Modeling and verification of natural language requirements based on states and modes

Yinling Liu, Jean-Michel Bruel

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

YINLING LIU*, CRAN, CNRS, Université de Lorraine, Nancy, France
JEAN-MICHEL BRUEL, IRIT, CNRS, Université de Toulouse, Toulouse, France

The relationship between states (status of a system) and modes (capabilities of a system) used to describe system requirements is often poorly defined. The unclear relationship could make systems of interest out of control because of the out of boundaries of the systems caused by the newly added modes. Formally modeling and verifying requirements can clarify the relationship, making the system safer. To this end, an innovative approach to analyzing requirements is proposed. The MoSt language (a Domain Specific Language implemented on the Xtext framework) is firstly designed for requirements modeling and a model validator is realized to check requirements statically. A code generator is then provided to realize the automatic model transformation from the MoSt model to a NuSMV model, laying the foundation for the dynamic checks of requirements through symbolic model checking. Next, a NuSMV runner is designed to connect the NuSMV with the validator to automate the whole dynamic checks. The grammar, the model validator, the code generator, and the NuSMV runner are finally integrated into a publicly available Eclipse-based tool. Two case studies have been employed to illustrate the feasibility of our approach. For each case study, we injected 14 errors. The results show that the static and dynamic checks can successfully detect all the errors.

CCS Concepts: • **Software and its engineering → Domain specific languages**; • **Theory of computation → Verification by model checking**.

Additional Key Words and Phrases: states and modes, requirements modeling and verification, domain specific language, model checking

1 INTRODUCTION

The ambiguity between states and modes threatens the safety of complex systems. If there is a problem during the development of the system, the Engineering mindset would possibly be to add one capability to prevent another capability or a scenario from occurring [39]. These capabilities have their limitations and could cause a system to fail because the values of variables exceed the thresholds of the system boundary. Thus, we need to know clearly the capabilities and the boundaries of the system. People tend to describe capabilities by using modes and states. For example, they could say that the aircraft is either in mode *taxi* or in state *taxi*. Here we argue *modes* and *states* represent the capabilities and the boundaries of the system, respectively. Indeed, it works well most of the time. However, if the system is becoming more and more complex, the described capabilities and boundaries should be more precise. The ambiguity between capabilities and boundaries has a negative impact on commanding, controlling, and monitoring the system, which finally may lead to catastrophe in reality.

For instance, the recent Boeing 737 MAX crashes caused many deaths. The summary report from the U.S. Federal Aviation Administration shows that one of the problems was that the control system read the wrong

Authors' addresses: Yinling LIU, yinling.liu@univ-lorraine.fr, CRAN, CNRS, Université de Lorraine, Campus Sciences BP 70239 54506, Vandœuvre-lès-Nancy, Nancy, France, 54506; Jean-Michel BRUEL, jean-michel.bruel@irit.fr, IRIT, CNRS, Université de Toulouse, Cr Rose Dieng-Kuntz, Toulouse, France, 31400.

ACM 0934-5043/2024/2-ART
https://doi.org/10.1145/3640822

angle of attack (AOA, which refers to the angle between the aircraft forward direction and the wing chord)[1]. The plane was supposed to ascend while, in fact, it was descending. The pilot could not take back the control of the plane, which finally gave rise to these catastrophes. To solve the problem, Boeing changed flight control laws to include a limit for MCAS (Maneuvering Characteristics Augmentation System) commands so that the MCAS will stop commanding stabilizer movement at a point that preserves enough elevator movement for sufficient pilot control of aircraft pitch attitude for current operation conditions. Boeing also added the second AOA sensor to be capable of alerting pilots that there is a problem for AOA sensors. If we analyze this example from the viewpoint of states and modes, the problem could be that when the plane is in mode *automated* and state *descending*, and we change the mode into *manual*, the plane cannot leave *descending* state to *ascending* state. If the states and modes of the plane were sufficiently addressed, the impact of modes on system behaviors would have been investigated. These catastrophes may have been avoided. This example illustrates that the boundaries, capabilities, and their relationships should be as precise as possible to keep the system safe. Therefore, the distinction of modes and states clarifies the capabilities and boundaries of the system, which helps improve the safety of complex systems.

The terms *states* and *modes* are widely used in expressing systems requirements [15–17, 34]. 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." [39]. That is the reason why we are so motivated to model and verify the natural requirements based on states and modes.

Various aspects have been emphasized to analyze requirements, including context [1, 2], the failures and successes of other requirements [37], and requirements evolution [40]. 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 proper usage of states and modes. They enable us to describe requirements that exist outside the normal operating environment [34]. They also aid in translating the user's version into the physical realization of the system [38]. They can be used as a medium to reduce misunderstandings between stakeholders such as users, acquirers, and developers as well [18].

Domain Specific Languages (DSLs) are programming languages or specification languages that target a specific problem domain [6]. When the domain of a 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 assist users to 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 and validator rules. Validator rules are user-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. However, several problems remain unsolved. For instance, how to check the internal logic between requirements? How to verify the satisfiability of property specifications regarding systems? To solve these problems, the symbolic model checking technique can further improve the quality of the written requirements. This technique is capable of demonstrating the correctness of system behaviors. 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 satisfiability 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[2] is a symbolic model checker designed to allow for the description of Finite State Machine (FSM), which ranges from completely synchronous to completely asynchronous, and from the detailed to the abstract on [11]. The primary purpose of the NuSMV input language is to describe the transition relation of the

[1]https://www.faa.gov/foia/electronic_reading_room/boeing_reading_room/media/737_RTS_Summary.pdf (accessed in July 2022)
[2]https://nusmv.fbk.eu/ (accessed in April 2023)

FSM, which is quite suitable for describing the state information in requirements. Therefore, in this paper, we choose NuSMV as the model checker.

This work is based on our previous work [30]. Our previous work mainly focuses on the design of MoSt modeling language and static checks of MoSt models. The present work significantly extends our previous work by:

- Proposing an innovative framework fully supporting the modeling and verification of requirements based on states and modes;
- Accomplishing algorithms to automatically perform the model transformation from MoSt models to NuSVM models;
- Developing a NuSMV runner to realize a seamless connection between the NuSMV model checker and validator, automating the dynamic checks including the traceability, reachability, and validation analysis;
- Developing another case study to illustrate the feasibility of our approach;
- Implementing an Eclipse-based tool to enable us to write the MoSt model, generate the NuSMV model, and conduct the static and dynamic requirement analyses.

Based on the tool, requirements engineers can use 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. Note that we suppose states and modes have already been underlined in requirements documents. So, conflicts on the system description can be reduced. 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 property specifications in CTL and LTL logics, and these specifications can be automatically verified.

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 provides an approach to verifying requirements. Section 6 gives a systematic evaluation of our approach to illustrate its feasibility. Section 7 concludes the paper with future perspectives.

2 LITERATURE REVIEW

In this section, we will first review the existing definitions of states and modes. Then, we will provide important insights into the work on requirements modeling and verification.

2.1 States and Modes

This section provides a literature review on the definition of states and modes. As one of our purposes is to differentiate states and modes, we exclude the references [13, 16–18] which are self-inconsistent in terms of the description of states and modes. For example, DI-IPSC-81431A (2000) [16] 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) [18] shares similar ideas about the relationship between states and modes. Obviously, this guidance creates more conflicts in differentiating states and modes. On the other hand, DMO (2011) [17] provides an example of a state transition diagram that 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 analyze the definitions of states and modes, including 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 implies the different levels in analyzing the system of interest, for example, system/subsystem/component. The aspect *capabilities* means a set of functions the

system can perform. The aspect *conditions* addresses the conditions that have an impact on system behaviors, for example, physical/environmental conditions. Furthermore, "involved" means 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) [3, 20] have simply emphasized 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 aids us in gaining important insights into the meaning of states and modes, which lays the foundation for designing a DSL containing these two concepts.

Table 1. An Overview of the Definitions of States and Modes where "-" and "NM" means "Not included" and "Not Mentioned", Respectively

		[3]	[14]	[20]	[15].	[26]	[38]	[27]	[8]	[7]	[5]
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 conditions	-	system conditions	a condition to a behavior	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 conditions	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	operations	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 behavior	expected behavior

2.2 Requirements Modeling and Verification

The requirements analysis has been recognized as the first phase of the system development process [36]. The later errors in the requirements are discovered, the higher the system development costs. Hence, the importance of requirements modeling and verification has been well addressed in systems 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 and implementing tools, etc. In the following, we report on a comprehensive analysis of the literature in a chronological way.

Leveson et al. (1994) [29] develop a Requirements State Machine Language (RSML) to describe requirements of real-time process control systems via a combination of graphical and tabular notations. The graphical notation of RSML is principally derived from Statecharts [23]. 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. Specifications in RSML include graphical, symbolic, tabular, and textual notations. Even though more aspects of the systems can be addressed via its notations, it seems that it is impossible to verify requirements (static and dynamic verification) in an automatic way.

Heitmeyer, Jeffords, and Labaw (1996) [24] 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. In addition, they suggest "A *system state* is a function that maps each entity name *r* in *RF* to a value" and "*RF* is a set of entity names. *RF* is partitioned into four subsets: *MR*, the set of mode class names; *IR*, the set of input variable names; *GR*, the set of term names; and OR, the set of output variable names". It seems that system states include modes. The relationship between states and modes is not mentioned. The principles of SCR and our DSL are different. We think modes in [24] are just one aspect statically describing system states. While modes in our paper actively influence system states by changing values of attributes.

Goldsby et al. (2008) [22] analyze 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 [41] 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 not easy to be formally verified. The graphic representation of requirements is informal. Transforming informal models into formal ones usually necessitates a lot of effort to ensure that the core information is captured in the formal models.

Mavin et al. (2009) [31] 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 expression. The results of their case study show qualitative and quantitative improvements compared with a conventional textual requirements specification. However, they only focus on how to express requirements in a better way. The requirements analysis concerning conflicting requirements and traceability links, etc., is missing.

Silva Souza et al. (2011) [37] 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 works. Some focus on uncertainty, assuming all the uncertain conditions are unknown and enumerated at design time [22, 28]. Some address that some uncertain conditions are still unanticipated [40]. Whittle et al. (2009) [40] 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 in 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) [4] 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) [1] propose a model-based requirement 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.

Carvalho et al. (2016) [9] investigate how to generate SCR specifications from controlled natural language requirements. We would argue that the concept "mode" is different from ours. It seems that they just mix the concepts of "state" and "mode". For example, the vending machine example in their paper says "After a coin is inserted, the system switches from *idle* to *choice* states.". This change is represented by changing the value of the signal mode from 1 to 0. It seems that 0 and 1 modes are the aliases for idle and choice states. However, in our paper, the modes show the capabilities of the system instead of representing the states of the system.

Moitra et al. (2019) [32] 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 analyses, 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. On the other hand, their viewpoint of analyzing requirements is dependent on data instead of states and modes.

Recently, Nalchigar, Yu, and Keshavjee (2021) [33] are interested in machine learning requirements (the processes for requirements elicitation, design, and development involve machine learning). They analyze these requirements from the viewpoint of businesspeople, data scientists, and data engineers. They use the case study method to perform requirements modeling. This method consists of providing an overview of the framework, constructing and revising business, analytic design and data preparation view models, and post-modeling interviews and interpretation. 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) [21] 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. Their work seems to be very close to our work. We both use the controlled natural language to describe requirements. We both use NuSMV to verify requirements. However, our work is quite different from their work. First, they do not focus on modes and states. They are particularly interested in defining the temporal logic of requirements. Secondly, the purpose of using NuSMV

for them is to check the satisfiability of a single requirement against the temporal properties. They also use EQUIVALENCE_CHECKER to check the consistency between different formalizations of the same template key. They basically check the truth of every single requirement. In our case, we check the satisfiability of every single requirement by static checking. We check not only the formats of requirements but the context of requirements (static relationship between requirements, for example, when a variable of a requirement has not been defined, the suggestion of defining this variable-related requirement will be proposed.); Thirdly, their verification does not involve dynamic checking. We check the internal logic between requirements by the NuSMV simulation.

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 [22], uncertainty [1, 40], awareness requirements [37], machine learning requirements [33], etc. Some other researchers are also concentrated on data warehouse requirements [19, 42]. 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 THE MOST MODELING LANGUAGE

In this section, we will first present the framework for modeling and verifying natural language requirements based on states and modes. Then, we will discuss our proper definitions of states and modes and their relationship in detail. Next, the meta-model of MoSt will be provided to explain the key concepts in the language. Finally, the MoSt grammar will be given.

3.1 Our Analysis Framework

This framework (Fig. 1) 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 static 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 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 it enables the investigation of how modes will influence the system behaviors. Finally, it allows system designers to better communicate with clients since “mode” is a common term that is easier to be accepted by clients when discussing specific needs.

As mentioned in Section 1, the requirements of designing a car are analyzed as a case study. They are inspired from *Dusan Rodina*[3]. Since few engineers analyze requirements from our perspective, it is not easy to find the

[3]https://www.softwareideas.net/a/1539/Car-States--UML-State-Machine-Diagram- (accessed in April 2023)

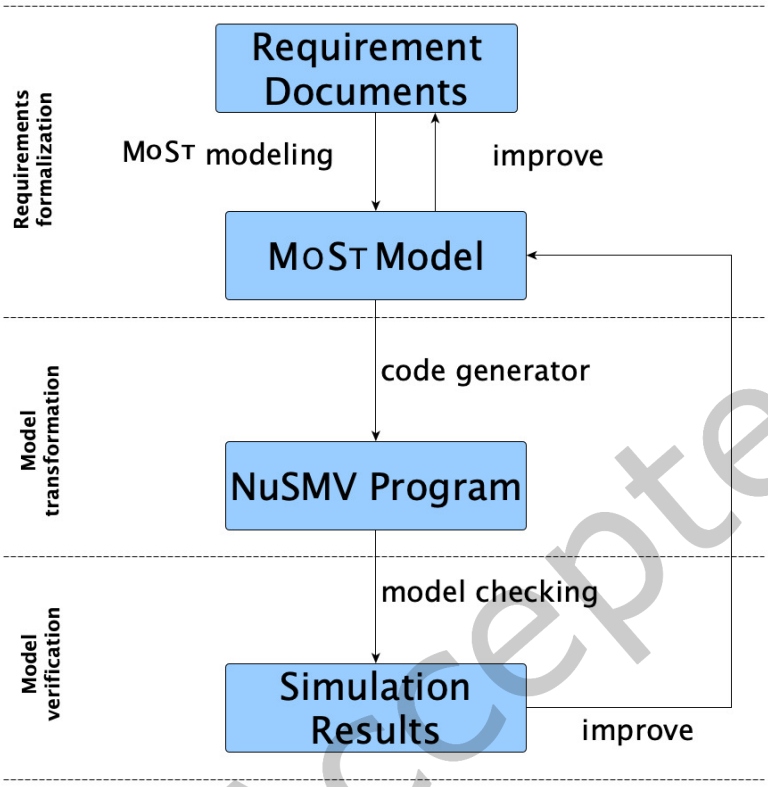

Fig. 1. Framework for Modeling and Verifying Natural Language Requirements based on States and Modes

existing requirements without reformulating them. We think this will not hinder the applicability of the proposed strategy for requirements verification. The proposed strategy can be used to describe requirements as generally as possible. It requires that users need to pay attention to the concepts of states and modes. Indeed, it will constrain a little the way we express requirements, but it gains more clarity and reduces ambiguity. Additionally, 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 be considered in the following requirements modeling and verification activities.

