Modelling and Verification of Natural Language Requirements based on States and Modes

Yinling Liu, Jean-Michel Bruel

Modelling and Verification of Natural Language Requirements based on States and Modes

Abstract

Natural language requirements (NLRs) modeling and verification allow us to detect errors in requirements as early as possible in an effective way. The NLRs of system development usually involve states and modes either explicitly or implicitly. Different stakeholders have different understandings of these two terms. The misunderstanding of states and modes in NLRs severely impedes requirements modeling and validation. For example, conflicts in system validation may occur when users and developers do not share a consensus on the terms. Modeling and verification of requirements from the viewpoint of states and modes have never been investigated. Thus, in this paper, an innovative approach to analyzing requirements is proposed. To this end, an overview of states and modes in literature is performed to help us understand the relationship between them. The MoSt language (a Domain Specific Language, DSL) is then designed for requirements modeling and implemented by Xtext framework. Meanwhile, a model validator is realized to define user-defined rules, in order to statically check requirements. In the following, a code generator is also accomplished to realize the automatic model transformation from the MoSt model to the NuSMV model, which aims at the dynamic checks of requirements. The grammar, the model validator, and the code generator are integrated into an Eclipse-based tool which is available on GitHub. A case study on requirements for designing cars has been conducted to illustrate the feasibility of our approach. In this case study, we injected 11 errors. The results show that 6, 4, and 1 were detected in the static analysis, traceability analysis, and completeness analysis of the model respectively. Finally, the feasibility

of our approach to reducing conflicts between user and developers has been illustrated in validation analysis.

Keywords: States and Modes, Requirements Modeling and Verification, Domain Specific Language, Model checking.

---

1. Introduction

Writing complete, unambiguous, and conflict-free requirements is not an easy thing. This demands not only effective approaches to model and verify the written requirements but appropriate perspectives to analyse requirements. Two terms "states" and "modes" have been widely used in expressing requirements. For example, the description of systems usually involves various states and modes, according to many system development methodologies (Olver and Ryan 2014). Systems engineering standards (DI-IPSC-81431A 2000; DFS 2007; DMO 2011) require the use of states and modes in formal systems engineering. However, little guidance has been proposed to distinguish states and modes. As Wasson said, "System modes and states are perhaps one of the most controversial topics in Engineering and SE. Every industry, profession, Enterprise, and Engineer has their own view as to what a mode and a state are." (Wasson 2015). Poor usage of the terms severely impedes the modeling and verification of requirements. This is because views on states and modes vary from person to person, which gives rise to conflicts when analyzing requirements. Therefore, in this paper, we focus on how to model and verify requirements extracted from requirements documents using the viewpoint of states and modes, in order to write high-quality requirements.

Various aspects have been emphasized to analyse requirements, including context (Ali, Dalpiaz, and Giorgini 2013; Ahmad, Belloir, and Bruel 2015), the failures and successes of other requirements (Silva Souza et al. 2011), and requirements evolution (Whittle et al. 2009). However, to the best of our knowledge, no one performs the modeling and verification of requirements based on states and modes. The requirement analysis benefits a lot from the pro-

per usage of states and modes. They enable us to describe requirements that exist outside the normal operating environment (Olver and Ryan 2014). They also help translate the user's version into the physical realization of the system (Wasson 2010). They can be used as a medium to reduce misunderstandings between stakeholders such as users, acquirers, and developers as well (Edwards 2003).

Domain-Specific Languages (DSLs) are programming languages or specification languages that target a specific problem domain Bettini (2016). When the domain of one problem is covered by a particular DSL, we will solve that problem in an easier and faster way via using that DSL rather than a general-purpose language like Java or C, etc. In our case, we aim to create a new DSL to help us write requirements in a controlled natural language. In this way, requirements can be better organized, expressed, and understood. On the other hand, writing requirements in natural languages is easier and more acceptable for stakeholders.

Proper DSLs are helpful in writing "correct" requirements. They are just the requirements that satisfy syntactic rules and validator rules. Validator rules are self-defined. For example, naming rules of the elements in requirements can be defined in the validator then the requirements can be checked by the defined naming rules. Several problems still remain unsolved. For instance, how to check the internal logic between requirements? How to verify the satisfaction of property specifications regarding systems? To solve these problems, the symbolic model checking technique can be used to further improve the quality of the written requirements. This technique is capable of demonstrating the correctness of system behaviours. The problem of model checking is formally expressed by $M \models \varphi$, where M represents the system model, φ is a property, and $\models$ is the satisfaction symbol to check whether the model M satisfies the property φ. If the property is not satisfied by the system, a counterexample is produced. NuSVM [1] is a symbolic model checker designed to allow

1. https://nusmv.fbk.eu/ (accessed in May 2021)

for the description of Finite State Machine (FSM) which ranges from completely synchronous to completely asynchronous, and from the detailed to the abstract (Cimatti et al. 2002). The primary purpose of NuSMV input language is to describe the transition relation of the FSM, which is quite suitable for describing the state information in requirements. Therefore, in this paper, we choose NuSMV as the model checker.

Furthermore, a framework for the modeling and verification of requirements is firstly provided, in order to explain how to improve the quality of requirements in an iterative way. Based on this framework, a new DSL is designed to better describe requirements from the viewpoint of states and modes. A code generator is then implemented to transform the DSL model into the NuSMV program. In the end, the requirements of designing a car are used to carry out the requirement verification, which illustrates the feasibility of the proposed approach. The main contributions of this paper are listed as follows :

— Proposing an innovative framework fully supporting the modeling and verification of requirements based on states and modes ;
— Designing the MoSt modeling language, including the language metamodel, formal syntax, and semantics to formalize the requirements based on states and modes easier ;
— Accomplishing algorithms to automatically perform the model transformation from the MoSt model to the NuSVM model ;
— Implementing an Eclipse-based tool to enable us to write the MoSt model, generate the NuSMV model and conduct the static and dynamic requirement analysis.

Based on the tool, requirements engineers can use the MoSt modeling language to formalize requirements, so as to better organize, accurately express, and effectively manage requirements. The extracted information on states and modes can serve as ”standard” terms when team members communicate with each other. So, unnecessary conflicts on the system description can be avoided. In addition, clients can be encouraged to expect the most suitable performance of the future systems since the MoSt language supports the description of the

specification in CTL and LTL logic, and these specifications can be verified in the NuSMV model checker.

The remainder of this paper is structured as follows. Section 2 reviews the main related work. Section 3 introduces the framework for requirements analysis and presents all the elements about how to design the MoSt modeling language. Section 4 realizes the algorithms to perform the model transformation from the MoSt model to the NuSMV model. Section 5 conducts static and dynamic requirements verification to illustrate the feasibility of the proposed approach. Section 6 concludes the paper with future perspectives.

2. Literature Review

2.1. States and Modes

This subsection provides a comprehensive literature review on the definition of states and modes. As one of our purposes is to differentiate states and modes, we exclude the references (Committee et al. 1990; DI-IPSC-81431A 2000; Edwards 2003; DMO 2011) which are self-inconsistent in terms of the description of states and modes. For example, DI-IPSC-81431A (2000) offers the following guidance : *The distinction between states and modes is arbitrary. A system may be described in terms of states only, modes only, states within modes, modes within states, or any other scheme that is useful*. Edwards (2003) shares the similar idea of the relationship between states and modes. Obviously, this guidance creates more conflicts in differentiating states and modes. Fortunately, they finally cancel this data item. On the other hand, DMO (2011) provides an example of a state transition diagram which does not depict any states at all. The synthesis analysis of the reviewed references is concluded in Table 1. A set of aspects have been proposed to analyse the definitions of states and modes, include state (*information*, *abstraction level*, *conditions*, *capabilities*, *dynamics*, *constraints*) and mode (*abstract concept*, *abstraction level*, *conditions*, *capabilities*, *dynamics*, *objectives*). It should be noted that *abstract concept* and *abstraction level* are different. The former implies denoting an idea of a thing rather than a concrete object ; the latter means the explanation will be changed due to the

concerns on the level of the abstraction of the system. Furthermore, "involved" implies the term has been just mentioned without any other details.

According to this table, the concepts of states and modes had been widely used from 2000 to 2010. However, it seems that researchers didn't pay enough attention to the difference between them. For example, Andrey (2002); Feiler, Lewis, and Vestal (2006) have simply emphasised states and modes respectively. Since 2010, the issue of the difference between states and modes has been gradually addressed, but opinions on states and modes still vary from person to person. In terms of states, researchers principally concentrate on the aspects of *conditions*, *dynamics* and *constraints*. They focus less on *information* and *abstraction level*. The aspect of *capabilities* is seldom addressed. When comparing states with modes, *capabilities*, on the contrary, is the most important aspect in defining modes. Researchers are also concerned with *abstraction level*, *conditions*, *dynamics* and *objectives* for modes. It seems that they do not care whether "mode" is an abstract concept or not. This analysis helps us gain important insights into the meaning of states and modes, which lays the foundation for designing a DSL containing these two concepts.

2.2. Requirements Modeling and Verification

Requirements analysis has been recognized as the first phase of the system development process (Royce 1987). The later errors in the requirements are discovered, the higher the cost of the system development. Hence, the importance of requirements modeling and verification has been well addressed in systems engineering and software engineering. A considerable amount of work on how to model and verify requirements has been performed. Numerous approaches have been proposed to improve the quality of requirements, including designing a new language, using a goal-based model, manipulating tools, and implementing a tool, etc. The following will conduct a comprehensive analysis of the literature.

Leveson et al. (1994) develop a Requirements State Machine Language (RSML) to describe requirements of real-time process control systems via a combina-

Table 1: An overview of the definitions of states and modes where "-" and "NM" means "Not included" and "Not Mentioned" respectively

		Andrey (2002)	Davis et al. (2005)	Feiler, Lewis, and Vestal (2006)	DFS (2007).	IEEE 24765 et al. (2010)	Wasson (2010)	Jenney (2011)	Buede and Miller (2016)	Bonnet et al. (2017)	Baduel (2019)
State	information	an object	NM	-	NM	variables, system	system	NM	involved	NM	type of information
	abstraction level	NM	system, subsystem	-	system, subsystem.	NM	element, subsystem, system	NM	NM	NM	relates to information
	conditions	NM	system condition	-	system conditions.	a condition to a behaviour	performance, physical	operating	a set of metrics	operating, physical	involved
	capabilities	NM	involved	-	NM.	multiple functions	involved	NM	involved	NM	NM
	dynamics	NM	NM	-	involved	NM	involved	state transition	state transition	state transition	involved
	constraints	time, space	NM	-	time	NM	involved	NM	time	environment	time
Mode	abstract concept	-	NM	detailed	involved	NM	abstract label	NM	NM	NM	abstract
	abstraction level	-	NM	system, subsystem	sub-mode	NM	system, product, service	top, lower	NM	NM	link to capabilities
	conditions	-	system condition	specific conditions	NM	NM	triggering events	a condition of a system	NM	specific conditions	invariant conditions
	capabilities	-	multiple capabilities	NM	operations	multiple capabilities	use case	multiple capabilities	operational	specific functioning	multiple capabilities
	dynamics	-	NM	mode transition	mode transition	NM	mode transition	NM	NM	mode transition	mode transition
	objectives	-	function	NM	NM	NM	mission	NM	function	expected behaviour	expected behaviour

tion of graphical and tabular notations. The graphical notation of RSML is principally derived from Statecharts (Harel 1987). A number of definitions have been mentioned in RSML, including *Interface*, *Input*, *Output*, *Transition*, *Macro* and *Function*. The tables in RSML describe the conditions under which the corresponding state transitions can happen. Indeed, they analyze requirements from the viewpoint of states but their way of expressing requirements complicates requirements analysis.

