigation Module State". However, these can be easily identified and rectified by engineers.

Table 4. The Statistics of the Verification Results

Modules	Reqs	AGVTS (Verify)	Wrong Req
Mode Selection	21	21	0
Ejection Judge	18	18	2
High Altitude Mode	26	26	0
Velocity Control Mode	20	16	0
Overload Mode	24	24	0
Stability Control	15	15	0

Requirements Translation Our method successfully translates all the preprocessed requirements written in the pattern based language into LTL and CTL formulae. Table 4 shows the translation statistics data. In the following we provide two requirements described in MPL to show the translation.

Example 2. "Globally, in the Inertial Navigation Valid State, the mode will be set to *reverse* mode, or the mode will be set to *low velocity low altitude* mode, or the mode will be set to *high velocity* mode, or the mode will be set to *high altitude* mode".

This is a property used to limit the output of the system whose corresponding LTL and CTL formulae are shown as follows.

$$LTL : G((state = state\_GDValid\_State) \rightarrow \\ F(mode = 4 \,|\, mode = 6 \,|\, mode = 0 \,|\, mode = 5)) \tag{3}$$

$$CTL : AG((state = state\_GDValid\_State) \rightarrow \\ AF(mode = 4 \,|\, mode = 6 \,|\, mode = 0 \,|\, mode = 5)) \tag{4}$$

Please note that, the terms map has matched the *reverse* mode, *low velocity low altitude* mode, *high velocity* mode, and *high altitude* mode to the constants defined in the SCADE model. For instance "the mode will be set to *reverse* mode" is translated to "$mode = 4$".

Example 3. "If the rational flag of inertial navigation module 1 is true, and the rational flag of inertial navigation module 2 is true, then in the next cycle, state will be set to calculate_angle_Both_Inertial_Valid state".

As the statement does not have a *Scope*, it is only checked in the first cycle. Formula (5) and (6) represent the LTL and CTL formula of this requirement.

$$LTL : (Validity\_GD1 = 1 \,\&\, Validity\_GD2 = 1) \rightarrow \\ X(state = state\_CalA\_BothGD\_State) \tag{5}$$

$$CTL : (Validity\_GD1 = 1 \,\&\, Validity\_GD2 = 1) \rightarrow \\ AX(state = state\_CalA\_BothGD\_State) \tag{6}$$

Model Verification The SCADE models of the six modules contain three types of structure: data flow, safety state machine and their combination. Our method successfully transforms them. Then we verify the nuXmv models with the generated formulae.

During the verification, two bugs caused by the "case" statement are found. The first one is that the front case condition covers the latter case condition. The second one is caused by two identical case condition, but their actions are different. This result shows that our method for verification is effective. Additionally, our method verifies the SCADE models of the six modules in a shorter time than DV.

Four requirements of the *Velocity Control* mode contain nonlinear calculations, such as square root, that cannot be verified by SCADE DV and our method. Note that, when the SCADE models contain unbounded integers and real numbers, nuXmv just verify the LTL specifications.

7 Related Work

Formal Specification Generation As writing formal specifications is time-consuming and error-prone, [14, 19] propose a set of patterns corresponding to different scenarios and their formal semantics to guide users to write formal specifications. [15] develops a SpeAR tool which translates requirements written in pattern language into PLTL. [10, 16, 21] provide a framework to translate requirements written in the pattern language FRETISH into formal specifications. [2] proposes a framework combining existing classical patterns. [22, 26, 29] utilize NLP techniques, such as POS tagging and dependency parsing, to pre-process requirements. Then they define translation rules to generate formal specifications based on the pre-processed requirements. [18, 25] treat the translation from natural language to formal specifications as a machine translation task and utilize deep learning models to solve it. [5, 11] utilize Large Language Models (LLMs) to complete the translation. [11] decomposes the natural language input into sub-translations by utilizing LLMs. However, these methods require either manual writing of formal specifications for atomic propositions, or utilizing plenty of patterns, or training with plenty of data.

Verification of SCADE Models As SCADE DV cannot adequately express complex temporal specifications and it may fail due to complexity problems such as floating numbers, there are several related works to enhance the verification capability of SCADE. [12] transforms SCADE models into UPPAAL and verify the liveness properties in TCTL. However, it may fail when the original SCADE model contains hierarchical structure. [24] transforms SCADE models into nuSmv [6] models for verification, but limited to safety state machines and incapable of verifying infinte-state system. [3] introduces LAMA as an intermediate language into which SCADE programs can be transformed and which easily can be transformed into SMT solver instances. However, the method performs worse than DV.

8 Conclusion and Future Work

We propose an AGVTS method for automatically generating temporal specifications and verifying aeronautics SCADE models. AGVTS begins by defining a modular pattern language (MPL) to express Chinese requirements precisely. Then it uses a rule-based method augmented with BERT to translate requirements in MPL into LTL and CTL formulae. Finally it verifies SCADE models by transforming them into nuXmv which supports SMT-based and SAT-based verification. The method is applied to an ejection seat control system and the results convince our industrial partners of the usefulness of formal methods to industrial systems.

We are currently incorporating LLMs to enhance the capability of terms extraction and the process of parsing requirements. The patterns will be refined to cover more domains and enable users to articulate requirements with greater flexibility. Additionally, we will consider theorem prover Coq to prove the correctness of formula generation and the transformation from SCADE to nuXmv, based on our previous researches [27, 28]. Improving the verification capability of nuXmv is also an interesting work.