3.2 Relationship between States and Modes

The relationship between states and modes has been discussed in the literature. For example, the explanation of Edwards [18] 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 capabilities. While states are changed when conditions are satisfied. It seems that Wasson [39] 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 mentioned by [39]. 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. 2 illustrates the relationship between states and modes.

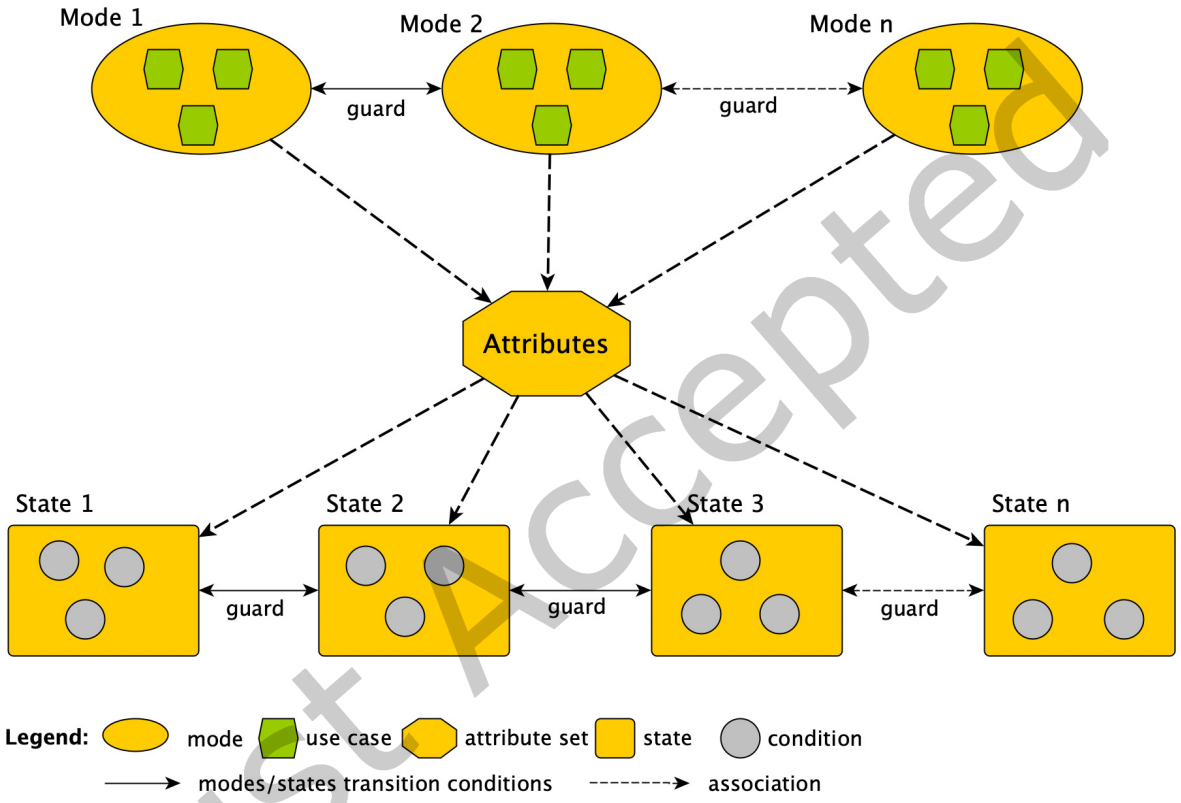

Fig. 2. The Relationship between States and Modes

3.3 MoSt Meta-model

The MoSt meta-model highlights the properties of the MoSt Modeling Language. As shown in Fig. 3, MoSt 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.

MoSt focuses on describing functional requirements non-functional requirements. MoSt-based formal requirements consist of concepts *Mode*, *State*, *Constraint* and *EnvironmentRequirement*. The reason why we need to introduce the last two concepts is that the MoSt model is supposed to support formal verification. This idea

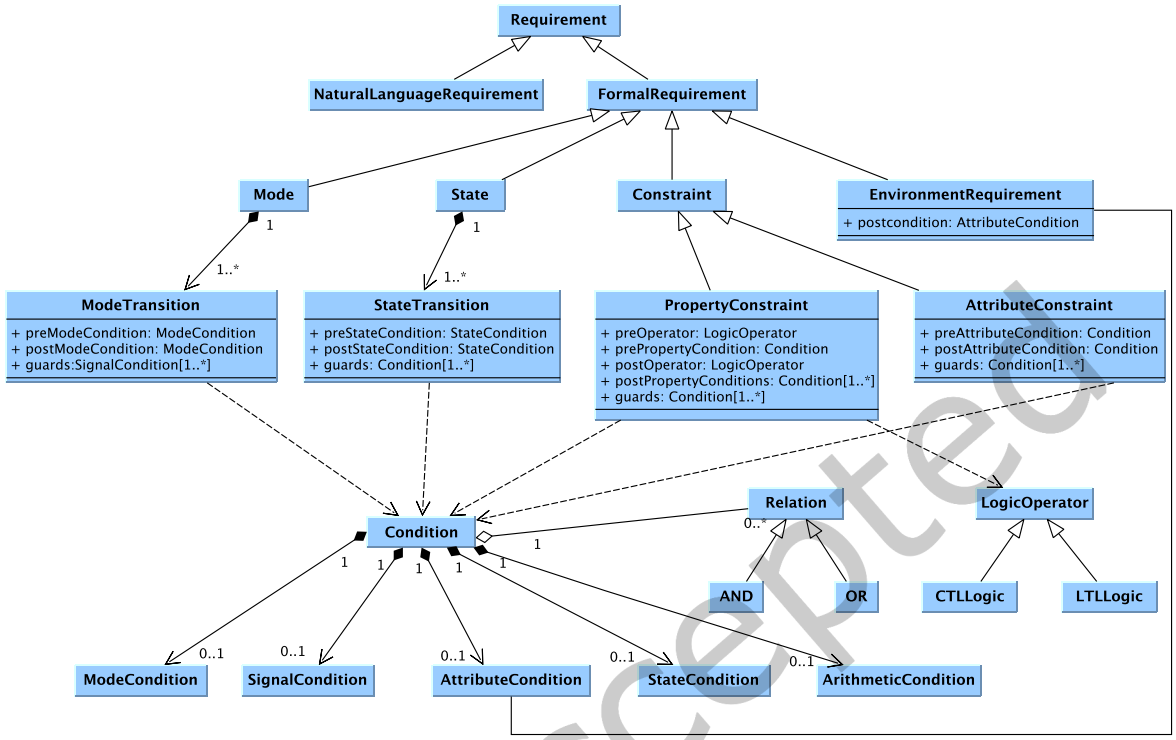

Fig. 3. The MoSt Meta-model

demands that the MoSt model must be self-contained to make it verifiable. The concepts of *Mode* and *State* describe functional requirements. The *Constraint* concept describes functional and non-functional requirements. More specifically, the concept *PropertyConstraint* depicts functional and non-functional requirements to be checked. On the other hand, *EnvironmentRequirement* concept initializes values and ranges of system attributes.

Concept *Constraint* aims at constraining attributes mentioned in requirements and providing property specifications to be checked in formal analysis.

Concept *Condition* is one of the most important concepts in this meta-model, which includes conditions *ModeCondition*, *StateCondition*, *AttributeCondition*, *SignalCondition*, *ArithmeticCondition*, and *Relation*. Concept *Relation* is used to enrich the expressive power of MoSt, which enables guards to be any combination of conjunctions, disjunctions, and conjunctions and disjunctions of specific conditions. However, concepts of *ModeTransition* and *EnvironmentRequirement* are exceptional, which are directly associated with specific conditions. The reasons have been mentioned in Section 3.2.

Concept *LogicOperator* describes two temporal logics: the logics *CTLLogic* (Computational Tree Logic) and *LTLLogic* (Linear Temporal Logic). The introduction of the concept *LogicOperator* lays the foundation for formally verifying requirements based on states and modes.

Additionally, environment requirements include only post-conditions because they are used to state the initial statuses of variables where preconditions are not needed.

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 [6]. The MoSt grammar defines the rules describing how to write different types of requirements. The complete grammar is shown in the annex (Figs. 23 - 27). 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. Here, the collection is implemented as a list. The star operator * means that the number of the elements can be any number >=0. The grammar of MoSt is not presented using the classical BNF (or EBNF) notation but using the syntax of Xtext since it is more understandable if interested readers would like to reproduce this work. Interested readers can also get details from [6].

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 identifiers.

3.4.2 Formal Requirements. Formal requirements include *ENVIRONMENT*, *MODE*, *STATE*, *ATTRIBUTECONSTRAINT*, and *PROPERTYCONSTRAINT*. The rule of formal requirements is represented as follows:

Requirement: ENVIRONMENT | MODE | STATE | ATTRIBUTECONSTRAINT | PROPERTYCONSTRAINT;.

Two generic templates are used including "when ..., then ..." and "... should be ...". The first template applies to mode, state, attribute constraint, and property constraint requirements. The second template applies to environment requirements. The first one implies when pre-conditions of the systems are satisfied, then the system can get the post-conditions satisfied. The idea is basically from Hoare Triple logic [25]. That is why we would like to use different kinds of specific conditions to define pre-conditions, post-conditions. Our template is similar to one of the EARS templates "WHEN <optional preconditions> <trigger> the <system name> shall <system response>" [31]. We argue "when ..., then ..." can express all the templates mentioned in their work in an abstract way because we think pre-conditions can express the characteristics of all their templates including ubiquitousness, events, unwanted behaviors, states, and optional features. A ubiquitous requirement in EARS template has no precondition or trigger. Property constraint requirements can describe ubiquitousness. For example, the CTL formula "AG ! engineSpeed > x" represents that the control system shall prevent engine overspeed. Optional features can be simply recognized as conditions. The other characteristics are naturally supported by the first template. The second template declares constraints for variables, which are suitable for describing environment requirements. Besides, this mapping illustrates the expressive power of MoSt.

1). Environment Requirements

Environment requirements are dedicated to describing initial statuses of variables, including the initialized values and the ranges of variables. That is why the rules of these requirements involve *ATTRIBUTEVALUE, UNIT* and *RANGE*. The rule of environment requirements is 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 25 and 27. Note that *ID* is predefined in Xtext framework as terminal ID: '^'?('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;. *ID* and *(ID)** allow to write only one word and any number of words, respectively.

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"	bound1 = "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/s^2.
Req 2: [2.2.2] The accSpeed should be greater or equal to 0 and less or equal to 10 m/s^2.

2). Mode Requirements

Mode requirements explain mode transitions that are associated with mode and signal conditions. Mode conditions indicate which mode the system is in. Signals are the core conditions for triggering mode transitions. The rule of mode requirements is listed as follows:
MODE:
modeReqID=ReqID 'when' preModeCondition = MODECONDITION (relations+=RELATION guards += SIGNALCONDITION) ',' 'then' postModeCondition = MODECONDITION'.' ;*
The details of the syntax are shown in the annex. 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 = "mode = economic"
	relation = "and"	guards [1] = "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, and signal conditions. The rule of state requirements is illustrated as follows:

STATE:

stateReqID=ReqID 'when' preStateCondition+=STATECONDITON (relations+=RELATION guards += (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 = "state = accelerate"
	relations[1] = "and"	guards[1] = "signal = Auto"
	relations[2] = "and"	guards[2] = "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/s^2, then it will be in state autonomy.

4). Attribute Constraint Requirements

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

ATTRIBUTECONSTRAINT:

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

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

Table 5. Instance of Attribute Constraint Requirement Req 5

Req 5	attributeReqID = "5.2"	preAttributeCondition = "accSpeed = 5 m/s^2"
	relations[1] = "and"	guards[1] = "speed > 80 km/h"
		postAttributeCondition = "displaySpeed = TRUE"

The corresponding MoSt code can be written as follows:

Req 5: [5.2] when the car accSpeed is equal to 5 m/s2 and its speed is greater than 80 km/h, then its displaySpeed is equal to TRUE.

5). Property Constraint Requirements

Property constraint requirements aid in describing functional 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 constraint requirements are expressed as follows:

PROPERTYCONSTRAINT:

propertyReqID=ReqID 'when' preOperator= (CTLOperator | LTLOperator) prePropertyCondition = (STATECONDITON | ATTRIBUTECONTION | MODECONDITION) (preRelations+=RELATION guards += (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 constraint requirement attributes. This requirement provides a CTL specification to check the function of the car. Note that post-property conditions can be an array because multiple post-property conditions can be imposed by property constraint requirements.

Table 6. Instance of Property Constraint Requirement Req 6

Req 6	propertyReqID = "7.1"	preOperator = "all globally"
	prePropertyCondition = "state = autonomy"	preRelations[1] = "and"
	guards [1] = "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 in state autonomy and it is in mode economic, then all next it is not in state accelerate.

Since we plan to transform a MoSt model into the corresponding NuSMV model, the semantics of the MoSt model will be the semantics of its underlying NuSMV model. NuSMV is the first model checker based on BDDs (Binary Decision Diagrams), which is a reimplementation and extension of SMV [12]. NuSMV has been designed to be an open architecture for model checking, which can be reliably used for the verification of industrial designs, as a core for custom verification tools, as a testbed for formal verification techniques, and applied to other research areas. We choose not to introduce the semantics to avoid repetition. The details of NuSMV semantics can be seen in NuSMV 2.6 User Manual[4].