Heitmeyer, Jeffords, and Labaw (1996) demonstrate the feasibility of formal methods for requirements modeling and analysis via three case studies. They choose different formal methods for the requirements of different case studies. They have successfully detected errors in requirements, thanks to requirements modeling, testing, verification, and initial human reading. Even though a number of errors have been identified, this work is done by experts in formal methods. Their approach requires a lot of knowledge in formal methods, which limits its use.

Goldsby et al. (2008) analyse requirements from the viewpoint of four types of developers : the system developer, the adaptation scenario developer, the adaptation infrastructure developer, and the dynamically adaptive system. They use i* goal models (Yu 1997) to describe the requirements of different types of developers. They detail the viewpoint of the developer for requirements analysis, but the graphic representation of requirements is hard to be formally verified.

Mavin et al. (2009) focus on the problem of how to design a structured natural language to improve the quality of requirements written by stakeholders. For this purpose, they develop five specific Easy Approach Requirements Syntax (EARS) templates to facilitate requirements expressing. The results of their case study show qualitative and quantitative improvements compared with a conventional textual requirements specification. However, their work remains on requirements expressing and further requirements analysis including conflicting requirements and traceability links, etc., has not been mentioned.

Silva Souza et al. (2011) present a new type of requirements called AW (Awareness Requirements). The requirements are associated with other requirements and their success/failures, constituting requirements for such feedback loops. They formalize AW by a variant of OCL (Object Constraint Language) called OCL_{TM} and validate them using a monitoring framework. Since the modeling process with OCL_{TM} is not a trivial task, they provide AW patterns and graphic representation to facilitate the elicitation and analysis of AW. However, in our case, we consider the facility of eliciting requirements by creating our own modeling language directly.

Requirements uncertainty has been studied in certain work. Some focus on uncertainty, assuming all the uncertain conditions are unknown and enumerated at design time (Letier and Van Lamsweerde 2004; Goldsby et al. 2008). Some address that some uncertain conditions are still unanticipated (Whittle et al. 2009). Whittle et al. (2009) design a new requirements specification language called RELAX to explicitly emphasize uncertainty without knowing all the uncertain conditions. RELAX is based on FBTL (Fuzzy Branching Temporal Logic), which can describe a branching temporal model with uncertain temporal and logical information. So, its expressive power on uncertainty is stronger than some other languages. The idea of designing a DSL to write certain requirements is similar to ours, but we are not specifically concentrated on requirements uncertainty.

Badger, Throop, and Claunch (2014) argue that a simple way to improve the quality of requirements written at various levels by different groups of people would be to standardize the design process using a set of tools and widely accepted requirements design constraints. They make full use of appropriate tools to realize the automatic requirements elicitation, formalization, analysis, and verification. However, they do provide us with a concrete case study. The limits of used tools have not been discussed.

Ahmad, Belloir, and Bruel (2015) propose a model-based requirements modeling and verification process for addressing uncertainty in the requirements of self-adaptive systems. They combine the proposed language RELAX with

the concepts of Goal-oriented Requirements Engineering for requirements eliciting and modeling. Since various tools have been used like RELAX editor, SysML/KAOS, and OMEGA2, requirements traceability is not sufficiently emphasized.

Moitra et al. (2019) implement a tool ASSERT^{TM} to perform requirements modeling and verification. The requirements are captured by a structured natural language and formal analysis is based on an automated theorem prover. They conduct a set of formal requirements analysis, including completeness analysis. The completeness analysis of ASSERT^{TM} is simply involved with values of monitored variables and all pairs of values for controlled variables. They do not consider the reachability of all states of a system. Their completeness analysis should be categorized because there is no need to write requirements about different combinations of values of variables when a system in *OFF* state. On the other hand, their viewpoint of analyzing requirements is dependant on data instead of states and modes.

Recently, Nalchigar, Yu, and Keshavjee (2021) are interested in machine learning requirements (the processes for requirements elicitation, design, development involve machine learning). They analyze these requirements from the viewpoint of business people, data scientists, and data engineers. They use the case study method to perform requirements modeling. Their approach is suitable for testing theories and artifacts in complex settings, however, it is principally a manual approach, which is difficult to be sufficiently validated.

Giannakopoulou et al. (2021) present a compositional approach to generating and verifying the formalization of structured natural language. They develop a Formal Requirements Elicitation Tool (FRET) to write, understand, formalize and analyze requirements. They also develop an automated verification framework for the fmLTL (future-time LTL) and pmLTL (past-time LTL) formulas. However, requirements consistency checks are missing. They just use EQUIVALENCE_CHECKER to check the consistency between different formalizations of the same template key. It seems that they treat requirements independently. The relationship between requirements should be addressed.

Clearly, a majority of work focuses on how to design a new language to facilitate requirements analysis. The new languages proposed include RSML, EARS, OCL_{TM}, RELAX, FRET, etc. Most of the languages are dedicated to better requirements eliciting and verification. Different viewpoints of analyzing requirements have been considered, including developers (Goldsby et al. 2008), uncertainty (Whittle et al. 2009; Ahmad, Belloir, and Bruel 2015), awareness requirements (Silva Souza et al. 2011), machine learning requirements (Nalchigar, Yu, and Keshavjee 2021), etc. Some other researchers are also concentrated on data warehouse requirements (Zepeda et al. 2010; El Beggar, Letrache, and Ramdani 2020). To the best of our knowledge, none of the work analyzes requirements from both users and developers. In other words, the viewpoint of states and modes has not been sufficiently addressed in analyzing requirements. This may lead to misunderstandings between users and developers, which can cause conflicts in systems validation. Inconsistency could happen in development teams as well, which gives rise to conflicts in system design. As a result, we are so motivated to conduct requirements analysis from the viewpoint of states and modes.

3. MoSt Modeling Language

3.1. Relationship between States and Modes

Section 2.1 made a comprehensive analysis on the concepts of states and modes, in this section, we will discuss the relationship between states and modes in detail.

The relationship between states and modes has been discussed in the literature. For example, the explanation of Edwards (Edwards 2003) may be confusing, but the example on the relationship between states and modes implies modes control states. In other words, modes can actively influence system states. Modes show more capabilities. While states are changed when conditions are satisfied. It seems that Wasson (Wasson 2015) shares the same idea. He argues states are observable and measurable physical attributes of a system or entity. It means states represent the attributes of a system or entity. And

he suggests modes enable us to accomplish objectives that produce results you can observe and measure. The activeness of modes is again recognized. The characteristics of modes and states are the basis for us to propose the relationship between them.

In this paper, we propose our proper definitions of modes and states. We argue modes are the abstraction of use cases as Wasson (2015) mentioned. Modes transitions happen when the corresponding signals from the system are received. Modes own capabilities to change the values of certain attributes. The values of these attributes are ones of conditions inside states. States hold certain conditions. States transitions happen when the corresponding conditions are satisfied. Fig .1 illustrates the relationship between states and modes.

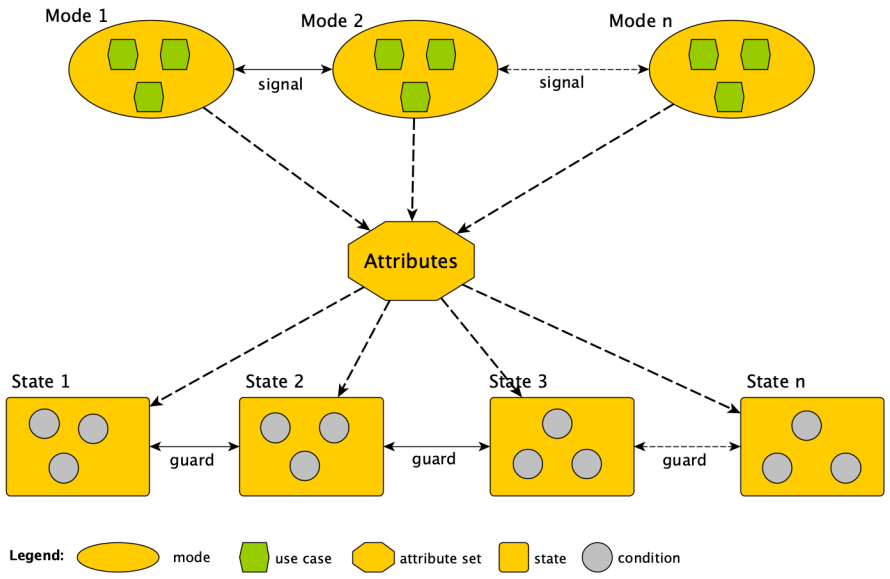

Figure 1: The Relationship between States and Modes

3.2. Our Analysis Framework

This framework aims to explain our approach to modeling and verifying requirements. Three steps are involved, including requirements formalization, model transformation, and model verification. In the first step, we will design

a DSL called MoSt to formalize requirements from requirements documents. Meanwhile, a code validator is implemented to impose self-defined rules on this language so that requirements can be statically checked in the MoSt editor. In the second step, a code generator will be accomplished to automatically realize the model transformation from the MoSt model into the NuSMV model. This step provides the foundation for dynamically checking requirements. The last step is to conduct model verification. The errors or counter-examples proposed by the model checker will be traced back to the MoSt model. Thus, we can improve the quality of requirements with the information proposed by the model checker.

Our framework for requirements modeling and verification clarifies the value of our work. Firstly, it offers a general approach to integrating states and modes into requirement modeling and verification in the early phase of system design. System designers who are accustomed to using terms like states and modes will benefit a lot from this work because the relationship between states and modes will be precisely explained. Secondly, system designers who are intended for getting a sense of what the future system will look like will profit from this work because modes and the combination of modes can be checked with model checkers. Finally, it allows system designers to better communicate with clients since "mode" is a common term which is easier to be accepted by clients when discussing the specific needs.

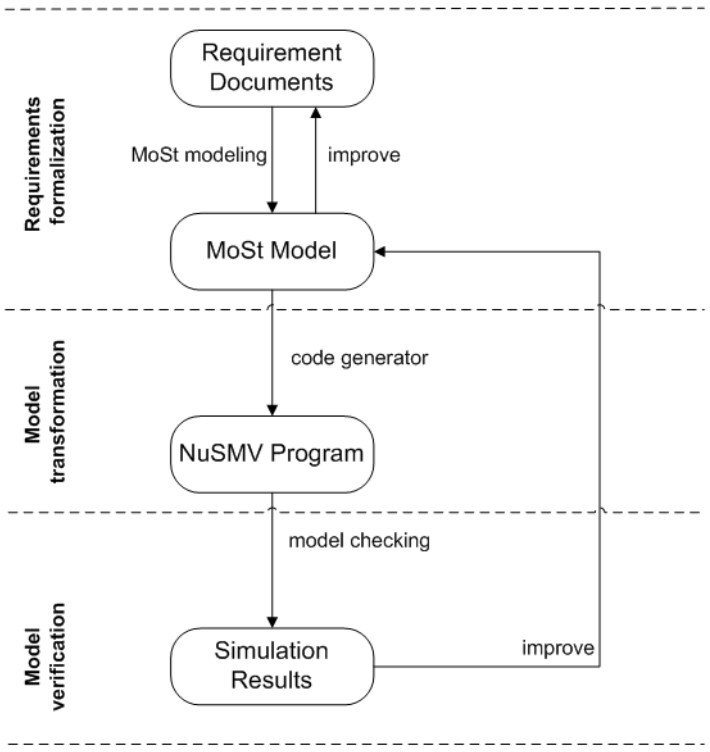

Figure 2: The framework for modelling and verifying natural language requirements based on states and modes

As mentioned in Section 1, the requirements of designing a car are analyzed as a case study. They are inspired from *Dusan Rodina* [2]. Since few engineers analyse requirements from our perspective, it is hard to find the existing requirements without reformulating them. On the other hand, states and modes are sometimes intertwined in requirements, which makes it difficult to use the existing requirements. Thus, we finally choose to design our proper requirements that are based on the correct logic. These requirements will apply to the following requirements modeling and verification.