References

1. Alur, R., Courcoubetis, C., Dill, D.: Model-checking in dense real-time. Information and computation **104**(1), 2–34 (1993)
2. Autili, M., Grunske, L., Lumpe, M., Pelliccione, P., Tang, A.: Aligning qualitative, real-time, and probabilistic property specification patterns using a structured english grammar. IEEE Transactions on Software Engineering **41**(7), 620–638 (2015)
3. Basold, H., Günther, H., Huhn, M., Milius, S.: An open alternative for smt-based verification of scade models. In: Formal Methods for Industrial Critical Systems: 19th International Conference, FMICS 2014, Florence, Italy, September 11-12, 2014. Proceedings 19. pp. 124–139. Springer (2014)
4. Bozzano, M., Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: nuxmv 2.0. 0 user manual. fondazione bruno kessler. Tech. rep., Technical report, Trento, Italy (2019)
5. Chen, Y., Gandhi, R., Zhang, Y., Fan, C.: NL2TL: Transforming natural languages to temporal logics using large language models. In: Bouamor, H., Pino, J., Bali, K. (eds.) Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing. pp. 15880–15903. Association for Computational Linguistics, Singapore (Dec 2023)
6. Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: Nusmv: A new symbolic model verifier. In: Computer Aided Verification: 11th International Conference, CAV'99 Trento, Italy, July 6–10, 1999 Proceedings 11. pp. 495–499. Springer (1999)
7. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Transactions on Programming Languages and Systems (TOPLAS) **8**(2), 244–263 (1986)
8. Clarke, E.M., Heinle, W.: Modular translation of statecharts to smv. Tech. rep., Citeseer (2000)
9. Colaço, J.L., Pagano, B., Pouzet, M.: Scade 6: A formal language for embedded critical software development. In: 2017 International Symposium on Theoretical Aspects of Software Engineering (TASE). pp. 1–11. IEEE (2017)
10. Conrad, E., Titolo, L., Giannakopoulou, D., Pressburger, T., Dutle, A.: A compositional proof framework for fretish requirements. In: Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs. pp. 68–81 (2022)
11. Cosler, M., Hahn, C., Mendoza, D., Schmitt, F., Trippel, C.: nl2spec: Interactively translating unstructured natural language to temporal logics with large language models. In: International Conference on Computer Aided Verification. pp. 383–396. Springer (2023)
12. Daskaya, I., Huhn, M., Milius, S.: Formal safety analysis in industrial practice. In: Formal Methods for Industrial Critical Systems: 16th International Workshop, FMICS 2011, Trento, Italy, August 29-30, 2011. Proceedings 16. pp. 68–84. Springer (2011)
13. Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805 (2018)
14. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of the 21st international conference on Software engineering. pp. 411–420 (1999)
15. Fifarek, A.W., Wagner, L.G., Hoffman, J.A., Rodes, B.D., Aiello, M.A., Davis, J.A.: Spear v2. 0: Formalized past ltl specification and analysis of requirements.

In: NASA Formal Methods: 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings 9. pp. 420–426. Springer (2017)

16. Giannakopoulou, D., Pressburger, T., Mavridou, A., Schumann, J.: Automated formalization of structured natural language requirements. Information and Software Technology **137**, 106590 (2021). https://doi.org/https://doi.org/10.1016/j.infsof.2021.106590
17. Gleirscher, M., van de Pol, J., Woodcock, J.: A manifesto for applicable formal methods. Software and Systems Modeling **22**(6), 1737–1749 (2023)
18. He, J., Bartocci, E., Ničković, D., Isakovic, H., Grosu, R.: Deepstl: from english requirements to signal temporal logic. In: Proceedings of the 44th International Conference on Software Engineering. pp. 610–622 (2022)
19. Konrad, S., Cheng, B.H.: Real-time specification patterns. In: Proceedings of the 27th international conference on Software engineering. pp. 372–381 (2005)
20. Leveson, N.G.: Engineering a safer world: Systems thinking applied to safety. The MIT Press (2016)
21. Mavridou, A., Bourbouh, H., Garoche, P.L., Giannakopoulou, D., Pessburger, T., Schumann, J.: Bridging the gap between requirements and simulink model analysis. In: Joint 26th International Conference on Requirements Engineering: Foundation for Software Quality Workshops, Doctoral Symposium, Live Studies Track, and Poster Track (2020)
22. Nayak, A., Timmapathini, H.P., Murali, V., Ponnalagu, K., Venkoparao, V.G., Post, A.: Req2spec: Transforming software requirements into formal specifications using natural language processing. In: International Working Conference on Requirements Engineering: Foundation for Software Quality. pp. 87–95. Springer (2022)
23. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science (sfcs 1977). pp. 46–57. ieee (1977)
24. Shi, J., Shi, J., Huang, Y., Xiong, J., She, Q.: Scade2nu: A tool for verifying safety requirements of scade models with temporal specifications. In: REFSQ Workshops (2019)
25. Wang, C., Ross, C., Kuo, Y.L., Katz, B., Barbu, A.: Learning a natural-language to ltl executable semantic parser for grounded robotics. In: Conference on Robot Learning. pp. 1706–1718. PMLR (2021)
26. Yan, R., Cheng, C.H., Chai, Y.: Formal consistency checking over specifications in natural languages. In: 2015 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 1677–1682. IEEE (2015)
27. Yang, Z., Bodeveix, J., Filali, M.: Towards a simple and safe objective caml compiling framework for the synchronous language SIGNAL. Frontiers Comput. Sci. **13**(4), 715–734 (2019)
28. Yang, Z., Bodeveix, J., Filali, M., Hu, K., Zhao, Y., Ma, D.: Towards a verified compiler prototype for the synchronous language SIGNAL. Frontiers Comput. Sci. **10**(1), 37–53 (2016)
29. Zhang, S., Zhai, J., Bu, L., Chen, M., Wang, L., Li, X.: Automated generation of ltl specifications for smart home iot using natural language. In: 2020 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 622–625. IEEE (2020)

Appendix 1 *AP* Patterns

We have defined nine *AP* patterns which are composed of the permutation and combination of different *Ingredient* tags. These *Ingredient* tags composed by two parts. One is traditional Part-of-Speech tag, such as adjective and adverb. The other is the tag defined by ourselves, for example "Ne" used to denotes term tokens in origin sentence. Table.5 shows all *Ingredient* tags. Notice that except for complete formula, the part of speech *Formula* also includes formulas start with binary operators preceded by terms written in natural language, like "angle of pitch $= max(x_1, x_2)$". We also map adjective and verb to figure according to the file provided by users.

Table 5. Ingredient Tags in AP Pattern

Ingredient tag	Meaning	Example
Ne	terms in requirements	angle of pitch
Verb	system components' action	ignite
Adj	status of system components	valid
Ope	mathematical or temporal operators	greater than (add)
Value	numerical value	30.0
Deny	negative tokens	don't
Formula	formulas written by users	$x \geq abs(pitch\_angle)$

When multiple entities have the same properties or action, users usually use "," or "/" to connect these entities, for example "$angle\_sensor1$, $angle\_sensor2$ is greater than $angle\_pitch$". Our *AP* patterns also support this, and all entities play the same role in a sentence will be classified to one *Ne* ingredient.