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. 4. The implementation of the process for the automatic model transformation is based on this mapping. The process is realized in Algorithms 1 - 5, which will be discussed in the following. The presented algorithms do not cover all details but focus on the main aspects of the mapping process. Note that Fig. 4 may give the wrong impression that the concept of modes exists already in NuSMV. In fact, we define modes as variables in NuSMV. Modules (NuSMV programs) have been designed to implement the functions of modes we defined.

As mentioned in Section 3, the MoSt modeling language consists of mode, state, environment, property, and 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 regarded 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 main mode requirement transformation is implemented in Algorithm 1.

[4]https://nusmv.fbk.eu/NuSMV/userman/v26/nusmv.pdf (accessed in April 2023)

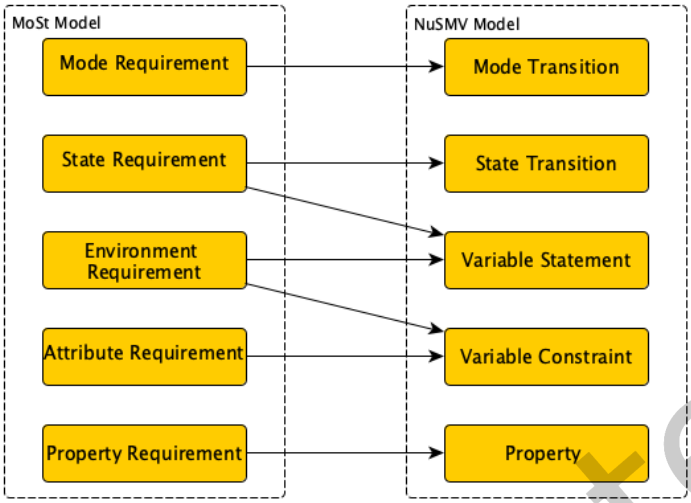

Fig. 4. Mapping Between Modules of the MoSt Model and the NuSMV Model

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 [10]. Thus, state modules can be transformed into the corresponding NuSMV code. The state requirement transformation is performed in Algorithm 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 property module is feasible as well. The processes for the model transformation for environment, and attribute modules are achieved in Algorithms 3-5 respectively. Technically speaking, implementing the model transformation depends on the MoSt grammar and the structure of the NuSMV code. The grammar helps identify different kinds of requirements and extract the corresponding condition information. We then need to reorganize this information to make it suitable for the NuSMV parser. The time complexities of Algorithms 1-5 are all $O(n^2)$ except for Algorithm 3 which is $O(n)$.

In order to make the process clearer, the car requirements have been formalized in Fig. 5. Five kinds of requirements have been elicited to illustrate the transformation process. Note that the complete NuSMV code is shown in Annex B. Requirement [8.1] is a natural language requirement explained in Section 3.4.1, which will not be translated.

5 REQUIREMENTS VERIFICATION

In this section, we first introduce the architecture of the MoSt modeling tool. Based on this tool, we discuss how we realize requirements static and dynamic checks.

5.1 MoSt Modeling Tool

The MoSt modeling tool is implemented by the Xtext framework in Eclipse. Fig. 6 shows the architecture of the MoSt modeling tool. This tool consists of four components: Editor, Validator, Generator, and Model checker. Editor includes a UI and MoSt language. UI is an Eclipse UI which aids in writing the MoSt model. MoSt is

Algorithm 1: Mode Requirement Transformation

```
input : root
output: modeTransitions, modes
// The variable root represents the root of the MoSt model.
ArrayList<String> modeTransitions = new ArrayList<String>();
HashMap<String, String> modes = new HashMap<String, String>;
String temp="";
int indexMode = 0;
for modeReq : root.model.filter(MODE) do
    indexMode = 0;
    temp ="";
    for preModeCondition : modeReq.preModeConditions do
        temp+=preModeCondition.condition;
        if indexMode <= modeReq.relation.size - 1 then
            temp+=modeReq.relation.get(indexMode++).relation ;
        end
    end
    // Post-mode conditions look like "mode = A", the value of the mode is just required, conforming to
       NuSMV code rules.
    temp+=postModeCondition.condition.split("=").get(1);
    modeTransitions.add(temp);
end
// Extract modes from mode transitions
for modeTransition : modeTransitions.entrySet do
    if modeTransition!=null then
        modes.put(modeTransition.keysplit("&").get(0).split("=").get(1)), "mode");
    end
end
```

Algorithm 2: State Requirement Transformation

```
input : root
output: stateTransitions
ArrayList<String> stateTransitions = new ArrayList<String>();
String temp="";
int indexState = 0;
for stateReq : root.model.filter(STATE) do
    indexState = 0;
    temp ="";
    for preStateCondition : stateReq.preStateConditions do
        temp+=preStateCondition.condition;
        if indexState <= stateReq.relation.size - 1 then
            temp+=stateReq.relation.get(indexState++).relation ;
        end
    end
    // Post-state conditions look like "state = A", the value of the state is just required, conforming to
       NuSMV code rules.
    temp+=postStateCondition.condition.split("=").get(1);
    stateTransitions.add(temp);
end
```

Algorithm 3: Environment Requirement Transformation

```
input  : root
output : variableConstraints, variables
HashMap<String,String> variableConstraints = new HashMap<String,String>();
HashMap<String,String> variables = new HashMap<String,String>();
String key="";
String pre="";
double max,min;
max=min=0;
for environmentReq : root.model.filter(ENVIRONMENT) do
    key = environmentReq.envirVariable;
    // initializing variables
    if environmentReq.range == null then
        pre = variableConstraints.get(key);
        if pre == null then
            pre="";
        end
        // The initial value of variables is stored in the left part of @ of pre.
        pre = environmentReq.envirAttributeValue.attributeValue + "@" + pre;
        variableConstraints.put(key,pre);
    end
    // setting the scope for variables
    else
        if environmentReq.range.bound1.attributeValue <= environmentReq.range.bound2.attributeValue then
            min=environmentReq.range.bound1.attributeValue;
            max=environmentReq.range.bound2.attributeValue;
        end
        else
            min=environmentReq.range.bound2.attributeValue;
            max=environmentReq.range.bound1.attributeValue;
        end
        variables.put(key,min+ ".."+max);
    end
end
```


created based on the Eclipse Modeling Framework and the core of Eclipse. Validator is a static checker written by Xtend language, which checks the MoSt model against the static rules. Generator is a MoSt-To-NuSMV transformer written by Xtend, which allows us to automatically transform MoSt models into NuSMV models. Finally, a NuSMV runner is a dynamic checker written as a Java class, linking the NuSMV model checker with this tool, in order to automate dynamic checks.

The implementation of the NuSMV runner is based on the method "java.lang.ProcessBuilder.ProcessBuilder(List< String> command)". This process builder helps us call the NuSMV model checker with the argument "List<String> command". The argument comprises two constant strings in sequence: "NUSMV_EXECUTE_PATH" and "GENERATED_SMV_FILE _PATH". The method "processBuilder.start()" starts the call of the checker, which produces results included in "p.getInputStream()" and "p.getErrorStream()". Input streams store the output of the checker except for the error information which is stored in error streams.

It should be noted that the information of errors implies the NuSMV model is not correct rather than the automatically generated specifications or user-defined specifications are false. The incorrectness of the NuSMV model

Algorithm 4: Attribute Requirement Transformation

```
input  : root
output : variableConstraints
HashMap<String,String> variableConstraints = new HashMap<String,String>();
String temp="";
String pre, key;
int indexAttribute = 0;
for attributeReq : root.model.filter(ATTRIBUTE) do
    indexAttribute = 0;
    temp ="";
    for preAttributeCondition : attributeReq.preAttributeConditions do
        temp += preAttributeCondition.condition;
        if indexAttribute <= modeReq.relation.size - 1 then
            temp+=attributeReq.relation.get(indexAttribute++).relation;
        end
    end
    key = attributeReq.postAttributeCondition.condition.split("=").get(0);
    temp += ":"+attributeReq.postAttributeCondition.condition.split("=").get(1)+";";
    pre = attributeConstraints.get(key);
    if pre!=null then
        temp += pre;
    end
    if pre != temp then
        attributeConstraints.put(key,temp);
    end
end
```

means this model hides run-time errors. The automatically generated specifications signify the specifications must need to be satisfied. The user-defined specifications signify the expectations of the users which are not necessarily realistic. The NuSMV runner functions with the validator via files such as "*.smv" and "*.txt". The executing sequence is illustrated as follows:

- Users write requirements in the editor; the validator dynamically executes static checks with static rules;
- Once the MoSt model is ready in the editor and saved, the code generator will be executed to transform the MoSt model into the NuSMV model;
- The NuSMV runner is called to simulate the NuSMV model by pressing the space key in any empty place of the editor; the runner then analyzes the results to make them understandable to the Validator;
- Finally, the Validator checks the MoSt model by the analyzed results to put them to the relevant requirements.

A screenshot of the tool is shown in Fig. 7. This project is publicly available on GitHub[5]. The details of the requirement checking will be discussed in the next sections.

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 static rules. Requirements static checks are triggered while writing the MoSt code. If the static rules are violated, errors will pop up in the MoSt modeling editor.

[5]https://github.com/liuyinling/MoSt-Modeling-Tool.git

Algorithm 5: Property Requirement Transformation

```
input  : root
output : propertySpecifications
ArrayList<String> propertySpecifications = new ArrayList<String>();
String temp="";
int indexPreProperty, indexPostProperty;
for propertyReq : root.model.filter(PROPERTY) do
    indexPreProperty = 0;
    indexPostProperty = 0;
    temp = propertyReq.preOperator.logicOperator;
    for prePropertyCondition : propertyReq.prePropertyConditions do
        temp+=prePropertyCondition.condition;
        if indexPreProperty <= propertyReq.preRelation.size - 1 then
            temp+=propertyReq.preRelations.get(indexPreProperty++).relation;
        end
    end
    temp += propertyReq.postOperator.logicOperator;
    for postPropertyCondition : propertyReq.postPropertyConditions do
        temp+=postPropertyCondition.condition;
        if indexPostProperty <= propertyReq.postRelation.size - 1 then
            temp+=propertyReq.postRelations.get(indexPostProperty++).relation;
        end
    end
    propertySpecifications.add(temp);
end
```

Requirements static checks include naming checks and consistency checks. Naming checks in the MoSt model concerning states, modes, and signals are: the names of states and modes should start with a lower case; the name of signals should begin with an upper case. Listing 1 shows the implementation of mode name checks. In the Xtext framework, rules are implemented in the validation package. Each rule starts with *@Check*. Since class *MoStMLValidator* (where we implement rules) extends class *AbstractMoStMLValidator*, we can directly access model objects such as *MODECONDITION*.

Listing 1. The Implementation of Mode Name Checks

```
@Check
def void checkModeName(MODECONDITION modeCondition){
    if(modeCondition.getModeName().charAt(0)<='Z'&&modeCondition.getModeName()
    .charAt(0)>='A'){
        error("Mode name should start with a lower case"+": error
        '"+modeCondition.getModeName().charAt(0)+"'",
        MoStMLPackage.Literals.MODECONDITION__MODE_NAME, INVALID_ReqID);
    }
}
```

Consistency checks analyze the legality of requirements, regarding static rules. For example, one of our rules is "Variables should only be initialized once". If two requirements are written as "[1.1] The doorIsOpen should be initialized to FALSE." and "[1.2] The doorIsOpen should be initialized to TRUE.", then error "Variables should

Fig. 5. An Example of the Model Transformation

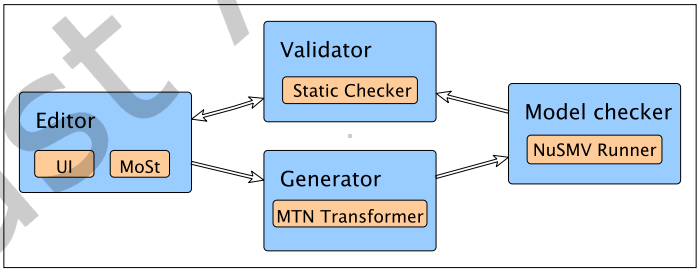

Fig. 6. Architecture of the MoSt Modeling Tool

only be initialized once! 'doorIsOpen"' will be shown on these two requirements. CC3 intends to define the value ranges for variables. CC6 is a syntactic check for state requirements. The requirements consistency checks (CC) include, but not limited to:

CC1: Variables should only be initialized once;

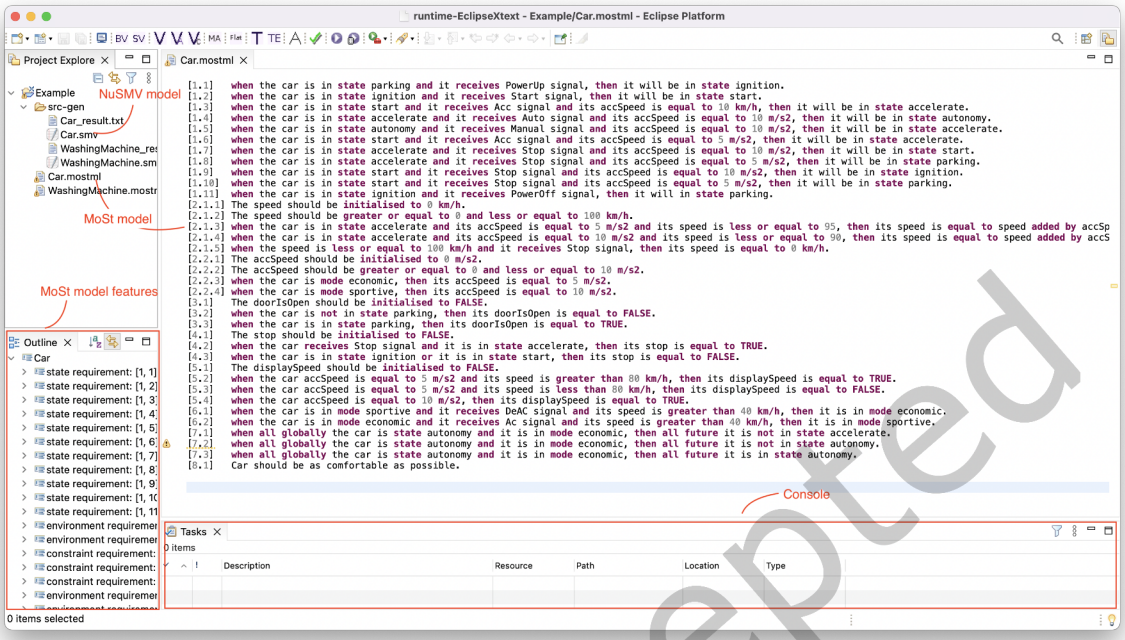

Fig. 7. Screenshot of the MoSt Modeling Tool

CC2: The variables mentioned in attribute requirements should be initialized;
CC3: The variable of *Integer* should be given the scope;
CC4: The repetition of requirement IDs is not allowed;
CC5: The repetition of requirements is not allowed;
CC6: Different post-conditions of state requirements cannot have the same preconditions and guards.
CC7: The variable of attribute conditions must be defined before using it.