2. https://www.softwareideas.net/a/1539/Car-States--UML-State-Machine-Diagram- (accessed in May 2021)

3.3. MoSt Meta-model

The MoSt meta-model highlights the properties of the MoSt Modelling Language (MoStML). As shown in Fig. 3, MoStML is capable of describing NLRs and formal requirements. Even though formal requirements are also modelled by the natural language, this natural language should conform to certain rules. The NLRs enable us to capture the important information from the requirement documents as much as possible, in order to serve traceability in case of troubleshooting. Note that extracting important information from free natural languages is out of our scope.

MoStML focuses on describe functional requirements and non-functional requirement. MoStML-based formal requirements consist of concepts *Mode*, *State*, *Constraint* and *EnvironmentRequirement*. The concepts of *Mode*, *State*, and *Constraint* describe functional requirements. Non-functional requirements can only be expressed via *Constraint* concept. More specifically, concepts *PropertyConstraint* depicts functional and non-function requirements to be checked. On the other hand, *EnvironmentRequirement* concept initializes values and ranges of system attributes.

Concept *Condition* is one of the most important concepts in this meta-model, which includes *ConjunctionCondition* and *DisjunctionCondition*. These two types of conditions enrich the expressive power of MoStML. For example, the precondition of one state transition can be any of conjunction, disjunction, and conjunction and disjunction of conditions *ModeCondition*, *StateCondition*, *AttributeCondition*, *SignalCondition* and *ArithmaticCondition*. However, concepts of *ModeTransition* and *EnvironmentRequirement* are exceptional, which are directly associated with specific conditions. The reasons have been mentioned in section 3.1.

3.4. MoSt Grammar

A grammar is a set of rules that describe the form of the elements that are valid according to the language syntax (Bettini 2016). The MoSt grammar illustrates the rules describing how to write different types of requirements. The

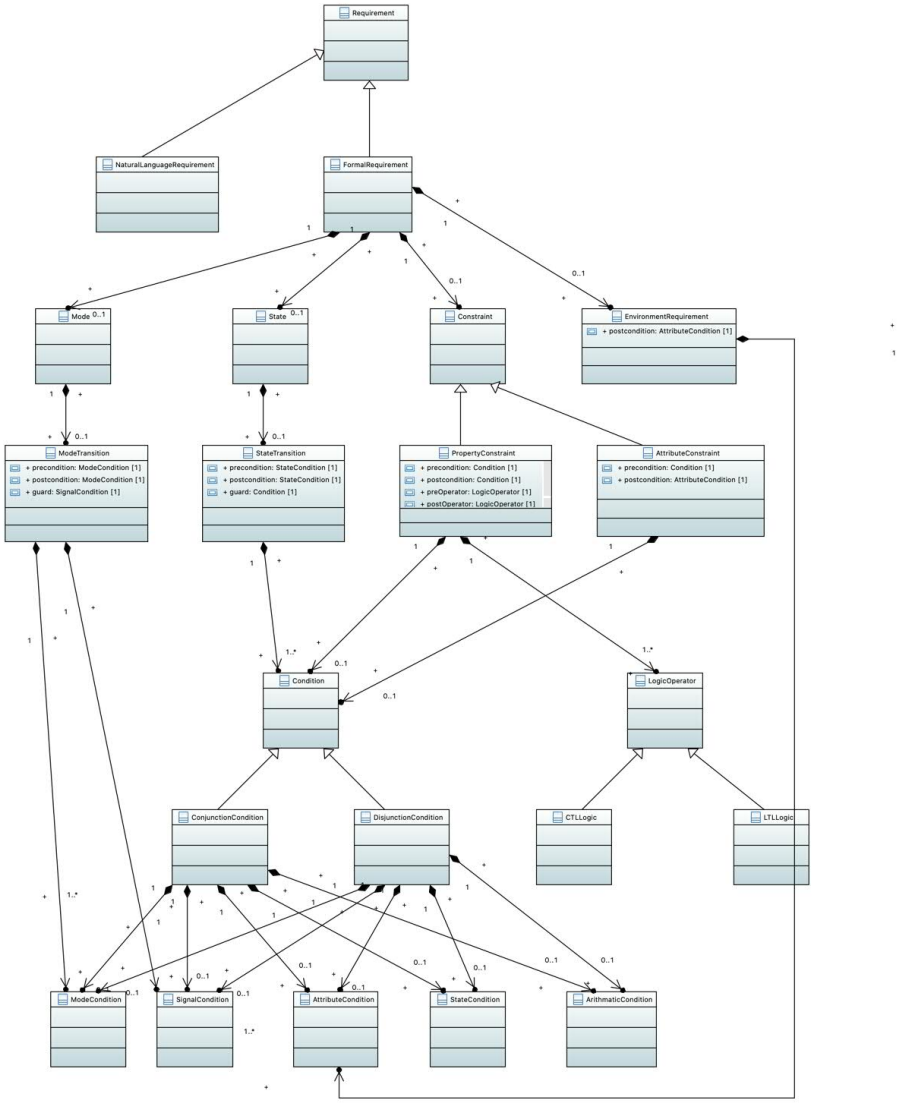

Figure 3: The Meta-model of MoSt Modelling Language

complete grammar is shown in the annex (Figs. 14 - 18). Every rule contains a name, a colon, a syntactic form, and a semicolon. The first rule of the grammar defines where the parser starts and the type of the root element of the MoSt model is *MoSt*. The shape of *MoSt* elements is expressed in its own rule :

MoSt : models+=(Requirement | NLRequirement) ;*

A collection of *Requirement* or *NLRequirement* elements are stored in feature *models* of a *MoSt* object. Formal requirements are stored in *Requirement* objects. NLRs are stored in *NLRequirement* objects. Note that += and * operators mean it is a collection and the number of elements is arbitrary respectively.

3.4.1. Natural Language Requirements

The natural language requirement rule is illustrated as follows :

NLRequirement : nlReqID=ReqID ID (ID) '.' ;*

ReqID :'[' reqID+=INT ('.' reqID+=INT) ']' ;*

It implies that NLRs begin with the *ReqID* (the identity of requirements) like "[1.2.3...N]". So this naming rule of *ReqID* signifies there is no limit to the number of NLRs. This rule applies to all the other requirements as well. As for *ID*, there is no rule defining it because that is one of the rules from the *Terminals* (mentioned in Xtext). It allows us to write any words as we want. As a result, the rule of NLRs is just to write natural language sentences with identities.

3.4.2. Formal Requirements

Formal requirements include *Environment*, *MODE*, *STATE*, *ATTRIBUTE*, and *PROPERTY*. The rule of formal requirements is represented as follows :

Requirement : ENVIRONMENT | MODE | STATE | ATTRIBUTE | PROPERTY ;

1). Environment Requirements

Environment requirements are dedicated to describing initial statuses of variables, including the initialized values and the ranges of variables. That's why the rules of environment requirements involve *ATTRIBUTEVALUE*, *UNIT* and *RANGE*, which are shown as follows :

ENVIRONMENT :

envirReqID=ReqID ID envirVariable=ID (ID) (('initialised' 'to' envirAttributeValue=ATTRIBUTEVALUE envirUnit=UNIT | range=RANGE)) (ID)* '.' ;*

In terms of *ATTRIBUTEVALUE*, three common data types are employed, including *INT*, *String* and *Boolean*. As for *UNIT*, the units of weight, time, speed,

and accelerate speed are applied. The rule of *RANGE* indicates the lower and upper bounds for the variables. This rule concerns the comparison operators, which limit the bounds and include *GREATER*, *GREATEREQUAL*, *LESS*, and *LESSEQUAL* rules. The details of the rules *ATTRIBUTEVALUE*, *UNIT*, *RANGE*, and *COMPARISONOPERATOR* are shown in Figs 16 and 18.

Table 2 lists the values of the attributes for two environment requirements (Reqs 1 and 2). Req 1 initializes variable ”accSpeed”. Req 2 provides the scope for this variable. Note that variables must be named as compound nouns (like ”accSpeed” instead of ”accelerate speed”) if they involve several nouns. This rule will apply to other attributes of the MoSt grammar.

Table 2: Instances of Two Environment Requirements

Req 1	envirReqID = ”2.2.1”	envirVariable = ”accSpeed”
	envirUnit = ”m/s2”	envirAttributeValue = ”0”
Req 2	envirReqID = ”2.2.2”	envirVariable = ”accSpeed”
	compOperator1 = ”greater or equal to”	bound 1 = ”0”
	compOperator2 = ”less or equal to”	bound2 = ”10”
	envirUnit = ”m/s2”	

The corresponding MoSt code can be written as follows :

Req 1 : [2.2.1] The accSpeed should be initialised to 0 m/s2.

Req 2 : [2.2.2] The accSpeed should be greater or equal to 0 less or equal to 10 m/s2.

2). Mode Requirements

Mode requirements explain mode transitions, which are expressed by the sentence pattern ”when..., then...”. This sentence pattern applies to state, property and constraint requirements. Mode transitions are associated with mode and signal conditions. Mode conditions indicate which mode the system is in. Signal conditions imply the condition for triggering mode transitions. The rules of mode requirements are listed as follows :

MODE :

modeReqID=ReqID 'when' preModeConditions+=MODECONDITION relation= RELATION preModeConditions+=SIGNALCONDITION ',' 'then' postModeCondition = MODECONDITION'.';

The details of *MODECONDITION*, *SIGNALCONDITION* and *RELATION* are shown in Figs. 15 and 18. Note that since Figs. 15 - 18 list the details of the basic rules of each requirement rule, we avoid repeating them in the following. Table 3 lists the values of the attributes for mode requirement Req 3. This requirement describes the transition between modes Economic and Sportive.

Table 3: Instance of Mode Requirement Req 3

Req 3	modeReqID = ”6.2”	preModeCondition[1] = ”mode = economic”
	relation = ”and”	preModeCondition[2] = ”signal = Ac”
		postModeCondition = ”mode = sportive”

The corresponding MoSt code can be written as follows :

Req 3 : [6.2] when the car is in mode economic and it receives Ac signal, then it is in mode sportive.

3). State Requirements

State requirements describe system functional requirements via state transitions. Three conditions are able to trigger state transitions, including attribute, mode, signal conditions. The rules of state requirements are illustrated as follows :

STATE :

stateReqID=ReqID ’when’ preStateConditions+=STATECONDITON (relations+=RELATION preStateConditions += (ATTRIBUTECONDITION | MODECONDITION | SIGNALCONDITION)) ’,’ ’then’ postStateCondition = STATECONDITON ’.’;*

Table 4 gives the attribute values of the state requirement Req 4. It depicts the transition between states Accelerate and Autonomy.