Let $\mathcal{M}$ be the system model we want to verify, $\mathbf{s}$ be a time point in the running of the system, $\mathbf{L}(\mathbf{s})$ be the set of true equations at the time point $\mathbf{s}$ and the elements enclosed by "[ ]" are optional.

Let val_adj, val_verb respectively denote corresponding value of adjective and verb, $\circ$ denote mathematical or temporal operators. "ne" denotes terms' format in SCADE model, "value" denotes the corresponding number of "Value" and ψ denotes the formula matches "Formula". If terms are plural, the number of corresponding terms is n, and i is the index of term which satisfy $0 \leq i \leq n$. As the model checker we used is nuXmv, the negative operator is "!".

All the patterns and their semantics are defined as follows.

1. ***Ne* [*Deny*] *Verb***: This pattern describes the action of system components. The semantic of this pattern is

$$\mathcal{M}, \mathbf{s} \models \boldsymbol{Ne\, Verb} \Leftrightarrow ne \text{ equals } val\_verb \tag{7}$$

Its corresponding formal specification is "$ne = val\_verb$". If ***Deny*** exists, the semantic of this pattern will be changed into

$$\mathcal{M}, \mathbf{s} \models \boldsymbol{Ne\, Deny\, Verb} \Leftrightarrow ne \text{ is not equal to } val\_verb \tag{8}$$

The formal specification of this pattern will be changed into "$!(ne = val\_verb)$". This pattern can also be written as $[\boldsymbol{Deny}]$ $\boldsymbol{Verb\ Ne}$.

2. $\boldsymbol{Ne\ Adj}$: This pattern describes the status of system component. The semantic of this pattern is

$$\mathcal{M}, \mathbf{s} \models \boldsymbol{Ne\ Adj} \Leftrightarrow ne \text{ equals } val\_adj \tag{9}$$

The formal specification of this pattern is "$ne = val\_adj$".

3. $[\boldsymbol{Deny}]$ $\boldsymbol{Verb\ Adj\ Ne}$: This pattern expresses changing the status of system components by specific action and let "val" be final value of the *Ne*. The semantic of this pattern is

$$\mathcal{M}, \mathbf{s} \models \boldsymbol{Verb\ Adj\ Ne} \Leftrightarrow ne \text{ equals } (val\_adj \odot val\_verb) \tag{10}$$

If $\boldsymbol{Deny}$ exists it's semantic will change as

$$\mathcal{M}, \mathbf{s} \models \boldsymbol{Deny\ Verb\ Adj\ Ne} \Leftrightarrow ne \text{ is not equal to } (val\_adj \odot val\_verb) \tag{11}$$

As described above the formal specification will change according to the val of *val_adj* and *val_verb*. If *val_adj* equals *val_verb*, the formal specification of this pattern is "$ne = 1$ $(ne = 0)$". If their values are different, the specification will change into "$ne = 0$ $(ne = 1)$". The specification in bracket is the format when $\boldsymbol{Deny}$ exists.

4. $\boldsymbol{Formula}$: This pattern expresses the situation that the requirements only consist of formula, so this pattern's semantic is

$$\mathcal{M}, \mathbf{s} \models \boldsymbol{Formula} \Leftrightarrow \psi \in \mathbf{L}(\mathbf{s}) \tag{12}$$

The formal specification of this pattern is ψ.

5. $\boldsymbol{Ne\ Formula}$: This pattern expresses the situation that terms written in natural language precede a formula. In this situation the formula always starts with a binary operator, so the formula of this pattern is "$ne\,\phi$". Let ψ denote $ne\,\phi$, The semantic of this pattern is

$$\mathcal{M}, \mathbf{s} \models \boldsymbol{Ne\ Formula} \Leftrightarrow (ne\ \psi) \in \mathbf{L}(\mathbf{s}) \tag{13}$$

6. $\boldsymbol{Ne\ Ope\ Value}$: This pattern is used to set system components value, or compare system components with specific value. The semantic of this pattern is

$$\mathcal{M}, \mathbf{s} \models \boldsymbol{Ne\ Ope\ Value} \Leftrightarrow ne \circ value \tag{14}$$

The formal specification of this pattern is "$ne \circ value$"

7. $\boldsymbol{Ne_1\ Ope\ Ne_2}$:This pattern is used to express the computation between terms, or the relation between terms. The semantic of this pattern is

$$\mathcal{M}, \mathbf{s} \models \boldsymbol{Ne_1\ Ope\ Ne_2} \Leftrightarrow ne_1 \circ ne_2 \tag{15}$$

The formal specification of this pattern is "$ne_1 \circ ne_2$"

8. ***Ne Ope Adj***: This pattern is used to express the comparison between terms and the value of "Adj" or the action of setting terms to the value of "Adj". The adjectives matches "Adj" is the words that can be treated as value, like "true". The semantic of this pattern is

$$\mathcal{M}, \mathbf{s} \models \boldsymbol{Ne\,Ope\,Adj} \Leftrightarrow ne \circ val\_adj \tag{16}$$

This formal specification of this pattern is "$ne \circ val\_adj$". This pattern can also be written as ***Ope Ne Adj***.
9. ***Ne Ope Formula***: This pattern describes the situation that terms and operators are written in natural language and are followed by a formula, so the formal specification of this pattern can be written as "$ne \circ \psi$". The semantic of this pattern is

$$\mathcal{M}, \mathbf{s} \models \boldsymbol{Ne\,Ope\,Formula} \Leftrightarrow (ne \circ \psi) \in \mathbf{L}(\mathbf{s}) \tag{17}$$

If the part of speech $\boldsymbol{Ne}$ in AP pattern matches more than one term, we build formal specification for every term separately and use $\vee$ or $\wedge$ connecting them. If the terms connected by ",", we choose $\vee$, otherwise we choose $\wedge$. For example, if $\boldsymbol{Ne}$ in the AP pattern $\boldsymbol{Ne}\,[\boldsymbol{Deny}]\,\boldsymbol{Verb}$, which does not have $[\boldsymbol{Deny}]$ in pattern, matches more than one term, it's semantic will change into

$$\begin{aligned}\mathcal{M}, \mathbf{s} \models \boldsymbol{Ne\,Verb} \Leftrightarrow\ & ne_1 = val\_verb \text{ and } ne_2 = val\_verb \cdots \\ & \text{and } ne_m = val\_verb\end{aligned} \tag{18}$$

In this semantic definition m is the number of terms matched by $\boldsymbol{Ne}$, $\neq_j$ $(1 \leq j \leq m)$ is terms matched by $\boldsymbol{Ne}$. The formal specifications of this pattern will be changed into "$ne_1 = val\_verb \wedge ne_2 = val\_verb \cdots \wedge ne_n = val\_verb$". If the terms are connected by "/", the *wedge* will be exchanged by *vee*. These effect are the same in other AP patterns.