Obviously, this list may be not exhaustive. However, adding new rules is possible. The Xtext framework allows us to use Validator to achieve these rules. Since the implementation of the naming and consistency rules is provided in the tool, it illustrates the way how we extract data from the model. Adding new rules simply involves the manipulation of extracted data. Indeed, they are helpful in writing a proper MoSt model. However, they cannot guarantee that the generated NuSMV model is correctly executable. In other words, we are not sure that the internal logic of the system is correct. For example, if "division by zero" exists, static checks cannot find this error. Therefore, requirements dynamic checks will be discussed in the following.

5.3 Requirements Dynamic Checking

Requirements dynamic checks rely on the model checker NuSMV. Requirements dynamic checking includes the analysis of the generated NuSMV model, the user-defined specifications, and the automatically generated specifications.

5.3.1 Traceability analysis. Traceability analysis offers a general approach about how to use the results of model checking to improve the quality of the requirements model. This analysis involves three types of errors: program errors, user-defined specification errors, and automatically generated specification errors. Program errors mean the NuSMV model has run-time errors because of bugs such as type errors, division by zero, etc. User-defined specification errors denote property requirements that are false. The last errors indicate the automatically generated properties which are false. Three types of errors require three specific analysis methods, respectively. The method for the first type needs to extract information from simulation results, including IDs or names of variables. If the line number does not relate to the code commented by the requirement ID, we should extract the names of the relevant variables. The way of extracting the names of variables depends on the contents of the errors identified by NuSMV. If errors explicitly indicate the names of variables, we extract the names directly. If not, we should infer the relevant variables from the simulation results. The inference comes from the manual analysis of the outputs of NuSMV against the errors of the same type. Currently, we found the lines "line number - 1" always show variable names. The relevant constraint requirements can be matched via variable names.

Fig. 8 shows an example of the traceability analysis for program errors. The traceability analysis involves three files: "Car_result.txt", "Car.smv", and "Car.mostml". The indexes indicate the sequence of analysis. In this example, we first found the keyword "variable" existed (step 1). Then, we found the line number was 88 (step 2-1). At the same time, the error information "cannot assign value 105 to variable speed" was stored (step 2-2). Next, we searched line 88 in the file "Car.smv" (step 3) and extracted requirement ID *[2.1.6]* (step 4). With the ID, we finally attached the error information to the corresponding requirement in the file "Car.mostml" (step 5). How to attach errors to the requirements can be seen in Listing 1. It should be noted that the list of errors of different output forms may not be exhaustive, which means if the NuSMV outputs another new type of error, our tool will not immediately be able to deal with it. However, we provide an interface printing "this tool is not able to handle this error - XXX, please report it to the developers" when a new type of errors appears.

The second method finds IDs by comparing property specifications. They are specifications of user-defined and re-assembled. We should re-assemble properties because the presentation of specifications is a little bit different from that of the NuSMV model. The extracted IDs serve as one of the links to ensure traceability between requirements and the NuSMV code since the code commented by requirements IDs corresponds to requirements. Fig. 9 gives an example of the traceability analysis for user-defined specification errors. This analysis also necessitates the three files mentioned above. In this example, we first found the keyword "false" (step 1). Then we extracted the property specification formula (step 2-1) and the counter-example (step 2-2). Based on the formula, we found the same formula in the file "Car.smv" (step 3) and extracted the corresponding requirement ID *[7.2]* (step 4). With the ID, we finally attached the counter-example to the corresponding requirement in the file "Car.mostml" (step 5).

The last method is simply to extract state conditions from the specifications because they are automatically generated. The third kind of errors is only concerned with state requirements. The reason can be seen in Section 5.3.2. We can also use state conditions to locate requirements with errors because they are used to match state requirements with the same post-condition. Fig. 10 presents an example of the traceability analysis for automatically generated specification errors. This analysis is only related to two files: "Car_result.txt" and "Car.mostml". In this example, we first identified the keyword "false" (step 1). We then noticed the keyword "EF" existed as well (step 2). Next, we extracted "state = autonomy" and the counter-example. In the following, we only identified requirement [1.5] where reaching state *autonomy* was one of the conditions (step 4). So, we finally attached the counter-example to the corresponding requirement in the file "Car.mostml" (step 5).

Fig. 11 shows the diagram for processing simulation results regarding different types of errors. The three analysis methods are implemented in the NuSMV runner. It should be noted that the NuSMV model checker produces program errors one by one and generates all the results of specifications checks at one time. If there are

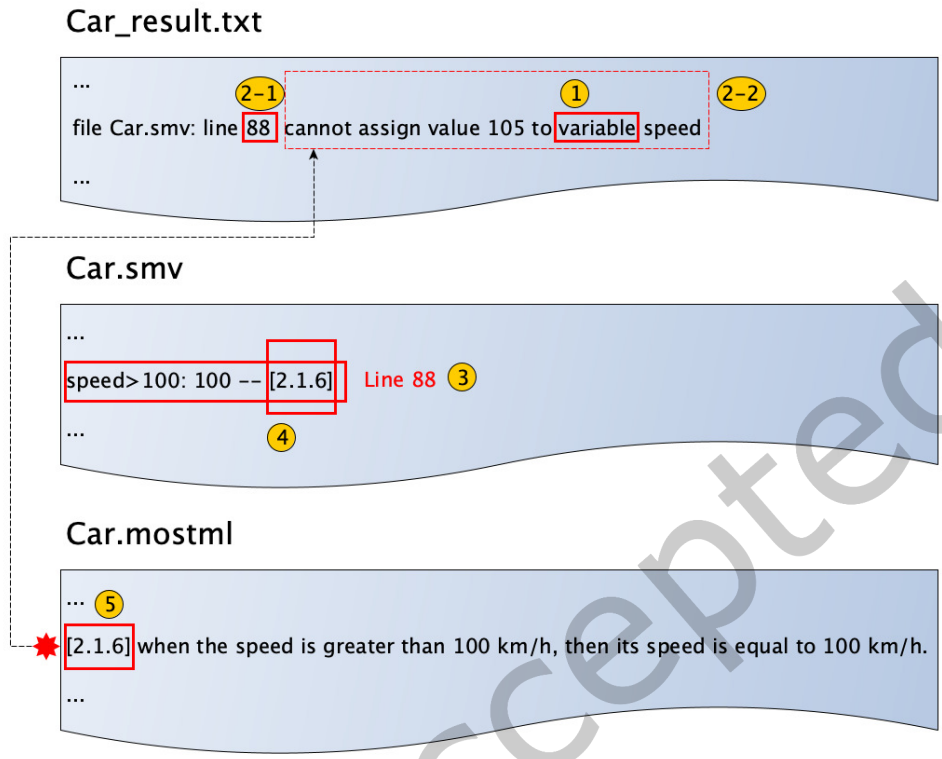

Fig. 8. Traceability Analysis for Program Errors

more than one program error, we should correct one, save the model, and press the space key, iteratively until they are all solved.

Since we have built the links between simulation results and requirements, we will continue to take advantage of model checking to help us verify the correctness of the requirements model. As the successfully compiled NuSMV code does not necessarily mean the states in the requirements are all reachable, therefore, we would like to check the reachability of the requirements.

5.3.2 Reachability analysis. Reachability analysis concerns the checking of the automatically generated property specifications, which provides the possibility to check the reachability of states. 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. So, the reachability analysis only relates to the state requirements. To facilitate the analysis process, additional property specifications for the reachability of each state are generated automatically from the previous model transformation.

5.3.3 Validation analysis. Validation analysis checks the satisfiability of property requirements. Property requirements are transformed into the CLT/LTL specifications in the NuSMV model. Thus, validation analysis checks the satisfiability of CTL/LTL specifications regarding the generated 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

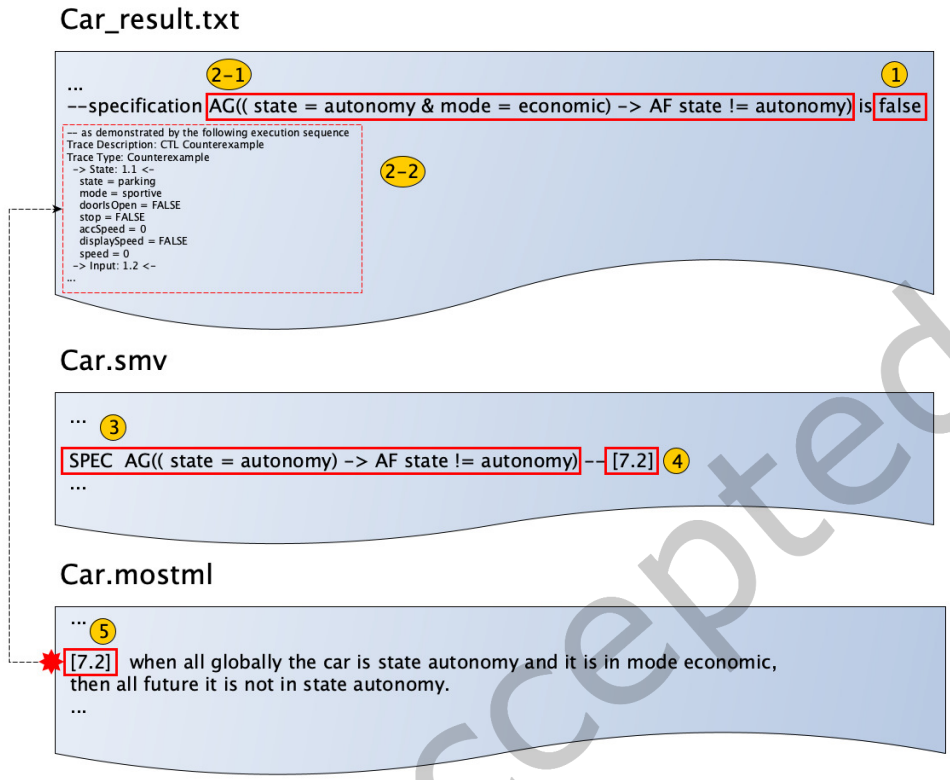

Fig. 9. Traceability Analysis for User-defined Specification Errors

modified or deleted as early as possible. This can avoid conflicts between developers and customers when the system is delivered.

Most importantly, as mentioned at the beginning of Section 1, when the system of interest is becoming more and more complex and intelligent, we need to check whether the conflicts between modes and states exist. In other words, the newly added capabilities to make the system more intelligent may cause the system out of boundary. This out-of-boundary may lead to catastrophes in safety-critical areas. To check whether the system has conflicts, we propose to use property specifications conforming to *SPEC AG((state = X & mode = M) − > AF state = Y*. The specifications of this form imply when the system is in state X and we change the mode to M, which state the system will eventually be in. If we do want to change its state with the activation of mode M, the future state should change. So, the specification should be true in this case. If the property specification turns out to be false, requirements designers should either make it true via the analysis of counter-examples or ensure that this situation is allowed in the system.

6 EVALUATION

In order to illustrate the feasibility of our approach, we gave systematic evaluations of a car example and a washing machine example. We discussed the first case study (the car example) in detail. In the second case, only necessary explanations were provided to prevent redundancy.

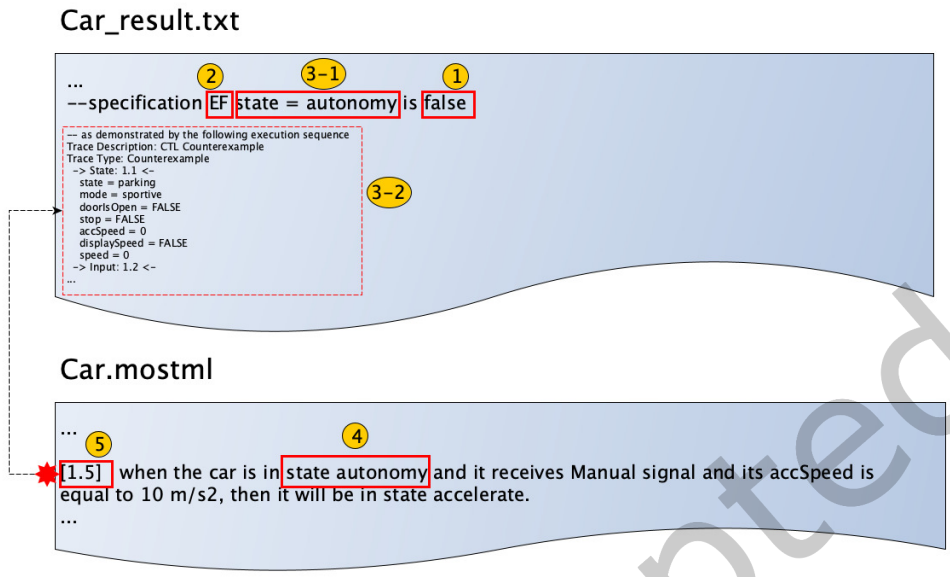

Fig. 10. Traceability Analysis for Automatically Generated Specification Errors

6.1 A Car Example

6.1.1 System description. This example is based on a UML state machine diagram for a car[6]. The car has five states including *Parking, Ignition, Start, Accelerate,* and *Autonomy*. We assume that this car has the function of autonomous driving. The details of the car's state transitions are shown in Fig. 12. Signals and attributes constitute the conditions for state transitions. Modes influence the system behaviour via changing the value of attributes. Here, we provide this car with two modes *economic* and *sportive*. They are related to variables *accSpeed* and *displaySpeed*. For example, if the car is in mode *economic* and mode *sportive*, it changes the acceleration speed to 5 m/s2 and 10 m/s2, respectively. Note that we avoid discussing all the details of the system. Interested readers can see the details of the impact of modes on variables in Annex A.