Table 4: Instance of State Requirement Req 4

Req 4	stateReqID = ”1.4”	preStateCondition[1] = ”state = accelerate”
	relations[1] = ”and”	preStateCondition[2] = ”signal = Auto”
	relations[2] = ”and”	preStateCondition[3] = ”accSpeed = 10 m/s2”
		postStateCondition = ”state = autonomy”

The corresponding MoSt code can be written as follows :

Req 4 : [1.4] when the car is in state accelerate and it receives Auto signal and its accSpeed is equal to 10 m/s2, then it will be in state autonomy.

4). Attribute Requirements

Attribute requirements aim at determining the values of attributes under different conditions. The conditions can be any of the combination of state, mode, signal, and attribute conditions. The value can be an arithmetic equation as well. The rules of attribute requirements are depicted as follows :

ATTRIBUTE :

attributeReqID=ReqID 'when' preAttributeConditions += (STATECONDITON | ATTRIBUTECONTION | MODECONDITION | SIGNALCONDITION) (relations += RELATION preAttributeConditions += (STATECONDITON | ATTRIBUTECONTION | MODECONDITION | SIGNALCONDITION))* ',' 'then' postAttributeCondition = (ATTRIBUTECONTION | ARITHMETICCONDITION) '.' ;*

Table 5 shows the values of attributes for attribute requirement Req 5. This requirement explains how the mode and the speed of a car influence the function of displaying speed.

Table 5: Instance of Attribute Requirement Req 5

Req 5	attributeReqID = "1.4"	preAttributeCondition[1] = "mode = economic"
	relations[1] = "and"	preAttributeCondition[2] = "speed > 80 km/h"
		postAttributeCondition = "displaySpeed = TRUE"

The corresponding MoSt code can be written as follows :

Req 5 : [5.2] when the car is in mode economic and its speed is greater than 80 km/h, then its displaySpeed is equal to TRUE.

5). Property Requirements

Property requirements support the description of functional requirements and non-functional requirements. They will be used as properties that need to be checked. They are often involved with temporal issues. Classic temporal logics are considered in our language. The expressive power of the language is significantly increased by introducing CTL (Computational Temporal Logic) and LTL (Linear Temporal Logic). The rules of property requirements are expressed as follows :

PROPERTY :

propertyReqID=ReqID 'when' preOperator= (CTLOperator | LTLOperator) prePro-

pertyConditions += (STATECONDITON | ATTRIBUTECONTION | MODECONDITION) (preRelations+=RELATION prePropertyConditions += (STATECONDITON | ATTRIBUTECONTION | MODECONDITION))* ',' 'then' postOperator = (CTLOperator | LTLOperator) postPropertyConditions += (STATECONDITON | ATTRIBUTECONTION | MODECONDITION)* (postRelations+=RELATION postPropertyConditions += (STATECONDITON | ATTRIBUTECONTION | MODECONDITION))* '.';*

Table 6 lists all the values of the property requirement attributes. This requirement provides a CTL specification to check the function of the car.

Table 6: Instance of Property Requirement Req 6

Req 6	attributeReqID = "7.1"	preOperater = "all globally"
	prePropertyConditions[1] = "state = autonomy"	preRelations[1] = "and"
	prePropertyConditions[2] = "mode = economic"	postOperator = "all next"
	postPropertyConditions[1] = "state != accelerate"	

The corresponding MoSt code can be written as follows :

Req 6 : [7.1] when all globally the car is state autonomy and it is in mode economic, then all next it is not in state accelerate.

3.5. MoSt Semantics

The meta-model of the MoSt modeling language is expressed by the UML class diagram. However, the semantics of models written in UML is not precisely defined(Szlenk 2006). This issue may give rise to concerns about the quality of the model. The OCL (Object Constraint Language) is a declarative language describing rules applying to UML models [3]. Thus, in this paper, the OCL will be used to describe the static semantics of the MoSt modeling language. OCL statements consist of four parts :

— a context that defines the limited situation in which the statement is valid ;

— a property that represents some characteristics of the context ;

— an operation that manipulates or qualifies a property ;

3. https://en.wikipedia.org/wiki/Object_Constraint_Language/ (accessed in April 2021)

— keywords that are used to specify conditional expressions.

The MoSt formal semantics in OCL is shown in Fig. 4. The OCL rules are based on the meta-model of the MoSt modeling language (Fig. 3). It means all the elements constituting the OCL rules are derived from the meta-model except for the names of invariants (”inv” in Fig .4). Five rules are mentioned to express the MoSt formal semantics, including environment, mode, state, attribute, property requirement rules.

1). *EnvironmentRequirement* OCL rule

The rule for environment requirements (lines 3-6) defines if one requirement is an environment requirement, it will contain a post-condition of type *AttributeCondition*. Note that the type of the post-condition is not *AttributeCondition* from the grammar point of view. We apply *AttributeCondition* to the post-condition, because the precise meaning of the post-condition is equal to *AttributeCondition*.

2). *ModeTransition* OCL rule

The semantics of mode requirements is represented by *ModeTransition* rule (lines 7-12). This rule indicates the types of precondition and the post-condition of mode requirements are both *ModeCondition*. Only condition *SignalCondition* is permitted as the guard of mode transitions.

3). *StateTransition* OCL rule

The rule for state requirements is shown in lines 13-20. Similar to mode requirements, the types of the precondition and the post-condition of state requirements are both *StateCondition*. The guard of the requirements is the combination of the conjunction and disjunction of all the conditions except for arithmetic and state conditions.

4). *AttributeConstraint* OCL rule

As shown in lines 21-27, no guard constraint is imposed on attribute requirements. The precondition can be any of the conjunction and disjunction of all the conditions except for the arithmetic condition. The post-condition is either the attribute condition or arithmetic condition.

5). *PropertyConstraint* OCL rule

The rule for property requirements is listed in lines 28-38. The precondition and the post-condition share the same constraints, that is, the conjunction and disjunction of all the conditions except for signal and arithmetic conditions. Property requirements contain logic operators (”preOperator” and ”postOperator”) for preconditions and post-conditions. Both operators are any of the sub-types of *LogicOperator*, that is, CTL and LTL logic operators.

```
1.  import 'RequirementModel.uml'
2.  package RequirementModel

3.  context EnvironmentRequirement
4.  inv environment_requirement_specification:
5.  self.oclIsTypeOf(EnvironmentRequirement) implies
6.  self.postcondition.oclIsTypeOf(AttributeCondition)

7.  context ModeTransition
8.  inv mode_transition_specification:
9.  self.oclIsTypeOf(Mode) implies
10. self.precondition.oclIsTypeOf(ModeCondition) and
11. self.guard.oclIsTypeOf(SignalCondition) and
12. self.postcondition.oclIsTypeOf(ModeCondition)

13. context StateTransition
14. inv state_transition_specification:
15. self.oclIsTypeOf(State) implies
16. self.precondition.oclIsTypeOf(StateCondition)
17. self.guard.oclIsKindOf(Condition) and not
18. self.guard.oclIsTypeOf(ArithmaticCondition) and not
19. self.guard.oclIsTypeOf(StateCondition) and
20. self.postcondition.oclIsTypeOf(StateCondition)

21. context AttributeConstraint
22. inv attribute_constraint_specification:
23. self.oclIsTypeOf(AttributeConstraint) implies
24. self.precondition.oclIsKindOf(Condition) and not
25. self.precondition.oclIsTypeOf(ArithmaticCondition) and
26. self.postcondition.oclIsTypeOf(AttributeCondition) or
27. self.postcondition.oclIsTypeOf(ArithmaticCondition)

28. context PropertyConstraint
29. inv property_constraint_specification:
30. self.oclIsTypeOf(PropertyConstraint) implies
31. self.precondition.oclIsKindOf(Condition) and not
32. self.precondition.oclIsTypeOf(SignalCondition) and not
33. self.precondition.oclIsTypeOf(ArithmaticCondition) and
34. self.postcondition.oclIsKindOf(Condition) and not
35. self.postcondition.oclIsTypeOf(SignalCondition) and not
36. self.postcondition.oclIsTypeOf(ArithmaticCondition) and
37. self.preOperator.oclIsKindOf(LogicOperator) and
38. self.postOperator.oclIsKindOf(LogicOperator)
39. endpackage
```

Figure 4: MoStML Formal Semantics

4. Model Transformation

The NuSMV model checker enables us to write the NuSMV code in different ways. It is essential to identify one of the most appropriate forms of NuSMV models, which corresponds to the MoSt model. Therefore, the mapping between modules of the MoSt model and the NuSMV model is provided, as shown in Fig. 5. The implementation of the process for the automatic model transformation is based on this mapping. The process is realized in Algos. 1 - 5, which will be discussed in the following.

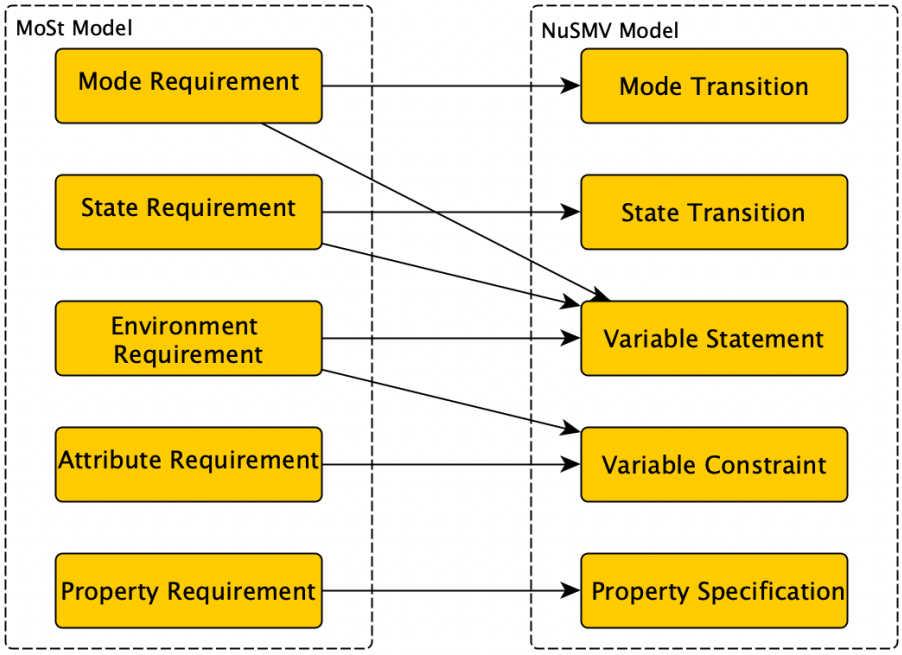

Figure 5: Mapping Between Modules of the MoSt Model and the NuSMV Model

As mentioned in section 3, the MoSt modeling language consists of mode, state, environment, property, attribute modules. Mode modules are able to actively and passively intervene in the behaviors of the system. The active intervention implies modes can change the values of variables that influence the state transitions. The passive intervention means the mode itself can be re-

garded as the trigger. At the same time, the mode owns its mode transitions impacted by signals. Therefore, the type of the mode variable can be defined as *enumeration*. The mode requirement transformation is implemented in Algo. 1.