To distinguish different *Ingredients*, we build four corpuses. These new corpuses are *Action Corpus*, *Adjective Corpus*, *Function Corpus* and *Operator Corpus*. The *Action Corpus* provides all the verbs that may appear in requirements and their value in the SCADE model, for example one record is the verb "ignite" and its value 1. The *Adjective Corpus* and similar records of adjectives. For example one record in *Adjective Corpus* is the word "true" and its value 1. The *Operator Corpus* record the mapping between operators in natural language and their format in SCADE models. For instance, one record is operator "less than" and its SCADE format is "<". The *Function Corpus* records all the functions that we may use in the requirement, like function "abs" for calculating the absolute value, as well as their parameter number.

Appendix 2 *Relation* Patterns

We have defined six patterns, *Response, Condition, Precedence, Conjunction, Disjunction* and *Simple*, to express the relation of sentences. The *simple* pattern is simple sentences described in AP pattern.

The Relation pattern is composed of keywords and clauses. The keywords connect is the symbol of different patterns clauses to express more complex semantics and can be used to distinguish different relation patterns. The clauses can be simple sentences or a complex sentence described in *Relation* patterns to support nesting. The syntax of *Relation* patterns is defined as follows where the bold Chinese character is keywords and others elements correspond to patterns with the same name. As the natural language we process is Chinese, we provide a map between Chinese keywords and English keywords at the end of this Appendix.

$$\begin{aligned} req ::=& Response \,|\, Condition \,|\, Precedence \\ & |\, Conjunction \,|\, Disjunction \,|\, Weak\_Conjunction \\ & |\, Weak\_Disjunction \,|\, \phi \\ Response ::=& \textbf{if } req \textbf{ holds, then in response } req \\ Condition ::=& \textbf{if } req\textbf{, then } req \\ Precedence ::=& req \textbf{ precedes } req \\ Conjunction ::=& req\ (\textbf{and} \,|\, \textbf{weak\_and})\ req\ \cdots \\ & (\textbf{and} \,|\, \textbf{weak\_and})\ req \\ Disjunction ::=& req\ (\textbf{or} \,|\, \textbf{weak\_or})\ req\ \cdots \\ & (\textbf{or} \,|\, \textbf{weak\_or})\ req \end{aligned} \tag{19}$$

To precisely build the grammar tree of requirements, we define the priority of these patterns. The priority is *Response* $=$ *Condition* $>$*Precedence* $>$*Conjunction* $>$*Disjunction* $>$*Weak_Conjunction* $>$*Weak_Disjunction* $>$*Simple*. If the priority of pattern A is higher than that of B, then pattern B is seen as a clause of pattern A. If the priority of pattern A equals pattern B and the index of pattern A's keywords is less than pattern B, pattern B is seen as the clause of pattern A.

Let $\mathcal{M}$ be the system model we want to verify, $s_0\, s_1\, \cdots\, s_t\, \cdots\, s_n\ (0 \leq t \leq n)$ be a running trace of $\mathcal{M}$ and $\mathbf{L}(\mathbf{s_t})$ be the set of true equations at the time point $\mathbf{s_t}$.

The semantic of every pattern is defined as follows, except for the *simple* pattern, where $\phi_i\ (1 \leq i \leq m)$ denotes different clauses of a pattern, m denotes the number of clauses:

- *Response* pattern: This pattern describes the situation that when ϕ_1 holds, ϕ_2 must hold in the next cycle after the holds of ϕ_1. The semantic of this pattern is defined as follows:

$$\begin{aligned} \mathcal{M}, \mathbf{s_t} \models \textbf{ if } \phi_1 \textbf{ holds, then in response } \phi_2 \Leftrightarrow \\ \phi_1 \in \mathbf{L}(\mathbf{s_t}) \text{ and } \phi_2 \in \mathbf{L}(\mathbf{s_{t+1}}) \end{aligned} \tag{20}$$

- *Condition* pattern: The pattern expresses that whenever ϕ_1 holds, ϕ_2 holds at same cycle. The semantic of this pattern is

$$\mathcal{M}, \mathbf{s_t} \models \textbf{ if } \phi_1\textbf{, then } \phi_2 \Leftrightarrow \phi_1 \in \mathbf{L}(\mathbf{s_t}) \text{ and } \phi_2 \in \mathbf{L}(\mathbf{s_t}) \tag{21}$$

- *Precedence* pattern: This pattern describes the situation that ϕ_2 must holds at least on time, before ϕ_1 holds.

$$\begin{aligned}\mathcal{M},\mathbf{s_t} \models \phi_2 \ \mathbf{precedes}\ \phi_1 \Leftrightarrow & \exists\, t,k \text{ satisfy } t \leq k < j \leq n \ , \\ & \forall\, a \text{ satisfies } t \leq a \leq k,\ \phi_1 \in \mathbf{L}(\mathbf{s_j}) \\ & \text{and } \phi_2 \in \mathbf{L}(\mathbf{s_k}) \text{ and } \phi_1 \notin \mathbf{L}(\mathbf{s_a})\end{aligned} \tag{22}$$

  Different from other patterns, this patterns' semantic can be effected by the content of ϕ_2. If ϕ_2 contains the adverb always, the formula of this pattern will be changed as ϕ_2 must hold all the time before ϕ_1 holds and the semantic will change into:

$$\begin{aligned}\mathcal{M},\mathbf{s_t} \models \phi_2 \ \mathbf{precedes}\ \phi_1 \Leftrightarrow & \exists\, j \text{ satisfies } t \leq j \leq n, \\ & \forall\, k \text{ satisfies } t \leq k < j, \\ & \forall\, a \text{ satisfies } t \leq a \leq k\ ,\phi_1 \in \mathbf{L}(\mathbf{s_j}) \\ & \text{and } \phi_2 \in \mathbf{L}(\mathbf{s_k}) \text{ and } \ \phi_1 \notin \mathbf{L}(\mathbf{s_a})\end{aligned} \tag{23}$$