6.1.2 Requirements formalization & Static checking. Formalizing requirements using MoSt can start with either requirements documents or the state machine diagram of the system. Since our case comes from a diagram, we will formalize the requirements of the car, based on Fig. 12. If we start with requirements documents, we will also need to extract the information on state transitions.

The procedures for requirements formalization consist of:

State transition writing all the state transitions from requirements; normally, the editor will show errors where attributes are not defined;

Attribute declaration defining all the missing attributes (this step will trigger the validator to pop up error messages explaining the missing parts such as no initialization, no scope, etc.); the information about the impacts of other conditions on attributes must be captured from requirements (if the information on modes and attributes is missing, errors will be displayed as well);

[6]https://www.softwareideas.net/a/1539/Car-States--UML-State-Machine-Diagram- (accessed in April 2023)

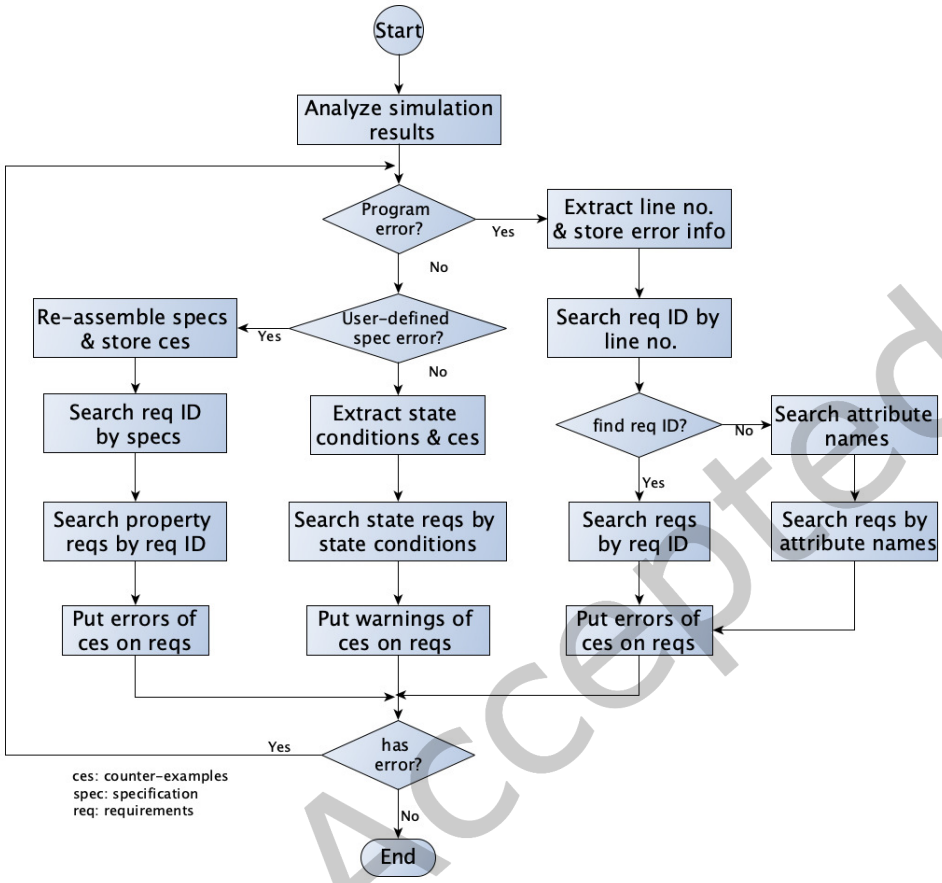

Fig. 11. The Diagram for Processing Simulation Results

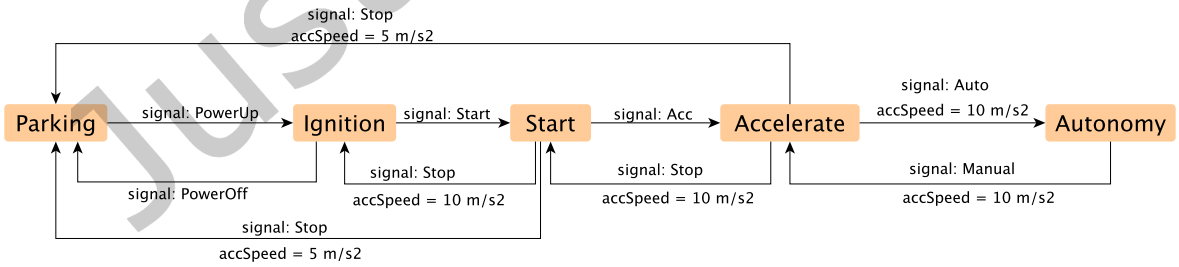

Fig. 12. The State Machine Diagram of a Car

Mode transition the information about mode transitions must be obtained from the requirements (if the information on attributes is missing, errors will be displayed); defining mode transitions;

Property declaration the information about the property must be obtained as well (if the information on states, modes, and attributes is missing, errors will be displayed); defining properties.

According to the above-mentioned procedures as well as the MoSt grammar and static checks, we provide examples for each part, which are illustrated as follows:

State transition :

[1.1] when the car is in state parking and it receives PowerUp signal, then it will be in state ignition.

Attribute declaration :

[2.1.1] The speed should be initialised to 0 km/h.

[2.1.2] The speed should be greater or equal to 0 less or equal to 100 km/h.

[2.1.3] when the car is in state accelerate and its accSpeed is equal to 5 m/s2, then its speed is equal to speed added by accSpeed.

Mode transition :

[6.1] when the car is in mode sportive and it receives DeAC signal and its speed is greater than 40 km/h, then it is in mode economic.

Property declaration :

[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.

In this example, "AC" and 'DeAC' imply the car accelerates or decelerates. The details of the MoSt model are shown in Annex A.

It should be noted that all the error messages popping up come from requirements static checks. It helps us write statically correct MoSt models. Fig. 13 shows the potential errors requirements static checks may find. These static checks ensure the well-formedness of the system requirements, which is very useful when there are a number of requirements that are frequently evolving.

6.1.3 Requirements dynamic checking.

Traceability analysis. To better demonstrate the feasibility of the traceability analysis, we manually inject three errors (Errors 2-4, shown in Fig. 14) in the MoSt model, including division by zero, illegal type of "case" list element, and illegal operand type. The editor pops up an unexpected error concerning the scope of the variable of "speed" (Error 1). Fig. 14 shows the errors appear in the editors. The four trace links are explained in Table 7. Errors 1, 3, and 4 are traced by variable names which are found via line numbers. Error 2 is traced directly by the requirement ID.

Table 7. The Details of Four Trace Links for the Car Example

No.	Error	Information	Correction
1	scope of variable "speed"	variable: speed	add range constraints for each branch
2	division by zero	ID: [2.1.3]	'0' − > 'x' (x! =0)
3	illegal types of "case" list elements: boolean and integer	variable: doorIsOpen	'1' − > 'TRUE'
4	illegal operand types of "+": integer and boolean	variable: displaySpeed	'displaySpeed' − > v

As for Error 1, we basically declare "speed" as a variable of range [0,100]. The MoSt and NuSMV code concerning variable "speed" are shown as follows:

```
MoSt code:
[2.1.1] The speed should be initialised to 0 km/h.
[2.1.2] The speed should be greater or equal to 0 less or equal to 100 km/h.
[2.1.3] when the car is in state accelerate and accSpeed is equal to 5 m/s2,
```

```
then its speed is equal to speed added by accSpeed.
[2.1.4] when the car is in state accelerate and accSpeed is equal to 10 m/s2,
then its speed is equal to speed added by accSpeed.
[2.1.5] when the car receives Stop signal, then its speed is equal to 0 km/h.
[2.1.6] when the speed is greater than 100 km/h, then its speed is equal to
100 km/h.

NuSMV code:
VAR speed: 0..100;
        ASSIGN
          init(speed):= 0;--[2, 1, 1]
          next(speed):=
            case
                speed>100:100;--[2, 1, 6]
                action = Stop:0;--[2, 1, 5]
                state = accelerate & accSpeed = 10:speed+accSpeed;--[2, 1, 4]
                state = accelerate & accSpeed = 5:speed+accSpeed;--[2, 1, 3]
                TRUE: speed;
            esac;
```

Error 1 explicitly says “cannot assign value 105 to variable speed”. However, we do impose a range constraint “speed>100:100;” on this variable. We were confused about this error at the beginning. After numerous attempts, we realized that we could choose a suitable range constraint so that every branch of variable “speed” (every case of the variable in the ASSIGN module) is allowed to pass and the final value of each pass must be within bounds. For example, we could use a range constraint “speed>90:100;”. This way of giving the range constraint complicates a lot the MoSt modeling. We then found another way to deal with the issue, that is, we have to add a range constraint at each branch of this variable so that each branch is allowed to be passed and the final value will be within bounds. The final code can be like this:

```
NuSMV code:
VAR speed: 0..100;
        ASSIGN
          init(speed):= 0;--[2, 1, 1]
          next(speed):=
            case
                speed <= 100 & action = Stop:0;--[2, 1, 5]
                speed <= 90 & state = accelerate & accSpeed = 10:
                speed+accSpeed;--[2, 1, 4]
                speed <= 95 & state = accelerate & accSpeed = 5:
                speed+accSpeed;--[2, 1, 3]
                TRUE: speed;
            esac;
```

This way is much easier and more acceptable because it is not difficult to provide a constraint allowing one single branch safely to obtain its maximum value. On the other hand, NuSMV does make our requirements safer. In fact, “VAR speed: 0..100;” works like an invariant. This invariant enforces us to provide a range constraint to make each branch safe. Thus, providing range constraints for branches of variables should be one of the rules of writing the MoSt model. Note that this kind of variable scope errors is just involved with variables with arithmetic operations.

As a result, we find all the injected errors and one unexpected error and successfully correct them. This step checks further the correctness of the MoSt model.

Completeness analysis. As for completeness analysis, additional specifications for the reachability of each state are generated automatically from the model transformation, which are shown in Listing 2.

Listing 2. The Automatically Generated Property Specifications for the Car Example

```
SPEC EF state=parking
SPEC EF state=autonomy
SPEC EF state=start
SPEC EF state=accelerate
SPEC EF state=ignition
```

To demonstrate the feasibility of the completeness analysis, we voluntarily deleted 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 states *accelerate* and *autonomy*. The error and the counter-example are shown in Fig. 15 regarding the CTL specification $SPEC\ EF\ state\ =\ autonomy$. This CTL specification cannot 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. Therefore, the completeness analysis can check the completeness of the state transitions.

Validation analysis. Validation analysis verifies the satisfiability of property requirements. The experiment shows property requirements [7.1] and [7.2] are violated. Let us take requirement [7.2] as an example. This requirement says, "*when all globally the car is in state autonomy and it is in mode economic, then all future 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 $SPEC\ AG((state = autonomy\&mode = economic) -> AF(state! = autonomy))$. The verification result of this property specification is false, shown in Fig. 16. To make it more clear and understandable, Table 8 showing the state transitions is provided. Note that '/' means the content is empty; '-' denotes the content does change regarding that of the latest previous step; the "step" in Italic implies the steps will be executed sequentially and iteratively. So, when the system is in state 2.10, the preconditions are satisfied - the car is in *autonomy* state and its mode is *economic*, the system will always stay in *autonomy* state. It is because the mode of the system will be changed in each step of the loop. The counter-example implies when the car is in *autonomy* state and we change the working mode of the car into *economic*, it is possible that it will always stay in *autonomy* state.

Requirement [7.2] is designed to investigate the reaction of the system when it is in *autonomy* state and somehow switches to *economic* mode. Fig. 12 indicates that the car has to switch to *sportive* mode to reach *autonomy* state. So, this requirement checks whether the car can return to *accelerate* state when it is in *autonomy* state and drivers switch the mode to *economic*. This experiment shows that requirement [7.2] may not be satisfied when the car is delivered. Based on this analysis result, the car requirements engineers must put forward solutions to sort it out. However, it is also possible that sometimes the "facts" shown in the counter-example of Fig. 16 do not correspond to reality but it is what the customers want to have. In this case, requirements engineers can negotiate with customers to either modify the requirements or delete it. Even though the customers do not necessarily understand the counterexample in this figure, they do understand this property specification cannot pass in the simulation, to some extent. This can serve as strong evidence why requirements engineers ask them to at least modify it.

Table 8. The States Transitions Shown in the Counter-example

Step	**Input**	**State**	***mode***	***accSpeed***
state 2.1	/	parking	sportive	0
state 2.2	PowerUp	ignition	-	10
state 2.3	Start	start	-	-
state 2.4	Acc	accelerate	-	-
state 2.5	DeAC	-	-	-
state 2.6	-	-	-	-
state 2.7	-	-	-	-
state 2.8	-	-	-	-
state 2.9	Auto	autonomy	-	-
state 2.10	DeAC	-	economic	-
state 2.11	Ac	-	sportive	5
state 2.12	DeAc	-	economic	10

6.2 A Washing Machine Example

6.2.1 System description. In this case study, we take a SAMSUNG washing machine as an example, since washing machines naturally have modes such as *BébéCotton*. Fig. 17 provides a state machine of the washing machine of model WW70J3483KW/EF[7]. All the states, the modes, and the transitions are inspired by the manual of the washing machine. This system has fifteen states including *Idle, Weighting, Locking, Filling Water, Heating Water, HT Pre-washing, MT-Pre-washing, HT Washing, MT Washing, NT Washing, Draining, Spinning, HS Spinning, MS Spinning, Unlocking*. This system has five modes including *BébéCotton, Sport, Express, Jeans, Wool*. The modes involve three attributes: *preWash, spinningSpeed, temperature*. According to the manual of this machine, the impacts of modes on the values of attributions are shown in Table 9. Note that *HT, MT, NT, HS, MS* mean High Temperature, Medium Temperature, Normal Temperature, High Speed, Medium Speed, respectively.

Table 9. The Relationship between Modes and Attributes

mode	**preWash**	**spinningSpeed (r/m)**	**temperature (°C)**
BébéCotton	TRUE	1400	95
Sport	TRUE	800	40
Express	FALSE	800	-
Jeans	FALSE	800	40
Wool	FALSE	800	40

6.2.2 Requirements formalization & Static checking. In this section, we just illustrate a few examples of state transitions, attribute declarations, mode transitions, and property declarations. We also show the results of static checks of the washing machine requirements. The general procedures for requirement formalization can be seen in Section 6.1.2.