Algorithm 1: Mode Requirement Transformation

```
   input  : root
   output: modeTransitions
 1 ArrayList<String> modeTransitions = new ArrayList<String>();
 2 String temp="";
 3 int indexMode = 0;
 4 for modeReq : root.model.filter(MODE) do
 5 |   indexMode = 0;
 6 |   temp ="";
 7 |   for preModeCondition : modeReq.preModeConditions do
 8 |   |    temp+=preModeCondition.condition;
 9 |   |   if indexMode <= modeReq.relation.size - 1 then
10 |   |   |    temp+=modeReq.relation.get(indexMode++).relation;
11 |   |   end
12 |   end
   |   // Post-mode conditions look like "mode = A", the value of the mode is
   |      just required, conforming to NuSMV code rules.
13 |   temp+=postModeCondition.condition.split("=").get(1);
14 |   modeTransitions.add(temp);
15 end
```

The NuSMV model naturally approves state modules. The input language of NuSMV supports the description of Finite State Machines (FSMs) which range from completely synchronous to completely asynchronous, and from the detailed to the abstract(Cavada et al. 2019). Thus, state modules can be transformed into the corresponding NuSMV code. The state requirement transformation is performed in Algo. 2.

In terms of other elements of the MoSt model, Environment and attribute modules are associated with the variable statement and the variable constraint. So, they can also be transformed into the NuSMV model. Since CTL and LTL specifications can be checked in the NuSMV model checker, the transformation of the specification module is feasible as well. The processes for the model transformation for specification, environment, and attribute modules are achieved in Algos. 3 - 5 respectively.

In order to make the process clearer, the car requirements have been forma-

Algorithm 2: State Requirement Transformation

```
   input  : root
   output: stateTransitions
   // The variable root represents the root of the MoSt model.
1  ArrayList<String> stateTransitions = new ArrayList<String>();
2  String temp="";
3  int indexState = 0;
4  for stateReq : root.model.filter(STATE) do
5  |   indexState = 0;
6  |   temp ="";
7  |   for preStateCondition : stateReq.preStateConditions do
8  |   |    temp+=preStateCondition.condition;
9  |   |   if indexState <= stateReq.relation.size - 1 then
10 |   |   |    temp+=stateReq.relation.get(indexState++).relation;
11 |   |   end
12 |   end
   |   // Post-state conditions look like "state = A", the value of the state
   |      is just required, conforming to NuSMV code rules.
13 |   temp+=postStateCondition.condition.split("=").get(1);
14 |   stateTransitions.add(temp);
15 end
```

lised in Fig. 6. Five kinds of requirements have been elicited to illustrate the transformation process. Note that the complete NuSMV code is shown in the annex.

Algorithm 3: Environment Requirement Transformation

```
   input  : root
   output: variableConstraints, variables
1  HashMap<String,String> variableConstraints = new HashMap<String,String>();
2  HashMap<String,String> variables = new HashMap<String,String>();
3  String key="";
4  String pre="";
5  double max,min;
6  max=min=0;
7  for environmentReq : root.model.filter(ENVIRONMENT) do
8      key = environmentReq.envirVariable;
       // initializing variables
9      if environmentReq.range == null then
10         pre = variableConstraints.get(key);
11         if pre == null then
12             pre="";
13         end
           // The initial value of variables is stored in the left part of @
              of pre.
14         pre = environmentReq.envirAttributeValue.attributeValue + "@" + pre;
15         variableConstraints.put(key,pre);
16     end
       // setting the scope for variables
17     else
18         if environmentReq.range.bound1.attributeValue <=
             environmentReq.range.bound2.attributeValue then
19             min=environmentReq.range.bound1.attributeValue;
20             max=environmentReq.range.bound2.attributeValue;
21         end
22         else
23             min=environmentReq.range.bound2.attributeValue;
24             max=environmentReq.range.bound1.attributeValue;
25         end
26         variables.put(key,min+ ".."+max);
27     end
28 end
```

Algorithm 4: Attribute Requirement Transformation

```
   input  : root
   output: variableConstraints
 1 HashMap<String,String> variableConstraints = new HashMap<String,String>();
 2 String temp="";
 3 String pre, key;
 4 int indexAttribute = 0;
 5 for attributeReq : root.model.filter(ATTRIBUTE) do
 6 |   indexAttribute = 0;
 7 |   temp ="";
 8 |   for preAttributeCondition : attributeReq.preAttributeConditions do
 9 |   |    temp+=preAttributeCondition.condition;
10 |   |   if indexAttribute <= modeReq.relation.size - 1 then
11 |   |   |    temp+=attributeReq.relation.get(indexAttribute++).relation;
12 |   |   end
13 |   end
14 |   key=attributeReq.postAttributeCondition.condition.split("=").get(0);
15 |   temp+=" :"+attributeReq.postAttributeCondition.condition.split("=".get(1))+";";
16 |   pre=attributeConstraints.get(key); if pre!=null then
17 |   |   temp+=pre;
18 |   end
19 |   if pre!=temp then
20 |   |   attributeConstraints.put(key,temp);
21 |   end
22 end
```

Algorithm 5: Property Requirement Transformation

```
   input  : root
   output: propertySpecifications
 1 ArraryList<String> propertySpecifications = new ArraryList<String>();
 2 String temp="";
 3 int indexPreProperty, indexPostProperty;
 4 for propertyReq : root.model.filter(PROPERTY) do
 5 |   indexPreProperty = 0;
 6 |   indexPostProperty = 0;
 7 |   temp = propertyReq.preOperator.logicOperator;
 8 |   for prePropertyCondition : propertyReq.prePropertyConditions do
 9 |   |   temp+=prePropertyCondition.condition;
10 |   |   if indexPreProperty <= propertyReq.preRelation.size - 1 then
11 |   |   |    temp+=propertyReq.preRelations.get(indexPreProperty++).relation;
12 |   |   end
13 |   end
14 |   temp += propertyReq.postOperator.logicOperator;
15 |   for postPropertyCondition : propertyReq.postPropertyConditions do
16 |   |   temp+=postPropertyCondition.condition;
17 |   |   if indexPostProperty <= propertyReq.postRelation.size - 1 then
18 |   |   |    temp+=propertyReq.postRelations.get(indexPostProperty++).relation;
19 |   |   end
20 |   end
21 |   propertySpecifications.add(temp);
22 end
```

FIGURE 6: An Example of the Model Transformation

5. Requirements Verification

5.1. MoSt Modeling Tool

The MoSt modeling tool is implemented by the Xtext framework in Eclipse. The screenshot of the tool is shown in Fig. 7. After correctly writing the MoSt code, the corresponding NuSMV model will be generated automatically. Multiple checking can be performed in this tool. Requirements static checking is accomplished in the *MoStMLValidator* by using the Xtend language. For example, names and requirement consistency can be checked. Requirements in the MoSt model can also be traced in the NuSMV model, thanks to the IDs of requirements. On the other hand, the completeness of requirements can be analysed with the help of the model checker NuSMV. The details of the requirement checking will be discussed in the next sections. This project is available on

GitHub [4].

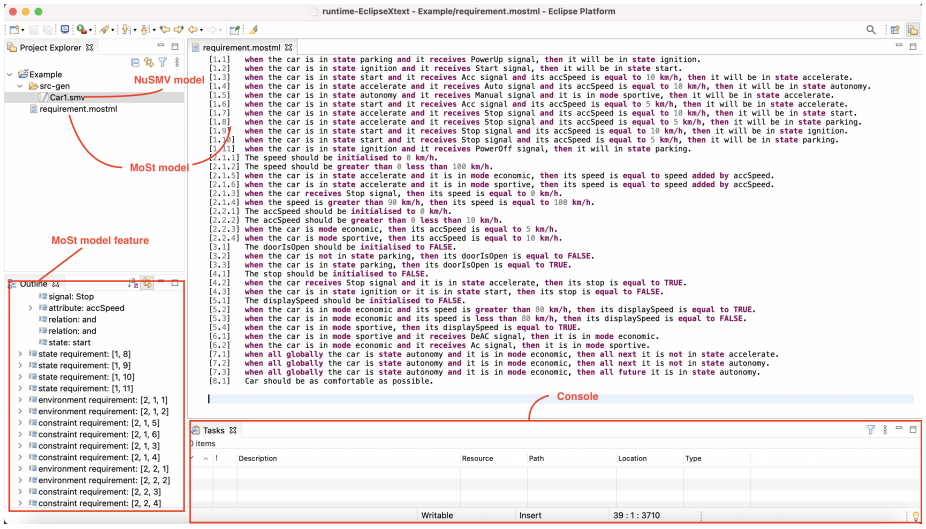

FIGURE 7: Screenshot of the MoSt Modeling Tool

5.2. *Requirements Static Checking*

Requirements static checking aims at verifying the names of the MoSt model elements and ensuring the requirement consistency from the user-defined standards. Requirement static checks are triggered while writing the MoSt code. If the user-defined standards are violated, errors will be prompted in the MoSt modeling editor.

The naming checks (NC)in the MoSt model concerning states, modes and signals are : the names of states and modes should be starting with a lower case (NC1) ; the name of signals should be beginning with an upper case (NC2).

The requirement consistency checks (CC) include :

CC1 : Non-integer variables should only be initialised once ;

CC2 : The variables mentioned in attribute requirements should be initialized ;

CC3 : The variable of *INT* should be given the scope ;

4. https://github.com/liuyinling/MoSt-Modeling-Tool.git

CC4 : The repetition of requirement IDs is not allowed ;

CC5 : The repetition of requirements is not allowed ;

CC6 : Different post-conditions of one type of requirements cannot have the same preconditions.

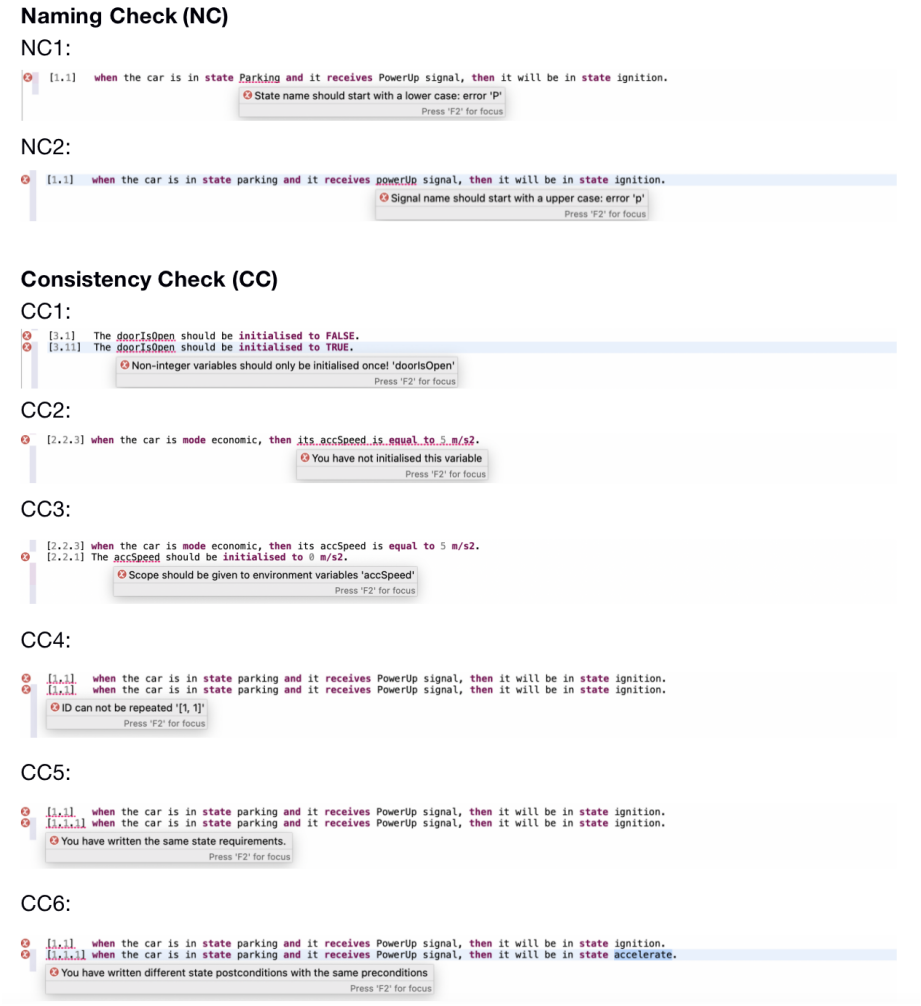

Figure 8: Results of Requirement Static Checks

Fig. 8 shows the results of requirement static checks. These checks help us write a proper MoSt model. However, they cannot guarantee that the generated

NuSMV model is correctly executable. Therefore, requirement dynamic checks should be carried out.