- *Conjunction*: This pattern describes the situation that all ϕ_i must happen in the same cycle. We define two types of this pattern to complete the nesting of conjunction and disjunction and their priority are defined in section *Modular Pattern Language*. The semantic of this pattern is

$$\begin{aligned}\mathcal{M},\mathbf{s_t} \models \phi_1(\mathbf{and}\,|\,\mathbf{weak\_and})\,\phi_2 \cdots (\mathbf{and}\,|\,\mathbf{weak\_and})\,\phi_n \Leftrightarrow \\ \forall\, i \text{ satisfies } 1 \leq i \leq n,\ \phi_i \in \mathbf{L}(\mathbf{s_t})\end{aligned} \tag{24}$$

- *Disjunction*: This pattern expresses that at least one ϕ_i happens in target cycle. We define two types of this patternto complete the nesting of conjunction and disjunction and their priority are defined in section *Modular Pattern Language*. The semantic of this pattern is

$$\begin{aligned}\mathcal{M},\mathbf{s_t} \models \phi_1(\mathbf{or}\,|\,\mathbf{weak\_or})\,\phi_2 \cdots (\mathbf{or}\,|\,\mathbf{weak\_or})\,\phi_n \Leftrightarrow \\ \exists\, i \text{ satisfies } 1 \leq i \leq n, \phi_i \in \mathbf{L}(\mathbf{s_t})\end{aligned} \tag{25}$$

Table.2 shows the LTL and CTL formulas of the *Relation* patterns, except for the *simple* pattern, where ϕ_i denotes clauses, $1 \leq i \leq n$ and n is the number of clauses. The example of each pattern is same with that in brackets above.

Table 6. The Map between English Keywords and Chinese Keywords

English Keywords	Chinese Keywords
if req **holds, then in response** req	req后，req
if req **, then** req	如果req，那么req
req **precedes** req	req前，req
req (**and** | **weak_and**) req ⋯ (**and** | **weak_and**) req	req，并且|且req，⋯，并且|且req
req (**or** | **weak_or**) req ⋯ (**or** | **weak_or**) req	req，或者|或req，⋯，或者|或req

Appendix 3 *Scope* Pattern

According to the semantics, our *Scope* patterns can be divided into four types, *Quantifier*, *Temporal*, *Cycle* and *State*. These scopes can only appear at the start of a clause or requirement.

Let $\mathcal{M}$ be the system model we want to verify, s_t $(0 \leq t \leq n)$ be a state in the running trace of $\mathcal{M}$, π be a trace of the running of $\mathcal{M}$ composed of $s_0, s_1, \cdots s_n$, the time that s_i transitions to s_{i+1} is called one cycle, $\mathbf{L}(\mathbf{s_t})$ be the set of true equations at the state $\mathbf{s_t}$, π^t be the trace starts from the state $\mathbf{s_t}$ and $\mathbf{p}$ be the formal specification of requirement written in a *Relation* patterns. We will use these to define the semantics of *Scope*.

Quantifier Scope denotes quantifiers in formal languages. We define *exists* and *for all* scopes for this kind. We mainly reference the definition in Computation Tree Logic (CTL) to define these two scopes, so when using these two scopes the trace will change from one line into a computation tree. As LTL and CTL are our target languages and LTL language doesn't have quantifiers, these scopes will be ignored when the target language is LTL. This kind has two scopes and their semantics are defined as follows where ϕ denotes the requirement the *Scope* modify. These two scopes can only be used with *Temporal Scope*. As the natural language we process is Chinese, we provide a map between Chinese keywords and English keywords at the end of this Appendix.

- **For all trace,** ϕ: This scope state that ϕ must hold on all pathes which extend from the point where ϕ starts. The semantic of this scope and the CTL format specifications are

$$\begin{aligned} semantic: \ & \mathcal{M}, \mathbf{s_t} \models \textbf{for all trace, } \phi \Leftrightarrow \forall \pi^t, \ \mathcal{M}, \pi^t \models \phi \\ CTL\ formula: \ & A\,\phi \end{aligned} \tag{26}$$

- **Exists one trace,** ϕ: This scope express that ϕ must hold in at least one path which extend from the point where ϕ starts. The semantic of this pattern and the CTL format specifications are defined as follows

$$\begin{aligned} semantic: \mathcal{M}, \mathbf{s_t} \models \ & \textbf{Exists one trace, } \phi \Leftrightarrow \\ & \exists\,\pi^t, \ \mathcal{M}, \pi^t \models \phi \\ CTL\ formula: \ & E\,\phi \end{aligned} \tag{27}$$

Temporal Scope denotes unary temporal operators in formal languages. We defined three scopes belonging to this pattern which denote temporal operators "G", "F" and "X" in LTL and CTL. The following are these scopes and their semantics where ϕ denotes the requirement the *Scope* modify.

- **Globally,** ϕ: this scope states that ϕ must hold every state after the state where ϕ is considered. The semantic and LTL, CTL format specifications of this scope are

$$\begin{aligned} semantic: \mathcal{M}, \pi^{\mathbf{t}} \ \models \ &\textbf{Globally, } \phi \ \Leftrightarrow \\ &\forall \pi^k, \ t \leq k \leq n \ \mathcal{M}, \pi^k \ , \models \ \phi, \\ CTL\ formula: \ &AG\,\phi \\ LTL\ formula: \ &G\,\phi \end{aligned} \tag{28}$$

The quantifier A in CTL formulas can be replaced by *Quantifier Scope*.

- **In the future,** ϕ: this scope expresses that ϕ must hold by at least one part of trace that starts from the state that ϕ is considered. The semantic of this pattern is

$$\begin{aligned} semantic: \mathcal{M}, \pi^{\mathbf{t}} \ \models \ &\textbf{In the future, } \phi \ \Leftrightarrow \\ &\exists \pi^k, \ t \leq k \leq n \ \mathcal{M}, \pi^k \ \models \ \phi, \\ CTL\ formula: \ &AF\,\phi \\ LTL\ formula: \ &F\,\phi \end{aligned} \tag{29}$$

The quantifier A in CTL formulas can be replaced by *Quantifier Scope*.