The example of the MoSt specification is listed as follows:

State transition :

[7]https://downloadcenter.samsung.com/content/UM/202008/20200813102835309/WW70J3467KW_DC68-03589F-07_EF.pdf (accessed in April 2023)

[1.1] when the machine is in state idle and it receives Put signal, then it will be in state weighting.

Attribute declaration :

[2.1.1] The temperature should be initialised to 0 °C.

[2.1.2] The temperature should be greater or equal to 0 and less or equal to 100 °C.

[2.1.3] when the machine is in mode bebeCotton and its temperature is less or equal to 94 °C, then its temperature is equal to temperature added by 1.

Mode transition :

[5.1] when the machine is in mode express and it receives ActJeans signal, then it is in mode jeans.

Property declaration :

[6.1] when all globally the machine is state hTPrewashing and it is in mode bebeCotton, then all next it is not in state hTPrewashing.

Nine errors (two naming check and seven consistency check errors) were injected into the requirements for static checks. All the errors have been identified by the validator. The results for static checks are shown in Fig. 18.

6.2.3 Requirements dynamic checking.

Traceability analysis. As discussed in Section 6.1.3, we still manually injected three kinds of errors (Errors 2-4, shown in Fig. 19) in the model to illustrate the feasibility of the traceability analysis. Similar results appeared in the editor. We finally corrected all the errors and the details are shown in Table 10. In terms of Error 1, the details of the correction are shown in requirements [2.1.41] - [2.1.43] in Annex C.

Table 10. The Details of Four Trace Links for the Washing Machine Example

No.	Error	Information	Correction
1	scope of variable "temperature"	variable: temperature	add range constraints for each branch
2	division by zero	ID: [4.3]	'0' − > 'x' (x! =0)
3	illegal types of "case" list elements: boolean and integer	variable: preWash	'5' − > 'TRUE' or 'FALSE'
4	illegal operand types of "+": integer and boolean	variable: temperature	'preWash' − > i (variable of integer)

Reachability analysis. The specifications for the reachability of the system states are automatically generated by the model transformation, which are listed in Listing 3. Here, we deliberately deleted requirement [1.17] ("*[1.17] when the machine is in state draining and it receives Spin signal, then it will be in state spinning.*"). This requirement tells the state transition between *draining* and *spinning*. The error and the counter-example are shown in Fig. 20 regarding the CTL specification *SPEC EF state=hSSpinning*. This property specification is false, which means the washing machine can never reach state *hSSpinning* (high-speed spinning). The missing of the state transition from states *Draining* to *Spinning* has been successfully detected. Therefore, the reachability of states can be ensured by the automatic reachability analysis.

Listing 3. The Automatically Generated Property Specifications for the Washing Machine Example

```
SPEC EF state=mSSpinning
SPEC EF state=mTPrewashing
SPEC EF state=unlocking
SPEC EF state=nTWashing
SPEC EF state=hTWashing
SPEC EF state=filling
SPEC EF state=draining
SPEC EF state=heating
SPEC EF state=idle
SPEC EF state=hTPrewashing
SPEC EF state=locking
SPEC EF state=weighting
SPEC EF state=hSSpinning
SPEC EF state=mTWashing
SPEC EF state=spinning
```

Validation analysis. Validation analysis aims to confirm whether the property requirements are achievable or not. The verification results show requirements [6.1] and [6.5] are violated, as shown in Figs. 21 and 22. Requirements [6.1] and [6.5] are: "*[6.1] when all globally the machine is in state hTPrewashing and it is in mode bebeCotton, then all next it is not in state hTPrewashing.*" and "*[6.5] when all globally the machine is in state mSSpinning and it is in mode jeans, then all next it is not in state mSSpinning.*", respectively. These two requirements utilize *Reductio ad absurdum* to prove that the washing machine can reach states *hTPrewashing* and *mSSpinning*, respectively. They are automatically translated into the CTL specifications *AG ((state = hTPrewashing & mode = bebeCotton) -> AX state != hTPrewashing)* and *AG ((state = mSSpinning & mode = jeans) -> AX state != mSSpinning)*, respectively. Since the two requirements are similar, we just analyze the counter-example of requirement [6.1] as an example. Table 11 shows the details of the counter-example. It implies the model takes 101 steps to reach state *hTPrewashing* from state *idle* under mode *bebeCotton*. Note that steps 1.4 - 1.97 are dedicated to heating water. So, we can conclude that when the washing machine is in the "High-Temperature Prewashing" state and we reset its mode to *bebeCotton*, it will stay in that state.

Table 11. The States Transitions Shown in the Counter-example for Requirement [6.1]

Step	**Input**	**State**	***mode***	***temperature***	***preWash***	***spinningSpeed***
state 1.1	/	idle	express	0	FALSE	0
state 1.2	Put	weighting	-	-	-	-
state 1.3	ActBebeCotton	-	bebeCotton	-	-	0
state 1.4	Heat	-	-	1	TRUE	0
...	-	-	-	...	-	-
state 1.97	-	-	-	94	-	-
state 1.98	Start	locking	-	-	-	-
state 1.99	Fill	filling	-	-	-	-
state 1.100	Heat	heating	-	95	-	-
state 1.101	Prewash	hTPrewashing	-	-	-	-
state 1.102	Start	-	-	-	FALSE	-

6.3 Discussion

In this paper, a "new" and "old" viewpoint, states and modes, has been introduced to requirements modeling and verification. The viewpoint is new because no one really conducts requirements analysis from this viewpoint. Since the terms of states and modes have been used in the industry, this point of view is also old. To make our approach clear, we propose an innovative approach to express 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, 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. This hypothesis is reasonable because they indeed exist. The requirements mentioned in many papers [1, 21, 29, 32] provide accurate information of the systems. For example, the range limits of every variable are clearly given. In addition, our DSL seems to be too restrictive because only two templates are supported. In fact, the two templates are general. They can express the characteristics of all the EARS templates. The precise definitions of rules provide precise signification of MoSt sentences. Two case studies have been used to illustrate the feasibility of

our DSL. For relaxing the grammar of MoSt, other grammatical frameworks could be helpful. One of the most cited frameworks is Grammatical Framework (GF) [35]. It is a special-purpose functional programming language for developing grammars. Its Resource Grammar Library (RGL) considers morphological and syntactic aspects of more than 30 natural languages. It naturally supports the complexities found in different natural languages, such as word inflection and word agreement. Using this framework, one can develop a special-purpose controlled natural language with a higher degree of flexibility. However, we would not choose GF. The reasons why we chose Xtext framework are: 1), GF is focused on multilingual grammar development and language translation, which is not our major concern; 2), GF generates abstract syntax trees from grammars and can be used for both parsing and generation of sentences in natural language; while Xtext framework covers all aspects of a complete language infrastructure, starting from the parser, code generator, up to a complete Eclipse IDE integration. It only needs a grammar specification. Xtext comes with better and smart default implementations for developing a DSL.

Secondly, algorithms have been designed to realize the automatic model transformation. We choose to use the NuSMV model checker 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 3.2. In addition, since the NuSMV model is not a hybrid automaton, our approach does not support continuous variables.

The last step of our approach is involved with requirements verification. We implemented a validator to statically check requirements. This validator is a user-defined static verification tool, which helps not only checking the naming rules for variables but also checking their 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 us take consistency check CC6 in Fig. 13 as an example, this tool can detect 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.

The dynamic checks are equally interesting. Thanks to the NuSMV runner, we are able to check the correctness of the requirements and the satisfiability of the user-defined and automatically generated requirements. The problems or counter-examples on requirements will pop up immediately, which greatly improves the efficiency of requirements verification. Note that the counter-examples concerning property specifications are not necessarily easily understandable for users without formal verification experiences but at least they will know these specifications are not ensured.

Our approach facilitates the validation process. The validation analysis mentioned in Section 5.3.3 improves the safety of the system. It is very helpful in analyzing the impacts of the newly added modes on the boundaries of the system, especially when the system is of great complexity and intelligence. Certainly, sometimes the counter-examples are unrealistic because the NuSMV model checker tries all the possible runs of the model. The analysis of the counter-examples is possibly limited to experts in formal verification. We hope our work is able to provide new ideas to designers who really work in the safety-critical domains.

Scalability concerns need to be addressed as well since we used model checking and Xtext framework. Table 12 shows the results on scalability. According to this table, if we double NR, LOC and ATMT will be approximately doubled. ATSV and ATDV are all significantly influenced by the increased number of requirements. For ATSV, it is because the MoSt validator always provokes all the checks when the code is saved. So, the Xtext framework we used should be optimized to provoke only the relevant checks. For ATDV, it is because the exploration state space is dramatically increased. So, the model checker we used should be also improved to optimize the exploration trace.

Table 12. Scalability analysis on the two cases regarding number of requirements (NR), number of states (NS), number of modes (NM), lines of code on derived NuSMV models (LOC), average time for model transformation (ATMT), average time for static verification in seconds (ATSV), and average time for dynamic verification in seconds (ATDV)

	NR	**NS**	**NM**	**LOC**	**ATMT**	**ATSV**	**ATDV**
Car	37	5	2	88	0,003	0.042	0,032
Washing machine	64	15	5	213	0,005	13,371	13,467

Finally, careful readers may notice that the generated NuSMV is similar to the MoSt model and then ask why not we directly use NuSMV. Here are our answers:

(1) The MoSt model is a controlled natural language for modeling the requirements based on states and modes, which is easy to read and write; While the NuSMV does not include the notion of modes; it does not provide a mechanism to build the relationship between modes and states; it does not support the modeling in controlled natural languages;
(2) The MoSt modeling tool automatically supports static and dynamic checks, which assists users to write correct requirements;
(3) MoSt eases the burden of requirements modeling and management because of reasons 2 and 3;
(4) MoSt eases the work of validation, to some extent, since the discussion of requirements and the product can remain at the level of natural language models; it ensures the safety of the requirements from the states-and-modes point of view;
(5) The MoSt platform provides a solution for users who are not experts in formal verification to modeling requirements based on states and modes.

This could be a quickly feasible solution for experts in formal verification as well. Because even though an expert in formal verification wants to use NuSMV to construct MoSt-like models, it could take time. The idea of designing models using NuSMV could be different. For example, we define mode as an array *VAR mode:{economic, sportive}*. Then we use *mode = economic* to represent the system of interest stays in *economic* mode. Others could define modes as *Var economicMode, sportiveMode: boolean*. Then they could use *economicMode = true* to say the system of interest stays in *economic* mode. The representations of the model will be changed as well. To some extent, we provide a "standard" way to model states-and-modes requirements using NuSMV.

7 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 clear relationship between states and modes implies that the impacts of the activation of one mode on system behaviors can be investigated. So, we will be still capable of controlling system behaviors when additional modes are added.

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 first proposed to lay the foundation for our requirements analysis strategy. A DSL called MoSt was designed for requirements modeling and implemented by the Xtext framework. We provided this DSL with the meta-model and the grammar. A model validator was then implemented to statically check the requirements against the static rules. Afterwards, a mapping of components between the MoSt model and NuSMV model was proposed. The algorithms for realizing the mapping were designed and implemented in a code generator. Since then, we were capable of automatically obtaining a NuSMV model from the MoSt modeling tool. To make an automatic process for running the NuSMV model checker and analyzing the simulation results, a NuSMV runner was implemented to achieve a seamless connection between NuSMV and our tool. The results analyzed were sent to the validator to make them visible

in the editor. Finally, a powerful and easy-to-use 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 extracting 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 much 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.

ACKNOWLEDGMENTS

This work was supported by the Excellence Laboratory "International Centre for Mathematics and Computer Science in Toulouse" (CIMI Labex). The authors would also like to thank their colleague Marc Pantel and all the team from SAMARES Engineering for interesting discussions about states and modes.

REFERENCES

[1] Manzoor Ahmad, 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 (2015), 50–70.
[2] Raian Ali, Fabiano Dalpiaz, and Paolo Giorgini. 2013. Reasoning with contextual requirements: Detecting inconsistency and conflicts. *Information and Software Technology* 55, 1 (2013), 35–57.
[3] NAUMENKO Andrey. 2002. *A paradigm for General System Modeling and its applications for UML and RM-ODP.* Ph. D. Dissertation. Ph. D thesis.
[4] Julia Badger, David Throop, and Charles Claunch. 2014. Vared: verification and analysis of requirements and early designs. In *2014 IEEE 22nd International Requirements Engineering Conference (RE)*. IEEE, 325–326.
[5] Ronan Baduel. 2019. *An integrated model-based early validation approach for railway systems.* Ph. D. Dissertation. Toulouse 2.
[6] Lorenzo Bettini. 2016. *Implementing domain-specific languages with Xtext and Xtend.* Packt Publishing Ltd.
[7] Stéphane Bonnet, 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. Wiley Online Library, 548–562.
[8] Dennis M Buede and William D Miller. 2016. *The engineering design of systems: models and methods.* John Wiley & Sons.
[9] Gustavo Carvalho, Ana Cavalcanti, and Augusto Sampaio. 2016. Modelling timed reactive systems from natural-language requirements. *Formal Aspects of Computing* 28 (2016), 725–765.
[10] Roberto Cavada, Alessandro Cimatti, Gavin Keighren, Emanuele Olivetti, Marco Pistor, and Marco Roveri. 2019. NuSMV 2.5 Tutorial. http://nusmv.fbk.eu/NuSMV/tutorial/v25/tutorial.pdf
[11] Alessandro Cimatti, 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*. Springer, 359–364. http://nusmv.fbk.eu/
[12] Alessandro Cimatti, 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*. Springer, 359–364. http://nusmv.fbk.eu/
[13] IEEE Standards Coordinating Committee et al. 1990. IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990). Los Alamitos. *CA: IEEE Computer Society* 169 (1990).
[14] D Davis et al. 2005. SMC Systems Engineering Primer & Handbook. *United States Air Force Space & Missile Systems Center* (2005), 13–17.
[15] 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
[16] DI-IPSC-81431A. 2000. MIL-STD-498B (Cancelled) Data Item Description, System/Subsystem Specification.
[17] DMO. 2011. Defence Materiel Organisation, DMH (ENG) 12-3-005 Function and Performance (FPS) Development Guide.
[18] MT Edwards. 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.*
[19] Omar El Beggar, Khadija Letrache, and Mohammed Ramdani. 2020. DAREF: MDA framework for modelling data warehouse requirements and deducing the multidimensional schema. *Requirements Engineering* (2020), 1–23.
[20] Peter H Feiler, 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*. IEEE, 1206–1211.