5.3. Requirements Dynamic Checking

Requirement dynamic checks rely on the model checker NuSMV. The simulation results will just be shown in the console of the model checker. Thus, it is important to trace the NuSMV code to the requirements in the MoSt model. After that, if errors are prompted when running the simulation, the NuSMV model checker will localise the errors by mentioning the line number of the code. With the proposed line number, we are able to trace the potential errors in the requirements. These errors are involved with the correctness of the NuSMV model and the underlying problems in the requirements. The successful running of the NuSMV simulation verifies its correctness. Underlying problems may be found when counter-examples are proposed.

5.3.1. Traceability Analysis

Since every requirement starts with a unique ID in the MoSt model, these IDs can be served as the medium to ensure traceability between requirements and the NuSMV code. To avoid the impact of the IDs in the code, the corresponding lines of the code are commented by the IDs. A screenshot of the NuSMV code with IDs is shown in Fig. 9.

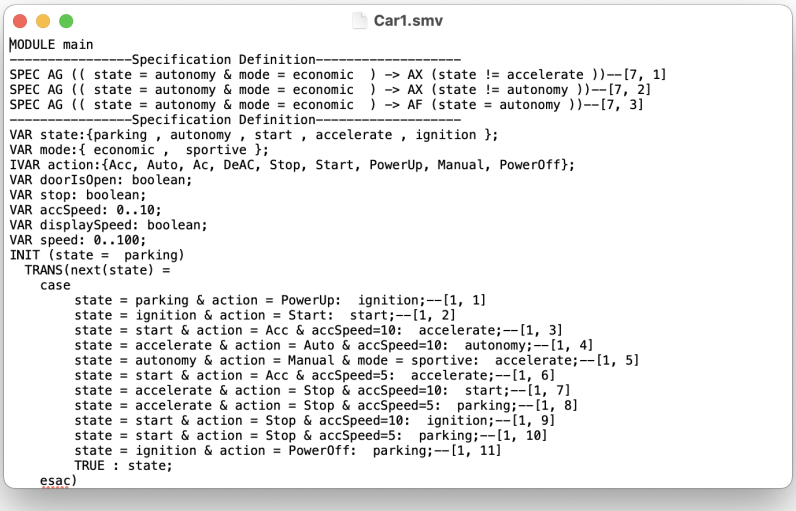

Figure 9: Screenshot of the NuSMV Code with IDs

In order to illustrate the feasibility of the traceability analysis, we manually inject four errors in the MoSt model, including an undefined identifier, division by zero, illegal operand type, and illegal type of ”case” list element. Fig. 10 provides the results of the simulation of model checking. Four trace links between the NuSMV code and the corresponding requirement have been identified. The process for the traceability analysis includes four steps : *Error Analysis*, *Localising NuSMV Code*, *Localising Requirement* and *Correcting Requirements*. We will firstly use Error 1 (undefined identifier) as an example to illustrate a trace link as follows :

Error analysis The ”autonmy” is undefined in line 3 of ”Car1.smv” file.

Localising NuSMV code The model checker shows line 3 of the NuSMV code corresponds to requirement [7.1];

Localising requirements Search requirement [7.1] in the MoSt Model;

Correct requirements Replace ”autonmy” with ”autonomy”.

This link not only precisely localises the error hidden in the MoSt model but provides the cause of the error. It helps improve the quality of requirements quickly and efficiently. The other three links are drawn in Fig. 10 and the details of four links are also shown in Table 7. Thus, we have finally identified all the errors injected. So far, we have realised two types of error checking. The one can be found with the help of the rules defined in *Validator* of the tool when writing the MoSt code in the editor. The other can be traced with requirement IDs when launching the simulation of the NuSMV code. However, the logic of the requirements is not able to be checked. In other words, the successful compiling of the NuSMV code does not necessarily mean the requirements are complete. As a result, the completeness analysis will be introduced in the next sub-section.

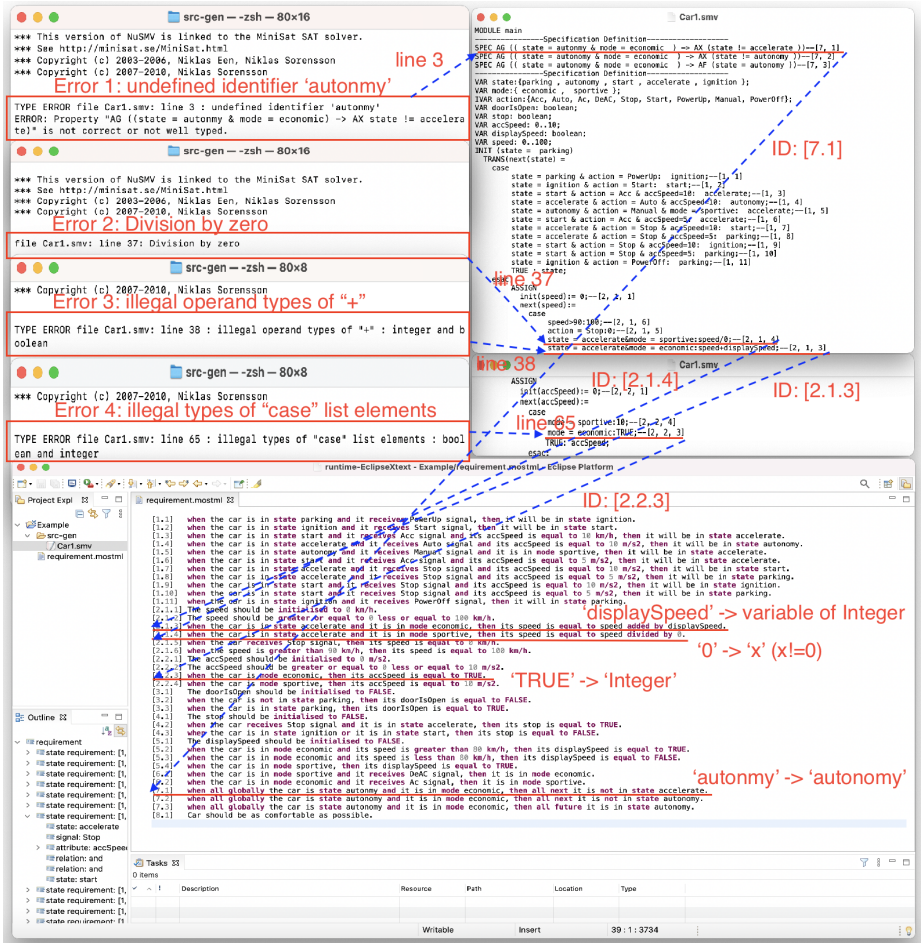

FIGURE 10: The Checking Results of the NuSMV Code Generated from the MoSt Model with Four Injected Errors

TABLE 7: The Details of Four Trace Links

Error	Line	ID	Correction
undefined identifier ’autonmy’	3	[7.1]	’autonmy’ – > ’autonomy’
division by zero	37	[2.1.4]	’0’ – > ’x’ (x! =0)
illegal operand types of ”+”	38	[2.1.3]	’displaySpeed’ – > variable of Integer
illegal types of ”case” list elements	65	[2.2.3]	’TRUE’ – > ’Integer’

5.3.2. *Completeness Analysis*

Completeness analysis offers the possibility to check the logic of the requirements. This step further improves the correctness of the requirements. This

analysis is performed by proposing additional CTL specifications. The specification is conformed to the form $\phi = EF\ (state = X)$ to check the reachability of each state. To facilitate the process of completeness analysis, additional specifications for the reachability of each state are generated automatically from the previous model transformation. Fig. 11 shows the automatically generated CTL specifications.

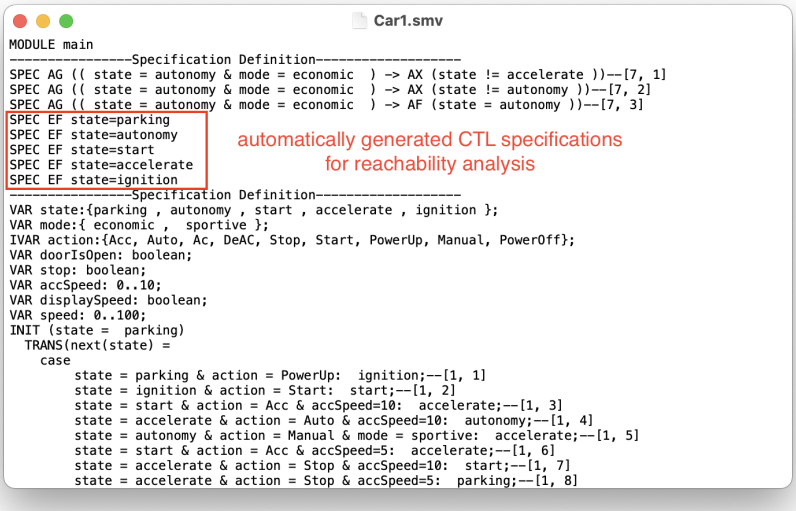

Figure 11: Automated Specification Generation for Completeness Analysis

To demonstrate the feasibility of the completeness analysis, we voluntarily delete requirement [1.4] ("[1.4] when the car is in state accelerate and it receives Auto signal and its accSpeed is equal to 10 m/s2, then it will be in state autonomy."). This requirement describes the state transition between state Accelerate and state Autonomy. Then, we add the CTL specification $SPEC\ EF\ state = autonomy$ to the NuSMV code. After launching the simulation, the counterexample is shown in Fig. 12. This CTL specification can not pass, which means the car can never reach state Autonomy. The result of the completeness result implies the transitions related to state Autonomy should be checked. There-

fore, the completeness analysis can check the completeness of the state transitions.

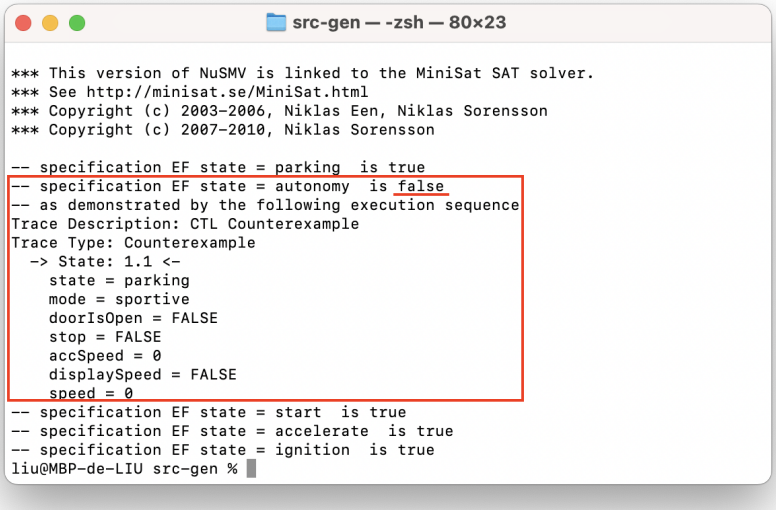

Figure 12: An Example of Completeness Analysis

5.3.3. Validation analysis

Validation analysis checks the satisfaction of property requirements. Property requirements are transformed into the CLT/LTL specifications in the NuSMV model. Thus, validation analysis checks the satisfaction of CTL/LTL specifications regarding the NuSMV model. This analysis enables customers to gain a global understanding of how the system will work. Requirements engineers should conduct validation analysis from the requirements level. Because the requirements that will never be satisfied should be modified or deleted as early as possible. This can avoid conflicts between developers and customers when the system is delivered.