- **In the next cycle,** ϕ: this scope expresses that ϕ must hold by the trace that starts from next state we specify. The semantic and LTL, CTL format specifications of this scope are

$$\begin{aligned} semantic: \mathcal{M}, \pi^{\mathbf{t}} \ \models \ &\textbf{In the next cycle, } \phi \ \Leftrightarrow \ \mathcal{M}, \pi^{t+1} \ \models \ \phi \\ CTL\ formula: \ &AX\,\phi \\ LTL\ formula: \ &X\,\phi \end{aligned} \tag{30}$$

The quantifier A in CTL formulas can be replaced by *Quantifier Scope*.

Cycle Scope expresses the effective extent in the format of cycles. Three scopes belong to this kind. This kind of scopes precisely define the scope of the requirements.

- **Every m cycles,** ϕ: this scope states that ϕ must hold every n states from the specific state. The semantic and LTL, CTL format specifications of this scope are

$$\begin{aligned} semantic: \mathcal{M}, \pi^{\mathbf{t}} \ \models \ &\textbf{Every m cycles, } \phi \ \Leftrightarrow \\ &\forall\, i \text{ satisfies } t + i * m \leq n \ \wedge i \geq 0, \ \mathcal{M}, \pi^{t+i*m} \ \models \ \phi \\ CTL\ formula: \ &\phi \wedge AG(\phi \rightarrow AX!\phi \wedge AXAX!\phi \cdots \wedge \underbrace{AX \ldots AX}_{m}\phi) \\ LTL\ formula: \ &\phi \wedge G(\phi \rightarrow X!\phi \wedge XX!\phi \cdots \wedge \underbrace{X \ldots X}_{m}\phi) \end{aligned} \tag{31}$$

The quantifier A in CTL formulas **cannot** be replaced by *Quantifier Scope*.

- **After m cycles,** ϕ: this scope expresses that ϕ must hold in the future before the system ending after m states starts from specific state in the running trace of the system. The semantic and LTL, CTL formulas of this scope are

$$\begin{aligned} semantic: \mathcal{M}, \pi^{\mathbf{t}} \models\ & \textbf{After m cycles,} \phi \Leftrightarrow \\ & \exists\, k \text{ satisfies } k > t + m, \mathcal{M}, \pi^{k} \models \phi \\ CTL\ formula: & \underbrace{AX \ldots AX}_{m} AF\, \phi \\ LTL\ formula: & \underbrace{X \ldots X}_{m} F\, \phi \end{aligned} \tag{32}$$

  The quantifier A in CTL formulas **cannot** be replaced by *Quantifier Scope*. If ϕ is modified by “Globally”, then F will be replaced by G.
- **In the next m-th cycles,** ϕ: this scope expresses that ϕ must hold by the trace start from the state that n states away from the state we specify. The semantic and LTL, CTL formulas are defined as follows:

$$\begin{aligned} semantic: \mathcal{M}, \pi^{\mathbf{t}} \models\ & \textbf{In the next m-th cycles, } \phi \Leftrightarrow \\ & \exists\, k \text{ satisfies } k = t + m,\ \mathcal{M}, \pi^{k} \models \phi \\ CTL\ formula: & \underbrace{AX \ldots AX}_{m}\ \phi \\ LTL\ formula: & \underbrace{X \ldots X}_{m}\ \phi \end{aligned} \tag{33}$$

  The quantifier A in CTL formulas **cannot** be replaced by *Quantifier Scope*.

State Scope is used when using state machine to model a system. This kind of scopes describe the state or mode of the system that the requirements should be considered. We defined three scopes belonging to this kind. *mode* denotes a subsystem and *state* denotes a state in state machine which is different from the **s** in the running trace of a system.

- **In the state A,** ϕ: this scope states that ϕ must hold at the point **s** if the system is in the state A. The semantic of this scope is

$$\mathcal{M}, \mathbf{s} \models \textbf{In the state A, } \phi \Leftrightarrow (state = A) \in \mathbf{L}(\mathbf{s}) \rightarrow \phi \in \mathbf{L}(\mathbf{s}) \tag{34}$$

- **In the mode A,** ϕ: this scope states that ϕ must hold at the point **s** if the system is in the mode A. The semantic of this scope is

$$\mathcal{M}, \mathbf{s} \models \textbf{In the mode A, } \phi \Leftrightarrow (mode = A) \in \mathbf{L}(\mathbf{s}) \rightarrow \phi \in \mathbf{L}(\mathbf{s}) \tag{35}$$

- **When entering the state A,** ϕ: this scope expresses that if the system enter the state A in the next cycle, ϕ must hold in this cycle. The semantic are defined as follows:

$$\begin{aligned}\mathcal{M}, \mathbf{s_t} \models \textbf{When entering the state A, } \phi \Leftrightarrow \\ (state = A) \notin \mathbf{L}(\mathbf{s_t}) \wedge (state = A) \in \mathbf{L}(\mathbf{s_{t+1}}) \\ \wedge\ \phi \in \mathbf{L}(\mathbf{s_t})\end{aligned} \tag{36}$$

If users want to let ϕ holds in the next cycle, then can use “In the next cycle,” to modify ϕ.

Table 7. The Map between English Keywords and Chinese Keywords of Scope Pattern

English Keywords	Chinese Keywords
For all trace,	在所有路径上,
Exists one trace,	存在某些路径,
Globally,	全局范围内,
In the future,	存在某个时间点,
In the next cycle,	在下个周期,
Every m cycles,	每m个周期,
After m cycles,	在m个周期后,
In the next m-th cycles,	在第m个周期时,
In the state A,	在A状态中,
In the mode A,	在A模态中,
When entering the state A,	进入a状态时,

Appendix 4

In this section, we introduce the algorithms for translating SCADE models into NuXMV models.

Algorithm 1 shows the pseudo-code of building the main tuple $< I, O, L, SM, M, Func >$ that record the syntax information of the source SCADE models. I, O and L respectively denotes the set of input, output and local variables of the SCADE model. SM denotes the safety state machine, $Func$ represents data flows and M denotes the monitor module. The main tuple will be used to generate the four types modules in the target nuXmv model.