[21] Dimitra Giannakopoulou, Thomas Pressburger, Anastasia Mavridou, and Johann Schumann. 2021. Automated formalization of structured natural language requirements. *Information and Software Technology* (2021), 106590.
[22] Heather J Goldsby, 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)*. IEEE, 36–45.
[23] David Harel. 1987. Statecharts: A visual formalism for complex systems. *Science of computer programming* 8, 3 (1987), 231–274.
[24] Constance L Heitmeyer, Ralph D Jeffords, and Bruce G Labaw. 1996. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 5, 3 (1996), 231–261.
[25] C. A. R. Hoare. 1971. Procedures and parameters: An axiomatic approach. In *Symposium on Semantics of Algorithmic Languages*, Erwin Engeler (Ed.). Lecture Notes in Mathematics, Vol. 188. Springer, 102–116. https://doi.org/10.1007/BFb0059696
[26] IEEE Standards Association IEEE 24765 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* (2010).
[27] J. Jenney. 2011. Define Life Cycle System Modes. http://themanagersguide.blogspot.com/2011/01/6322-define-life-cycle-system-modes.html accessed on 25/11/2020.
[28] Emmanuel Letier 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.
[29] Nancy G Leveson, Mats Per Erik Heimdahl, Holly Hildreth, and Jon Damon Reese. 1994. Requirements specification for process-control systems. *IEEE transactions on software engineering* 20, 9 (1994), 684–707.
[30] Yinling Liu and Jean-Michel Bruel. 2022. Modeling of Natural Language Requirements based on States and Modes. In *2022 IEEE 30th International Requirements Engineering Conference Workshops (REW)*. IEEE, 190–194.
[31] Alistair Mavin, Philip Wilkinson, Adrian Harwood, and Mark Novak. 2009. Easy approach to requirements syntax (EARS). In *2009 17th IEEE International Requirements Engineering Conference*. IEEE, 317–322.
[32] Abha Moitra, 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 (2019), 341–364.
[33] Soroosh Nalchigar, Eric Yu, and Karim Keshavjee. 2021. Modeling machine learning requirements from three perspectives: a case report from the healthcare domain. *Requirements Engineering* (2021), 1–18.
[34] Anthony M Olver 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*.
[35] Aarne Ranta. 2011. *Grammatical framework: Programming with multilingual grammars*. Vol. 173. CSLI Publications, Center for the Study of Language and Information Stanford.
[36] Winston W Royce. 1987. Managing the development of large software systems: concepts and techniques. In *Proceedings of the 9th international conference on Software Engineering*. 328–338.
[37] Vítor E Silva Souza, 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.
[38] Charles S Wasson. 2010. System Phases, Modes, and States Solutions to Controversial Issues. *Wasson Strategics, LLC. http://www.wassonstrategics. com* (2010).
[39] Charles S Wasson. 2015. *System engineering analysis, design, and development: Concepts, principles, and practices*. John Wiley & Sons.
[40] Jon Whittle, 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*. IEEE, 79–88.
[41] Eric SK Yu. 1997. Towards modelling and reasoning support for early-phase requirements engineering. In *Proceedings of ISRE'97: 3rd IEEE International Symposium on Requirements Engineering*. IEEE, 226–235.
[42] Leopoldo Zepeda, 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*. IEEE, 261–265.

A THE MOST MODEL FOR THE CAR EXAMPLE

```
[1.1]  when the car is in state parking and it receives PowerUp signal, then it will be
in state ignition.
[1.2]  when the car is in state ignition and it receives Start signal, then it will be
in state start.
[1.3]  when the car is in state start and it receives Acc signal and its accSpeed is
equal to 10 m/s2, then it will be in state accelerate.
[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.
```

[1.5] when the car is in state autonomy and it receives Manual signal and its accSpeed is equal to 10 m/s2, then it will be in state accelerate.
[1.6] when the car is in state start and it receives Acc signal and its accSpeed is equal to 5 m/s2, then it will be in state accelerate.
[1.7] when the car is in state accelerate and it receives Stop signal and its accSpeed is equal to 10 m/s2, then it will be in state start.
[1.8] when the car is in state accelerate and it receives Stop signal and its accSpeed is equal to 5 m/s2, then it will be in state parking.
[1.9] when the car is in state start and it receives Stop signal and its accSpeed is equal to 10 m/s2, then it will be in state ignition.
[1.10] when the car is in state start and it receives Stop signal and its accSpeed is equal to 5 m/s2, then it will be in state parking.
[1.11] when the car is in state ignition and it receives PowerOff signal, then it will be in state parking.
[2.1.1] The speed should be initialised to 0 km/h.
[2.1.2] The speed should be greater or equal to 0 less or equal to 100 km/h.
[2.1.3] when the car is in state accelerate and accSpeed is equal to 5 m/s2, then its speed is equal to speed added by accSpeed.
[2.1.4] when the car is in state accelerate and accSpeed is equal to 10 m/s2, then its speed is equal to speed added by accSpeed.
[2.1.5] when the car receives Stop signal, then its speed is equal to 0 km/h.
[2.1.6] when the speed is greater than 90 km/h, then its speed is equal to 100 km/h.
[2.2.1] The accSpeed should be initialised to 0 m/s2.
[2.2.2] The accSpeed should be greater or equal to 0 less or equal to 10 m/s2.
[2.2.3] when the car is mode economic, then its accSpeed is equal to 5 m/s2.
[2.2.4] when the car is mode sportive, then its accSpeed is equal to 10 m/s2.
[3.1] The doorIsOpen should be initialised to FALSE.
[3.2] when the car is not in state parking, then its doorIsOpen is equal to FALSE.
[3.3] when the car is in state parking, then its doorIsOpen is equal to TRUE.
[4.1] The stop should be initialised to FALSE.
[4.2] when the car receives Stop signal and it is in state accelerate, then its stop is equal to TRUE.
[4.3] when the car is in state ignition or it is in state start, then its stop is equal to FALSE.
[5.1] The displaySpeed should be initialised to FALSE.
[5.2] when accSpeed is equal to 5 m/s2 and its speed is greater than 80 km/h, then its displaySpeed is equal to TRUE.
[5.3] when accSpeed is equal to 5 m/s2 and its speed is less than 80 km/h, then its displaySpeed is equal to FALSE.
[5.4] when its accSpeed is equal to 10 m/s2, then its displaySpeed is equal to TRUE.
[6.1] when the car is in mode sportive and it receives DeAC signal, then it is in mode economic.
[6.2] when the car is in mode economic and it receives Ac signal, then it is in mode sportive.
[7.1] when all globally the car is state autonomy and it is in mode economic, then all future it is not in state accelerate.
[7.2] when all globally the car is state autonomy and it is in mode economic, then all future it is not in state autonomy.
[7.3] when all globally the car is state autonomy and it is in mode economic, then all future it is in state autonomy.
[8.1] Car should be as comfortable as possible.

B THE GENERATED NUSMV MODEL FOR THE CAR MOST MODEL

```
MODULE main
----------------Specification Definition--------------------
SPEC AG (( state = autonomy  &  mode = economic  ) -> AF (state != accelerate ))--[7, 1]
SPEC AG (( state = autonomy  &  mode = economic  ) -> AF (state != autonomy ))--[7, 2]
SPEC AG (( state = autonomy  &  mode = economic  ) -> AF (state = autonomy ))--[7, 3]
SPEC EF state=parking
SPEC EF state=autonomy
SPEC EF state=start
SPEC EF state=accelerate
SPEC EF state=ignition
----------------Specification Definition--------------------
VAR state:{parking , autonomy , start , accelerate , ignition };
VAR mode:{ economic ,  sportive };
IVAR action:{Acc, Auto, Ac, DeAC, Stop, Start, PowerUp, Manual, PowerOff};
VAR doorIsOpen: boolean;
VAR stop: boolean;
VAR accSpeed: 0..10;
VAR displaySpeed: boolean;
VAR speed: 0..100;
INIT (state =  parking)
  TRANS(next(state) =
    case
        state = parking  &  action = PowerUp:  ignition;--[1, 1]
        state = ignition  &  action = Start:  start;--[1, 2]
        state = start  &  action = Acc  &  accSpeed=10:  accelerate;--[1, 3]
        state = accelerate  &  action = Auto  &  accSpeed=10:  autonomy;--[1, 4]
        state = autonomy  &  action = Manual  &  accSpeed=10:  accelerate;--[1, 5]
        state = start  &  action = Acc  &  accSpeed=5:  accelerate;--[1, 6]
        state = accelerate  &  action = Stop  &  accSpeed=10:  start;--[1, 7]
        state = accelerate  &  action = Stop  &  accSpeed=5:  parking;--[1, 8]
        state = start  &  action = Stop  &  accSpeed=10:  ignition;--[1, 9]
        state = start  &  action = Stop  &  accSpeed=5:  parking;--[1, 10]
        state = ignition  &  action = PowerOff:  parking;--[1, 11]
        TRUE : state;
    esac)

      ASSIGN
        init(doorIsOpen):= FALSE;--[3, 1]
        next(doorIsOpen):=
          case
              state = parking:TRUE;--[3, 3]
              state != parking:FALSE;--[3, 2]
              TRUE: doorIsOpen;
          esac;
      ASSIGN
        init(stop):= FALSE;--[4, 1]
        next(stop):=
          case
```

```
        state = ignition | state = start:FALSE;--[4, 3]
        action = Stop & state = accelerate:TRUE;--[4, 2]
        TRUE: stop;
    esac;
ASSIGN
  init(accSpeed):= 0;--[2, 2, 1]
  next(accSpeed):=
    case
        mode = sportive:10;--[2, 2, 4]
        mode = economic:5;--[2, 2, 3]
        TRUE: accSpeed;
    esac;
ASSIGN
  init(displaySpeed):= FALSE;--[5, 1]
  next(displaySpeed):=
    case
        accSpeed = 10:TRUE;--[5, 4]
        accSpeed = 5 & speed<80:FALSE;--[5, 3]
        accSpeed= 5 & speed>80:TRUE;--[5, 2]
        TRUE: displaySpeed;
    esac;
ASSIGN
  init(speed):= 0;--[2, 1, 1]
  next(speed):=
    case
        speed>90:100;--[2, 1, 6]
        action = Stop:0;--[2, 1, 5]
        state = accelerate & accSpeed = 10:speed+accSpeed;--[2, 1, 4]
        state = accelerate & accSpeed = 5:speed+accSpeed;--[2, 1, 3]
        TRUE: speed;
    esac;

ASSIGN
init(mode):= sportive;
next(mode):=
  case
        mode = sportive  &  action = DeAC: economic;--[6, 1]
        mode = economic  &  action = Ac: sportive;--[6, 2]
        TRUE:mode;
  esac;
```

C THE MOST MODEL FOR THE WASHING MACHINE EXAMPLE

[1.1] when the machine is in state idle and it receives Put signal, then it will be in state weighting.
[1.2] when the machine is in state weighting and it receives Remove signal, then it will be in state weighting.
[1.3] when the machine is in state weighting and it receives Start signal, then it will be in state locking.
[1.4] when the machine is in state locking and it receives Fill signal, then it will be in state filling.
[1.5] when the machine is in state filling and it receives Heat signal, then it will be in state heating.
[1.6] when the machine is in state filling and it receives Wash signal, then it will be in state nTWashing.
[1.7] when the machine is in state heating and it receives Heat signal, then it will be in state heating.