Let's take requirement [7.2] as an example. This requirement said, "when all globally the car is in state autonomy and it is in mode economic, then all next it is not in state autonomy". This requirement focuses on the relationship

between autonomy state and economic mode. The code generator will translate this requirement into a CTL specification $SPEC\ AG((state = autonomy \& mode = economic) -> AX(state != autonomy))$. The verification result of this specification is false, shown in Fig. 13. The counter-example implies when the car is autonomy state and we change the working mode of the car into economic mode, the state of the car will not be changed.

This experiment shows that requirement [7.2] will not be satisfied when the car is delivered. Based on this analysis result, requirements engineers can negotiate with customers to either modify this requirement or delete it. On the other hand, even though the customers do not necessarily understand the counterexample in this figure, to some extent, they do understand this specification can not pass in the simulation. This serves as strong evidence why requirements engineers ask them to at least modify it.

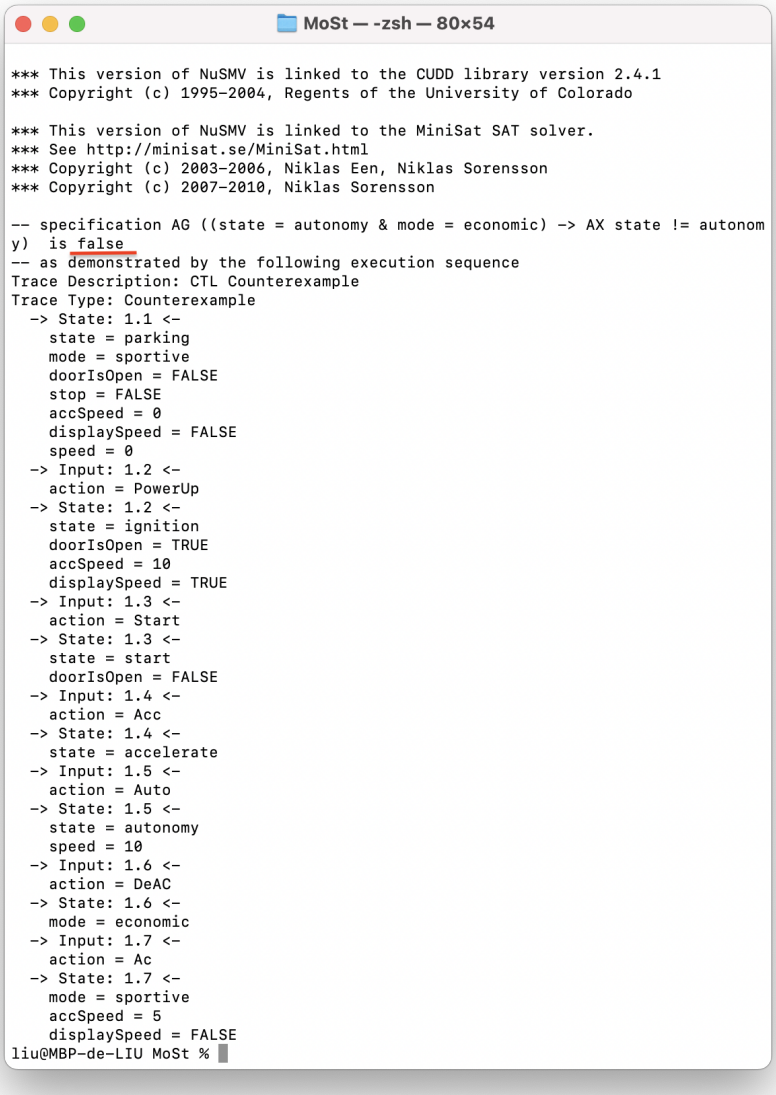

Figure 13: An Example of Validation Analysis

5.4. Discussion

In this paper, a ”new” and ”old” viewpoint (states and modes) has been introduced to requirements modeling and verification. The viewpoint is new be-

cause no one really conducts requirements analysis from the viewpoint. Since the terms of states and modes have been using in the industry, this point of view is also old. To make our approach clear, we propose an innovative approach to emphasize states and modes in requirements. This approach is divided into three steps.

The first step is to design a DSL. Our DSL is a controlled natural language with its proper grammar and semantics, which not only facilitates requirements expressing for both users and developers but enables us to capture the information of states and modes. In this paper, we suppose the requirements described by our DSL should be as precise as possible. In other words, the information of states and modes is supposed to be recognized in advance in requirements. This hypothesis is reasonable because they indeed exist. On the other hand, the requirements mentioned in many papers (Leveson et al. 1994; Ahmad, Belloir, and Bruel 2015; Moitra et al. 2019; Giannakopoulou et al. 2021) provide accurate information of the systems. For example, the range limits of every variable are clearly given.

Secondly, the algorithms have been designed to realize the automatic model transformation. We choose to use the NuSMV model to conduct model checking. In order to connect states and modes, they are both defined as variables. Thus, in the NuSMV model, mode transitions will cause the value of the corresponding variables to change. This change can provoke state transitions. The relationship between states and modes is strictly derived from that mentioned in section 3.1.

The last step of our approach is involved with requirements verification. We implemented a validator to statically check requirements. This validator is user-defined static verification ”tool”, which helps not only check the naming rules for variables but check the consistency. This ”tool” can be highly effective in detecting errors in requirements specifications. For example, it is capable of checking whether the same variable is initialized several times. Sometimes, it may help requirements engineers understand requirements better. Let’s take consistency check CC6 in Fig. 8 as an example, this ”tool” can pro-

pose errors when requirements [1.1] and [1.1.1] with different post-conditions own the same preconditions. This error shows non-determinism in state transitions. Different solutions can be raised to solve this issue. For example, states are mistakenly written. Or, some underlying information between state transitions is ignored. These ideas can liberate people from the tedious and error-prone task of checking specifications for consistency.

Our approach is helpful for conducting dynamic requirements verification thanks to the NuSMV model checker. Since the MoSt-expressed requirements are able to be transformed into the NuSMV model, we will get full use of this model checker to help us dynamically check requirements. However, the issue is how we can trace back to requirements from counter-examples proposed by the model checker. A certain level of traceability has been preserved thanks to the definition of requirement ID. Each line of code of the transformed NuSMV model is associated with its specific requirement ID in the form of comments, except for the auto-generated test code. So, the code of the NuSMV model is not influenced when executed. Note that counter-examples concerning specifications are not necessarily easily understandable for users without formal verification experience but at least they will know these specifications are not ensured.

Finally, our approach facilitates the validation process. The validation analysis mentioned in section 5.3.3 has avoided conflicts between customers and developers from the requirements level, to some extend. Conflicts in requirements should be found as early as possible. On the other hand, the explanations about why conflicts exist in requirements play an important role in solving conflicts. As we retain the terms familiar to customers throughout the analysis process, this makes the complex and difficult to understand results of the verification easier to understand and accept by customers.

6. Conclusion and future work

NLRs modeling and verification are never an easy thing. In this paper, we discussed how to model and verify NLRs from the viewpoint of states and

modes. The emphasis on these two terms helps reduce the conflicts in the process of system development and validation. Customers' expectations can be expressed with their familiar terms - states and modes, in requirements. These terms will be kept and the corresponding expectations will finally be verified in requirements analysis. This provides direct feedback on customers' expectations so that customers will gain more confidence and satisfaction in obtaining the systems with their proper interests.

For this purpose, an innovative approach was proposed to conduct requirements modeling and verification from the viewpoint of states and modes. The relationship between states and modes was firstly proposed to lay the foundation for the next requirements analysis. A DSL called MoSt was designed for requirements modeling and implemented by Xtext framework. We provided this DSL with the meta-model, grammar, and semantics. In the following, a code generator was accomplished to realize the automatic model transformation from the MoSt model to the NuSMV model. A model validator was also implemented to statically check the requirements. Finally, an Eclipse-based tool was developed to perform requirements modeling and verification.

Our requirements documents are supposed to have all the information of states and modes highlighted. However, this is not always the case. Therefore, our future work will focus on how to extract the information of states and modes from requirements documents in an automatic way. This work necessitates the techniques of text processing and artificial intelligence. We are going to spend a lot of effort on this aspect. Another perspective is to integrate the identification, modeling, and verification processes into a unified process so that we will be able to provide a solution for analyzing requirements from the viewpoint of states and modes.

Acknowledgements

This work was supported by the Labex CIMI.

Appendices

```
1.  MoSt:model+=(Requirement | NLRequirement )*;
2.  NLRequirement:nlReqID=ReqID ID (ID)* '.';

3.  Requirement:
4.  MODE | STATE | PROPERTY |    CONSTRAINT | Environment;

5.  Environment:
6.  envirReqID=ReqID ID envirVariable=ID (ID)*    (('initialised' 'to'
7.  envirAttributeValue=ATTRIBUTEVALUE
8.  (envirUnit+=UNIT)*) | (range=RANG)) (ID)*'.';

9.  MODE:
10. modeReqID=ReqID 'when' preModeConditions+=MODECONDITION
11. (preRelations+=RELATION preModeConditions+=(STATECONDITON  |
12. ATTRIBUTECONTION | MODECONDITION | SIGNALCONDITION))*    ',' 'then'
13. postModeCondition = MODECONDITION '.';

14. STATE:
15. stateReqID=ReqID 'when' preStateConditions+=STATECONDITON
16. (relations+=RELATION preStateConditions+=(STATECONDITON | ATTRIBUTECONTION
17. | MODECONDITION | SIGNALCONDITION))* ',' 'then' postStateCondition =
18. STATECONDITON '.';

19. CONSTRAINT:
20. constraintReqID=ReqID 'when' preConstraintConditions+=(STATECONDITON
21. |ATTRIBUTECONTION|MODECONDITION|SIGNALCONDITION)*  (relations+=
22. RELATION preConstraintConditions+=(STATECONDITON|ATTRIBUTECONTION|
23. MODECONDITION|SIGNALCONDITION))*',' 'then' postConstraintCondition =
24. (STATECONDITON | ATTRIBUTECONTION | MODECONDITION |
25. ARITHMETICCONDITION)'.';

26. PROPERTY:
27. propertyReqID=ReqID 'when' preOperator= (CTLOperator | LTLOperator)
28. prePropertyConditions+=(STATECONDITON | ATTRIBUTECONTION |
29. MODECONDITION )*    (preRelations+=RELATION prePropertyConditions
30. +=(STATECONDITON | ATTRIBUTECONTION | MODECONDITION))* ',' 'then'
31. postOperator=(CTLOperator | LTLOperator) postPropertyConditions
32. +=(STATECONDITON | ATTRIBUTECONTION | MODECONDITION )*
33. (postRelations+=RELATION postPropertyConditions +=(STATECONDITON |
34. ATTRIBUTECONTION | MODECONDITION))* '.';
```