Algorithm 1 Building Main Tuple

```
Input: A SCADE Textual Model
Output: main < I, O, L, SM, M, Func >
I ← the Input variables of this Model
O ← the output variables of this Model
L ← the local variables of this Model
create empty sets: main, SM, M, Func
Parser_Tree ← the grammar tree of the input SCADE model
for each node on the Parser Tree do
  if the type of node is safety state machine then
    state_machine ← the structure of the safety state machine of this model
    create empty set VAR, MVAR, TRANS, SUB_SM
    VAR ← all the variables that control the state transitions in state_machine
    for each state of state_machine do
      TRANS ← the transition of state
      if state has a sub state machine sub_SM_state then
        SUB_SM ← sub_SM_state
      end if
      if state assign the value of variable A ∧ A ∈ (O ∪ L) then
        Mvar ← create Monitor variable set_variable_state
        Mvar ← create Monitor variable reset_variable_state
        action ← the assignment expression of A
        MVAR ← Mvar
        M ← < A, Mvar, action >
      end if
    end for
    SM ← < VAR, MVAR, TRANS, SUB_SM >
  end if
  if the type of node is pure data flow then
    create empty set VAR, ACTION
    VAR ← all the parameters of this node
    ACTION ← the expression of the data flow in this node
    Func ← < VAR, ACTION >
  end if
end for
main ← < I, O, L, SM, M, Func >
return main
```

Generating *Main* Module Algorithm 2 shows the pseudo-code for generating *Main* module in the target nuXmv model generated by the SCADE2nuXmv method mentioned in the paper. This module instantiates the interface of the SCADE model, the local variables, all monitor variables and modules, and the top state machine. It also assigns the initial value of all monitoring variables. In the algorithm, $\emptyset$ denotes the empty set and "*print*" means generate the keyword that follows it. The input of this algorithm is the tuple constructed in algorithm 1, the output is the *Main* module of the target nuXmv model.

Algorithm 2 Generation of *Main* Module

Input: main $= < I, O, L, SM, M, Func >$
Output: main Module

print **VAR** keyword
for each $input_i \in I$ **do**
    declaration of $input_i$
end for
for each $local_i \in L$ **do**
    declaration of $local_i$
end for
for each $output_i \in O$ **do**
    declaration of $output_i$
end for
if $SM \neq \emptyset$ **then**
    declaration of $default$ and $active$
    for each $M_i \in M$ **do**
        declaration of $set\_variable\_state$
        declaration of $reset\_variable\_state$
        instantiation of M_i
    end for
    instantiation of top State Machine
    print **ASSIGN** keyword
    initialization and assignment of $default$
    assignment of $active$
    for each $M_i \in M$ **do**
        initialization of $set\_variable\_state$
        initialization of $reset\_variable\_state$
    end for
else
    instantiation of top function
end if

Generating *State Machine* Module Each *State Machine* module corresponds to one state machine in the original SCADE model. To generate *State Machine* modules, we have defined five generating rules to its functions, as shown in Fig.10, where the variables are defined as follows:

- *default_ SubSM_ S* and *active_ SubSm_ S* represent the "*default*" and "*active*" (mentioned in section 5) utilized to implement the hierarchical structure in the SCADE model.
- *S* denotes the state in the state machine *SMi*, which has sub-state machine.
- s_0 denotes the initial state of the state machine *SMi*.
- *preS* denotes the states that can transitions to state *S*.
- *T(preS, S)* denotes the transition condition of the transition from *preS* to *S*.
- *set_ V_ SM_ S* and *reset_ V_ SM_ S* represent the monitor variables that monitor the output variable V.

```
MODULE SMi(active, default, ...)
VAR
    state := {…, S, …, InactiveState};
                                                        Rule 1
    default_SubSM_S : boolean;
    active_SubSM_S : boolean;
    Sub_SubSM : SubSM(default_SubSM_S, active_SubSM_S, …);
ASSIGN
                                                        Rule 2
    init(default_SubSM_S) := FALSE;
    next(default_SubSM_S) := (active & next(active)
        & (state = preS) & (next(state) = S)) ? TURE : FALSE;
    active_SubSM_S := active & (state = S);
                                                        Rule 3
    init(state) := (default & actve) ? s0 : InactiveState;
    next(state) := case
        (next(default) & next(active) & !active) : s0;
        !next(active) : InactiveState;
        active & next(active) & (state = preS) & T(preS, S) : S;
        …
    esac;
                                                        Rule 4
    next(set_V_SM_S) := active & next(active) & (state = S)
        & (!T(preS1, S) & … & !T(preSn, S)) : TRUE : FALSE;
                                                        Rule 5
    next(reset_V_SM_S) := case
        !active & next(active) & !default & next(default) : TRUE;
        active & next(active)
            & (state = preS1 | … | state = preSn)
            & (next(state) = S) : TRUE;
        TRUE : FALSE;
    esac;
```

Fig. 10. Generation Rules for the *State Machine* Module

Rule 1 defines the declaration of sub-state machines *Sub_ SubSM*, *default_ SubSM_ S*, and *active_ SubSm_ S*. The assignment of *default_ SubSM_ S* and *active_ SubSm_ S* are described in *Rule 2*. Please note that if *S* is in the initial state of SM_i, the initial value of *default_ SubSM_ S* is set to **TRUE**. *Rule 3* describes the state transition in *SMi*. The *InactiveState* is a state denotes that *SMi* is inactive. Please note the order of the "case" branches should obey the priority of the transition in the SCADE model. *Rule 4* and *Rule 5* describe the assignments to the monitor variables. When all transition conditions T(*preS*, *S*) are not met, $set\_V\_SM\_S$ IS **TRUE**. Otherwise it is set to **FALSE**. When the state transitions from *preS* to *S*, the $reset\_V\_SM\_S$ should be **TRUE**. Please note that in cases where *S* is the initial state of *SMi*, the $reset\_V\_SM\_S$ is *TRUE* when *SMi* becomes active.

With the five rules, we define algorithm 3 to describe the process of generating *State Machine* module. The *SMi* corresponds to a safety state mahine in the SCADE model. The input of the algorithm is the *main* tuple generated by algorithm 1. The *VAR* denotes the variables that control the state transitions, and *TRANS* denotes all the state transition in *SMi*. *SUB_ SM* is the set of sub-state machines of *SMi*. *MVAR* denotes the monitor variables related to *SMi*.