Figure 14: The Grammar of the Types of Formal Requirements

```
1.  STATECONDITON:
2.  ((ID (ID)* 'state' stateName=ID) | ((ID)* compOperator=COMPARISONOPERATOR) (ID)*
3.  'state' stateName=ID);
4.  MODECONDITION:
5.  ID (ID)* 'mode' modeName=ID;
6.  SIGNALCONDITION:
7.  ID (ID)* 'receives' signalName=ID ID;
8.  ReqID: '[' reqID+=INT ('.'reqID+=INT)* ']';
9.  ATTRIBUTECONTION:
10. ID (ID)* attributeName=ID ID operator=COMPARISONOPERATOR attributeValue =
11. ATTRIBUTEVALUE (unit+=UNIT)*;
12. ARITHMETICCONDITION:
13. ID result=ID (ID)* compcondition=COMPARISONOPERATOR var1=ID arithmeticOperator
14. = ARITHMETICOPERATOR (var2=ID | var3=INT);
```

Figure 15: The Grammar of the Conditions of Formal Requirements

```
1.  ARITHMETICOPERATOR:
2.  (ADD | SUBTRACTION | MULTIPLICATION | DIVISION | MODULE);
3.  MODULE: add= 'moduled' 'divided';
4.  DIVISION: division='divided' 'by';
5.  MULTIPLICATION:multiplication='multiplied' 'by';
6.  SUBTRACTION: substraction='subtracted' 'by';
7.  ADD: add='added' 'by';
8.
9.  COMPARISONOPERATOR:
10. (EQUAL | LESS | GREATER | NOTEQUAL | LESSEQUAL | GREATEQUAL | NOT);
11. NOT: not="not";
12. GREATEQUAL:
13. greateEqual+='greater' greateEqual+='or' greateEqual+='equal' greateEqual+='to';
14. LESSEQUAL: lessEqual='less' 'or' 'equal' 'to'  ;
15. NOTEQUAL: notEqual='not' 'equal' 'to';
16. GREATER: greater='greater' 'than';
17. LESS:less+='less' less+='than';
18. EQUAL: equal='equal' 'to';
```

Figure 16: The Grammar of the Arithmetic and Comparison Operators of Formal Requirements

```
1.  CTLOperator:
2.  AG | AF | EF | EG | AX;
3.  AX:ax='all' 'next';
4.  EG:eg='exist' 'globally';
5.  EF:ef='exist' 'future';
6.  AF:af='all' 'future';
7.  AG:ag='all' 'globally'  ;
8.  LTLOperator:F | G | X;
9.  F:f='future'  ;
10. G:g='globally';
11. X:x='next';
```

Figure 17: The Grammar of the Logic Operators of Formal Requirements

```
1.  RANG:
2.  compOperator1=COMPARISONOPERATOR bound1=ATTRIBUTEVALUE
3.  compOperator2=COMPARISONOPERATOR bound2=ATTRIBUTEVALUE unit=UNIT;
4.  RELATION:relation=('and' | 'or');
5.  UNIT:SPEED | ACC | TIME | WEIGHT;
6.  ACC:acc='m/s2';
7.  WEIGHT:weight='kg';
8.  TIME:time='s';
9.  SPEED:speed='m/s2';
10. ATTRIBUTEVALUE: INTTYPE | STRINGTYPE | BOOLEANTYPE;
11. STRINGTYPE:string=STRING;
12. INTTYPE:int=INT;
13. BOOLEANTYPE:value=('TRUE' | 'FALSE');
```

Figure 18: The Grammar of miscellaneous elements

Références

Ahmad, Manzoor, Nicolas Belloir, and Jean-Michel Bruel. 2015. "Modeling and verification of functional and non-functional requirements of ambient self-adaptive systems." *Journal of Systems and Software* 107 : 50–70.

Ali, Raian, Fabiano Dalpiaz, and Paolo Giorgini. 2013. "Reasoning with contextual requirements : Detecting inconsistency and conflicts." *Information and Software Technology* 55 (1) : 35–57.

Andrey, NAUMENKO. 2002. "a paradigm for General System Modeling and its applications for UML and RM-ODP." PhD diss., Ph. D thesis.

Badger, Julia, David Throop, and Charles Claunch. 2014. "Vared : verification and analysis of requirements and early designs." In *2014 IEEE 22nd International Requirements Engineering Conference (RE)*, 325–326. IEEE.

Baduel, Ronan. 2019. "An integrated model-based early validation approach for railway systems." PhD diss., Toulouse 2.

Bettini, Lorenzo. 2016. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd.

Bonnet, Stéphane, Jean-Luc Voirin, Daniel Exertier, and Véronique Normand. 2017. "Modeling system modes, states, configurations with Arcadia and Capella : method and tool perspectives." In *INCOSE International Symposium*, Vol. 27, 548–562. Wiley Online Library.

Buede, Dennis M, and William D Miller. 2016. *The engineering design of systems : models and methods*. John Wiley & Sons.

Cavada, Roberto, Alessandro Cimatti, Gavin Keighren, Emanuele Olivetti, Marco Pistor, and Marco Roveri. 2019. "NuSMV 2.5 Tutorial." Accessed 2019-03-1. `http://nusmv.fbk.eu/NuSMV/tutorial/v25/tutorial.pdf`.

Cimatti, Alessandro, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. 2002. "NuSMV 2 : An opensource tool for symbolic model checking." In *International Conference on Computer Aided Verification*, 359–364. Springer. `http://nusmv.fbk.eu/`.

Committee, IEEE Standards Coordinating, et al. 1990. "IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990). Los Alamitos." *CA : IEEE Computer Society* 169.

Davis, D, et al. 2005. "SMC Systems Engineering Primer & Handbook." *United States Air Force Space & Missile Systems Center* 13–17.

DFS. 2007. "UNMANNED SYSTEMS SAFETY GUIDE FOR DOD ACQUISITION." https://www.dau.edu/cop/esoh/DAU%20Sponsored%20Documents/Unmanned%20Systems%20Safety%20Guide%20forDOD%20Acquisition%2027June%202007.pdf.

DI-IPSC-81431A. 2000. "MIL-STD-498B (Cancelled) Data Item Description, System/Subsystem Specification." .

DMO. 2011. "Defence Materiel Organisation, DMH (ENG) 12-3-005 Function and Performance (FPS) Development Guide." .

Edwards, MT. 2003. "A Practical Approach to State and Mode Definitions for the Specification and Design of Complex Systems." In *Systems Engineering Test and Evaluation. Practical Approaches for Complex Systems Conference, Rydges Capital Hill, Canberra, Australia*, .

El Beggar, Omar, Khadija Letrache, and Mohammed Ramdani. 2020. "DAREF : MDA framework for modelling data warehouse requirements and deducing the multidimensional schema." *Requirements Engineering* 1–23.

Feiler, Peter H, Bruce A Lewis, and Steve Vestal. 2006. "The SAE Architecture Analysis & Design Language (AADL) a standard for engineering performance critical systems." In *2006 IEEE Conference on Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control*, 1206–1211. IEEE.

Giannakopoulou, Dimitra, Thomas Pressburger, Anastasia Mavridou, and Johann Schumann. 2021. "Automated formalization of structured natural language requirements." *Information and Software Technology* 106590.

Goldsby, Heather J, Pete Sawyer, Nelly Bencomo, Betty HC Cheng, and Danny Hughes. 2008. "Goal-based modeling of dynamically adaptive system requirements." In *15Th annual IEEE international conference and workshop on the engineering of computer based systems (ecbs 2008)*, 36–45. IEEE.

Harel, David. 1987. "Statecharts : A visual formalism for complex systems." *Science of computer programming* 8 (3) : 231–274.

Heitmeyer, Constance L, Ralph D Jeffords, and Bruce G Labaw. 1996. "Automated consistency checking of requirements specifications." *ACM Transactions on Software Engineering and Methodology (TOSEM)* 5 (3) : 231–261.

IEEE 24765, IEEE Standards Association, et al. 2010. "ISO/IEC/IEEE 24765 : 2010 Systems and software engineering-Vocabulary. Iso/Iec/Ieee 24765 : 2010 25021." *Institute of Electrical and Electronics Engineers, Inc* .

Jenney, J. 2011. "Define Life Cycle System Modes." Accessed on 25/11/2020, http://themanagersguide.blogspot.com/2011/01/6322-define-life-cycle-system-modes.html.

Letier, Emmanuel, and Axel Van Lamsweerde. 2004. "Reasoning about partial goal satisfaction for requirements and design engineering." In *Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, 53–62.

Leveson, Nancy G, Mats Per Erik Heimdahl, Holly Hildreth, and Jon Damon Reese. 1994. "Requirements specification for process-control systems." *IEEE transactions on software engineering* 20 (9) : 684–707.

Mavin, Alistair, Philip Wilkinson, Adrian Harwood, and Mark Novak. 2009. "Easy approach to requirements syntax (EARS)." In *2009 17th IEEE International Requirements Engineering Conference*, 317–322. IEEE.

Moitra, Abha, Kit Siu, Andrew W Crapo, Michael Durling, Meng Li, Panagiotis Manolios, Michael Meiners, and Craig McMillan. 2019. "Automating requirements analysis and test case generation." *Requirements Engineering* 24 (3) : 341–364.

Nalchigar, Soroosh, Eric Yu, and Karim Keshavjee. 2021. "Modeling machine learning requirements from three perspectives : a case report from the healthcare domain." *Requirements Engineering* 1–18.

Olver, Anthony M, and Michael J Ryan. 2014. "On a useful taxonomy of Phases, Modes, and States in Systems Engineering." In *Systems Engineering/Test and Evaluation Conference, Adelaïde, Australia*, .

Royce, Winston W. 1987. "Managing the development of large software systems : concepts and techniques." In *Proceedings of the 9th international conference on Software Engineering*, 328–338.

Silva Souza, Vítor E, Alexei Lapouchnian, William N Robinson, and John Mylopoulos. 2011. "Awareness requirements for adaptive systems." In *Proceedings of the 6th international symposium on Software engineering for adaptive and self-managing systems*, 60–69.

Szlenk, Marcin. 2006. "Formal semantics and reasoning about uml class diagram." In *2006 International Conference on Dependability of Computer Systems*, 51–59. IEEE.

Wasson, Charles S. 2010. "System Phases, Modes, and States Solutions to Controversial Issues." *Wasson Strategics, LLC. http ://www. wassonstrategics. com* .

Wasson, Charles S. 2015. *System engineering analysis, design, and development : Concepts, principles, and practices*. John Wiley & Sons.

Whittle, Jon, Pete Sawyer, Nelly Bencomo, Betty HC Cheng, and Jean-Michel Bruel. 2009. "Relax : Incorporating uncertainty into the specification of self-adaptive systems." In *2009 17th IEEE International Requirements Engineering Conference*, 79–88. IEEE.

Yu, Eric SK. 1997. "Towards modelling and reasoning support for early-phase requirements engineering." In *Proceedings of ISRE'97 : 3rd IEEE International Symposium on Requirements Engineering*, 226–235. IEEE.

Zepeda, Leopoldo, Elizabeth Ceceña, R Quintero, Ramón Zatarain, Liliana Vega, Z Mora, and Garcia Gerardo Clemente. 2010. "A MDA tool for data warehouse." In *2010 International Conference on Computational Science and Its Applications*, 261–265. IEEE.