Algorithm 3 Generation of *State Machine* Module

Input: main = $< I, O, L, SM, M, Func >$
Output: State Machine Module
for each $SM_i < VAR, MVAR, TRANS, SUB\_SM > \in SM$ **do**
  print **MODULE** SM_i*(**default**, **active**, VAR, MVAR)*
  if $SUB\_SM \neq \emptyset$ **then**
    for each $SubSM_i \in SUB\_SM$ **do**
      print **VAR** keyword
      declaration of $default\_SubSM_i\_S$ by Rule 1
      declaration of $active\_SubSM_i\_S$ by Rule 1
      instantiation of $SubSM_i$ by Rule 1
      print **ASSIGN** keyword
      initialization of $default\_SubSM_i\_S$ by Rule 2
      assignment of $default\_SubSM_i\_S$ by Rule 2
      assignment of $active\_SubSM_i\_S$ by Rule 2
    end for
  end if
  if $TRANS \neq \emptyset$ **then**
    print **VAR** keyword
    declaration of state
    print **ASSIGN** keyword
    initialization of state by Rule 3
    state transitions generation by Rule 3
  end if
  if $MVAR \neq \emptyset$ **then**
    for each $MVar_i \in MVAR$ **do**
      print **ASSIGN** keyword
      assignment of $set\_var\_SM_i\_S$ by Rule 4
      assignment of $reset\_var\_SM_i\_S$ by Rule 5
    end for
  end if
end for

Generating *Monitor* Module Every *Monitor* module manipulates the assignment of an output or local variable of the top state machine in the SCADE model via monitor variables. The *Monitor* module executes the branch where the monitor variable is **TRUE**. As shown in Fig.11, each *Monitor* module incorporates two expressions, each with multiple execution branches, to assign the value of the monitored variable. One initializes the monitored variable, while the other updates it after the first cycle.

```
MODULE Set_V(V, …)
VAR
    func_function1 : function(…);
    …

ASSIGN
    -- Rule 6
    init(V) := case
        set_V_SM_S : Init_val(V);
        reset_V_SM_S : ResetValue(V);
        …
        TRUE : Init_val(V);
    esac;

    -- Rule 7
    next(V) := case
        next(set_V_SM_S) : V;
        next(reset_V_SM_S) : NextResetValue(V);
        …
        TRUE : NextDefVal(V);
    esac;
```

Fig. 11. Generation Rules for the *Monitor* Module

Rule 6 describes the generating rules for the initial value assignments. The *Init_ val* function is used to get the default initial value of the variable V, which returns one of the following three values: *default_ V*, *last_ V* and *Zero_ V*. *default_ V* represents the default property of V and *last_ V* represents the last attribute of V in the SCADE model. *Zero_ V* varies according to the data type. When V is a numeric variable, the *Zero_ V* is 0. *Zero_ V* is set to **FALSE**, when the data type of V is Boolean. The *Init_ val* returns value according to the priority. The priority relationship between these three values is $default\_V > last\_V > Zero\_V$. The *ResetValue* function is the same as the assignment statement in the SCADE model which is utilized by state S in state machine SM to assign the value of variable V.

The generation rule for the expression used to assign a value to the variable V after the first cycle is described in *Rule 7*. The *Monitor* module keeps the value of V when the system cannot transition from state S to other states. The module utilize *NextResetValue* to assign the value of V as the system transition from other states to state S. The *NextResetValue* is similar to *ResetValue* but with each variable enclosed by a next operator. In cases where all the monitor variables are not met, the *Monitor* module utilizes *NextDefVal* to manipulate the value. *NextDefVal* returns one of the following two values: V or *default_ V*.

When the variable V has default property in the SCADE model, *NextDefVal* will return the *default_ V*, otherwise it returns V.

With *Rule 6* and *Rule 7*, we define the generation algorithm of the monitoring module, as shown in algorithm 4. The V denotes the monitored variable of the *Monitor* module, $MVAR$ denotes the monitor variables related to V and ACT represents all assignment expressions of V. The temporary variables represents the connection lines in the SCADE model.

Algorithm 4 Generation of *Monitor* Module

Input: main = $< I, O, L, SM, M, Func >$
Output: Monitor Module
for each M_i $< V, MVAR, ACT > \in M$ **do**
  print **MODULE** $M_i < V, MVAR >$
  create a empty set $FUNC$
  for each $func_i$ in $Func$ **do**
    if $func_i$ is called by M_i **then**
      $FUNC \leftarrow func_i$
    end if
  end for
  if $FUNC \neq \emptyset$ **then**
    print **VAR** keyword
    for each $func_i \in FUNC$ **do**
      instantiation of $func_i$
    end for
  end if
  print **ASSIGN** keyword
  initialization of V by Rule 6
  assignment of V by Rule 7
end for
reduce the temporary variables with their equivalent variables

Generating *Function* Module A *Function* module corresponds to a data flow operator in the original model. Each output variable of it is defined by two expressions in the corresponding *Function* module. One, the same as the expression in the SCADE model, assigns its initial value. The other, similar to the expression in the SCADE model but with each variable enclosed by a *next* operator, assigns the value of the variable after the first cycle. Algorithm 5 shows the generation rule for *Function* module. In the algorithm ACT represents the set of assignment expressions of output variables.

Algorithm 5 Generation of *Function* Module

Input: main $= < I, O, L, SM, M, Func >$
Output: Function Module

for each $Func_i < I, O, L, ACT > \in Func$ **do**
  print **MODULE** $Func_i(I)$
  print **VAR** keyword
  for each $output_i \in O$ **do**
    declaration of $output_i$
  end for
  for each $local_i \in L$ **do**
    declaration of $local_i$
  end for
  for each $func_i \in Func$ **do**
    declaration of $func_i$
  end for
  print **ASSIGN** keyword
  for each $output_i \in O$ **do**
    instantiation of $output_i$
    for each $act_i \in ACT$ **do**
      if act_i assigns $output_i$ **then**
        print **init(**$output_i$**)** $:= act_i$
        $act_i \leftarrow$ enclose every variable in act_i with *next* operator
        print **next(**$output_i$**)** $:= act_i$
        break this for expression
      end if
    end for
  end for
  for each $act_i \in ACT$ **do**
    if act_i is not printed **then**
      print act_i
    end if
  end for
  reduce the equivalent variables
end for