[1.8] when the machine is in state heating and it receives Prewash signal and its temperature is equal to 95 °C,

```
then it will be in state hTPrewashing.
[1.9]   when the machine is in state heating and it receives Prewash signal and its temperature is equal to 40 °C,
then it will be in state mTPrewashing.
[1.10]  when the machine is in state heating and it receives Wash signal and its temperature is equal to 95 °C,
then it will be in state hTWashing.
[1.11]  when the machine is in state heating and it receives Wash signal and its temperature is equal to 40 °C,
then it will be in state mTWashing.

[1.12]  when the machine is in state hTPrewashing and it receives Drain signal and its preWash is equal to FALSE,
then it will be in state draining.
[1.13]  when the machine is in state mTPrewashing and it receives Drain signal and its preWash is equal to FALSE,
then it will be in state draining.

[1.14]  when the machine is in state hTWashing and it receives Drain signal, then it will be in state draining.
[1.15]  when the machine is in state mTWashing and it receives Drain signal, then it will be in state draining.
[1.16]  when the machine is in state nTWashing and it receives Drain signal, then it will be in state draining.

[1.17]  when the machine is in state draining and it receives Spin signal, then it will be in state spinning.
[1.18]  when the machine is in state draining and it receives Fill signal, then it will be in state filling.
[1.19]  when the machine is in state spinning and it receives Spin signal and its spinningSpeed is equal to 1400 r/m,
then it will be in state hSSpinning.
[1.21]  when the machine is in state spinning and it receives Spin signal and its spinningSpeed is equal to 800 r/m,
then it will be in state mSSpinning.

[1.22]  when the machine is in state hSSpinning and it receives Unlock signal, then it will be in state unlocking.
[1.23]  when the machine is in state mSSpinning and it receives Unlock signal, then it will be in state unlocking.

[1.24]  when the machine is in state unlocking and it receives Free signal, then it will be in state idle.

[2.1.1] The temperature should be initialised to 0 °C.
[2.1.2] The temperature should be greater or equal to 0 and less or equal to 100 °C.
[2.1.3] when the machine is in mode bebeCotton and its temperature is less or equal to 94 °C,
then its temperature is equal to temperature added by 1.
[2.1.41] when the machine is in mode jeans and its temperature is less or equal to 39 °C,
then its temperature is equal to temperature added by 1.
[2.1.42] when the machine is in mode wool and its temperature is less or equal to 39 °C,
then its temperature is equal to temperature added by 1.
[2.1.43] when the machine is in mode sport and its temperature is less or equal to 39 °C,
then its temperature is equal to temperature added by 1.

[3.1]   The preWash should be initialised to FALSE.
[3.2]   when the machine is in mode bebeCotton or it is mode sport, then its preWash is equal to TRUE.
[3.3]   when the machine is in mode express or it is in mode jeans or it is in mode wool, then its preWash is equal to FALSE.
[3.4]   when the machine is in state hTPrewashing or it is in state mTPrewashing, then its preWash is equal to FALSE.

[4.1]   The spinningSpeed should be initialised to 0 r/m.
[4.2]   The spinningSpeed should be greater or equal to 0 and less or equal to 2000 r/m.
[4.3]   when the machine is in mode bebeCotton and its spinningSpeed is less or equal to 1399 r/m,
then its spinningSpeed is equal to spinningSpeed added by 1.
[4.41]  when the machine is in mode jeans and its spinningSpeed is less or equal to 799 r/m,
then its spinningSpeed is equal to spinningSpeed added by 1.
[4.42]  when the machine is in mode wool and its spinningSpeed is less or equal to 799 r/m,
then its spinningSpeed is equal to spinningSpeed added by 1.
[4.43]  when the machine is in mode sport and its spinningSpeed is less or equal to 799 r/m,
then its spinningSpeed is equal to spinningSpeed added by 1.


[5.1]   when the machine is in mode express and it receives ActJeans signal, then it is in mode jeans.
[5.2]   when the machine is in mode express and it receives ActWool signal, then it is in mode wool.
[5.3]   when the machine is in mode express and it receives ActBebeCotton signal, then it is in mode bebeCotton.
[5.4]   when the machine is in mode express and it receives ActSport signal, then it is in mode sport.
```

```
[5.5]   when the machine is in mode jeans and it receives ActJeans signal, then it is in mode jeans.
[5.6]   when the machine is in mode jeans and it receives ActWool signal, then it is in mode wool.
[5.7]   when the machine is in mode jeans and it receives ActBebeCotton signal, then it is in mode bebeCotton.
[5.8]   when the machine is in mode jeans and it receives ActSport signal, then it is in mode sport.

[5.9]    when the machine is in mode bebeCotton and it receives ActJeans signal, then it is in mode jeans.
[5.10]  when the machine is in mode bebeCotton and it receives ActWool signal, then it is in mode wool.
[5.11]  when the machine is in mode bebeCotton and it receives ActBebeCotton signal, then it is in mode bebeCotton.
[5.12]  when the machine is in mode bebeCotton and it receives ActSport signal, then it is in mode sport.

[5.13]  when the machine is in mode sport and it receives ActJeans signal, then it is in mode jeans.
[5.14]  when the machine is in mode sport and it receives ActWool signal, then it is in mode wool.
[5.15]  when the machine is in mode sport and it receives ActBebeCotton signal, then it is in mode bebeCotton.
[5.16]  when the machine is in mode sport and it receives ActSport signal, then it is in mode sport.

[5.17]  when the machine is in mode wool and it receives ActJeans signal, then it is in mode jeans.
[5.18]  when the machine is in mode wool and it receives ActWool signal, then it is in mode wool.
[5.19]  when the machine is in mode wool and it receives ActBebeCotton signal, then it is in mode bebeCotton.
[5.20]  when the machine is in mode wool and it receives ActSport signal, then it is in mode sport.

[6.1]   when all globally the machine is state hTPrewashing and it is in mode bebeCotton,
then all next it is not in state hTPrewashing.
[6.2]   when all globally the machine is state hTPrewashing and it is in mode express,
then exist future it is in state nTWashing.
[6.3]   when all globally the machine is state nTWashing and it is in mode bebeCotton,
then exist future it is in state hTWashing.
[6.4]   when all globally the machine is state nTWashing and it is in mode bebeCotton,
then exist future it is in state hTPrewashing.
[6.5]   when all globally the machine is state mSSpinning and it is in mode jeans,
then all next it is not in state mSSpinning.
```

D MOST GRAMMAR

Naming Check (NC)

NC1:

```
[1.1]  when the car is in state Parking and it receives PowerUp signal, then it will be in state ignition.
```
State name should start with a lower case: error 'P'
Press 'F2' for focus

NC2:

```
[1.1]  when the car is in state parking and it receives powerUp signal, then it will be in state ignition.
```
Signal name should start with a upper case: error 'p'
Press 'F2' for focus

Consistency Check (CC)

CC1:

```
[3.1]   The doorIsOpen should be initialised to FALSE.
```
Variables should only be initialized once! 'doorIsOpen'
Press 'F2' for focus
```
[3.11]   The doorIsOpen should be initialised to TRUE.
```

CC2:

```
[2.2.3] when the car is mode economic, then its accSpeed is equal to 5 m/s2.
```
You have not initialised this variable
Press 'F2' for focus

CC3:

```
[2.2.1] The accSpeed should be initialised to 0 m/s2.
```
Scope should be given to environment variables 'accSpeed'
Press 'F2' for focus

CC4:

```
[1.1]  when the car is in state parking and it receives PowerUp signal, then it will be in state ignition.
[1.1]  when the car is in state parking and it receives PowerUp signal, then it will be in state ignition.
```
ID can not be repeated '[1, 1]'
Press 'F2' for focus

CC5:

```
[1.1]   when the car is in state parking and it receives PowerUp signal, then it will be in state ignition.
[1.1.1] when the car is in state parking and it receives PowerUp signal, then it will be in state ignition.
```
You have written the same state requirements.
Press 'F2' for focus

CC6 :

```
[1.11]  when the car is in state ignition and it receives PowerOff signal, then it will in state parking.
[1.12]  when the car is in state ignition and it receives PowerOff signal, then it will in state start.
```
You have written different state postconditions with the same preconditions and guards
Press 'F2' for focus

CC7:

```
[1.3]   when the car is in state start and it receives Acc signal and its accSpeed1 is equal to 10 km/h, then it will be in state accelerate.
```
The requirement of attribute accSpeed1 is missing.
Press 'F2' for focus

Fig. 13. Results of Car Requirements Static Checks

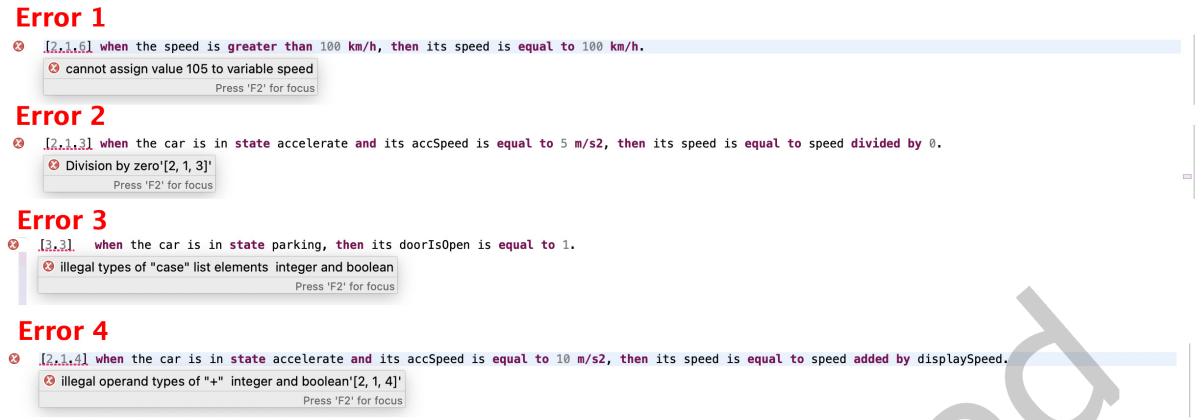

Fig. 14. The Screenshots of Errors Detected by Traceability Analysis for the Car Example

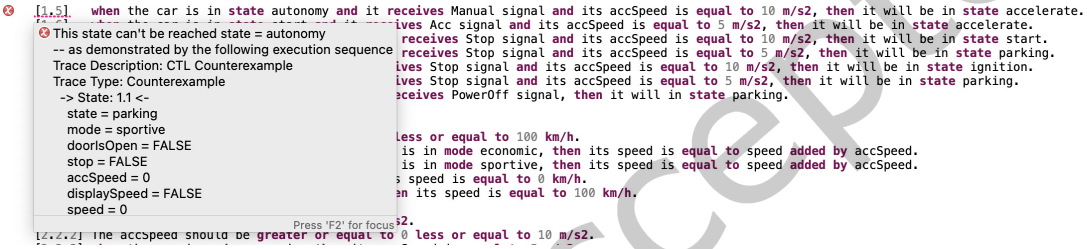

Fig. 15. Verification Result for Requirement [1.5]

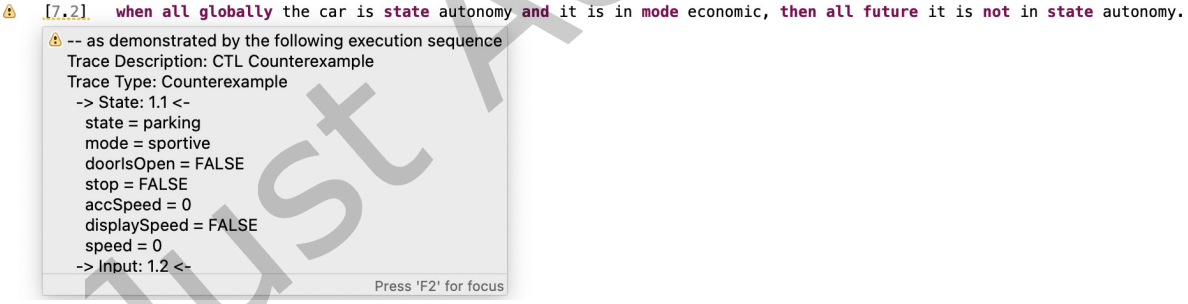

Fig. 16. Verification Result for Requirement [7.2]

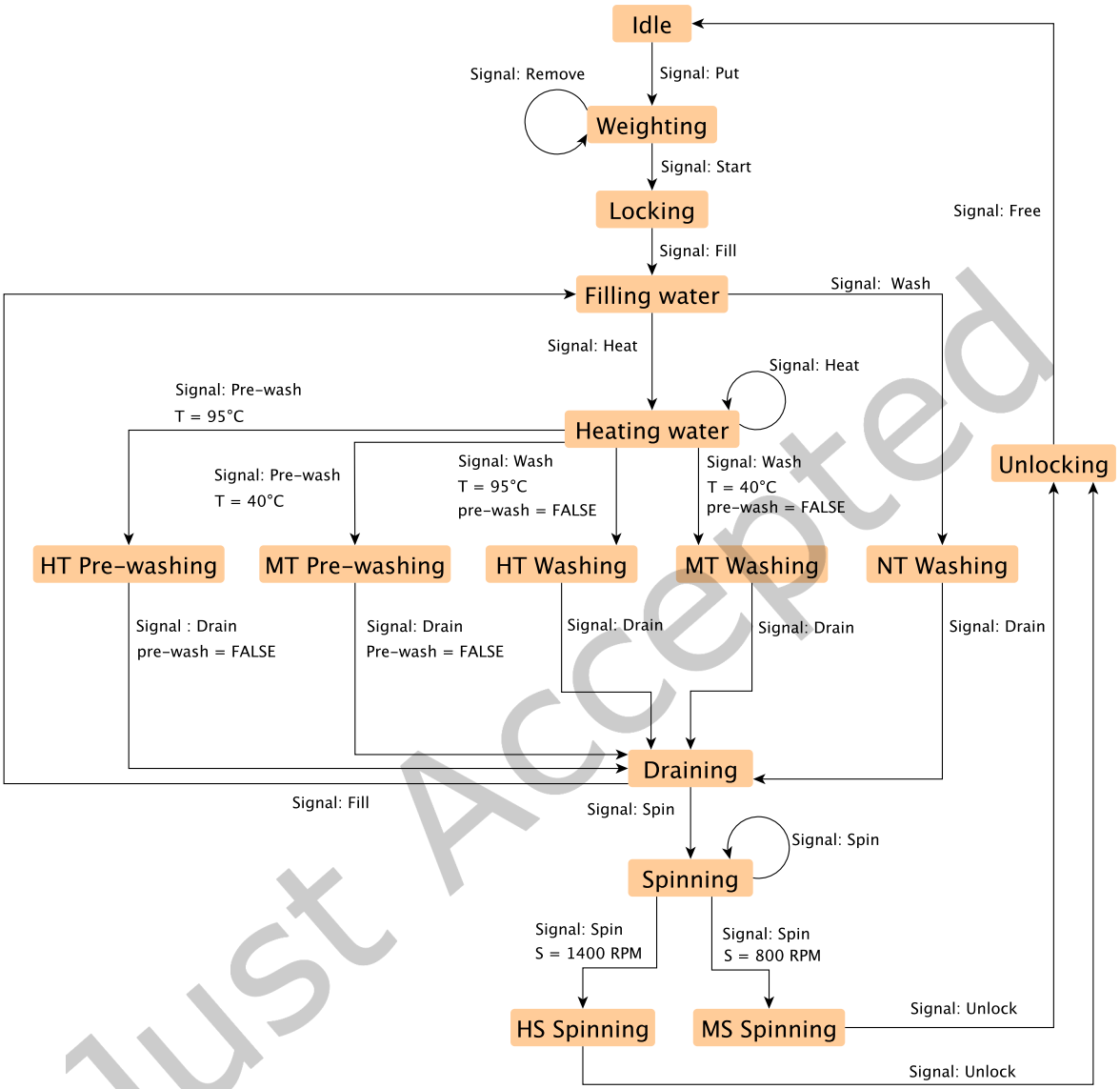

Fig. 17. The State Machine Diagram of a Washing Machine

Naming Check

NC1:

```
[1.1]  when the machine is in state Idle and it receives Put signal, then it will be in state weighting.
```
State name should start with a lower case: error 'I'
Press 'F2' for focus

NC2:

```
[1.1]  when the machine is in state idle and it receives put signal, then it will be in state weighting.
```
Signal name should start with a upper case: error 'p'
Press 'F2' for focus

Consistency Check

CC1:

```
[3.1]  The preWash should be initialised to FALSE.
[3.11]  The preWash should be initialised to TRUE.
```
Non-integer variables should only be initialised once! 'preWash'
Press 'F2' for focus

CC2:

```
[2.1.3] when the machine is in mode bebeCotton and its temperature is less or equal to 94 °C, then its temperature is equal to temperature added by 1.
```
You have not initialised this variable
Press 'F2' for focus

CC3:

```
[2.1.1] The temperature should be initialised to 0 °C.
```
Scope should be given to environment variables 'temperature'
Press 'F2' for focus

CC4:

```
[1.1]  when the machine is in state idle and it receives Put signal, then it will be in state weighting.
[1.1]  when the machine is in state idle and it receives Put signal, then it will be in state weighting.
```
ID can not be repeated '[1, 1]'
Press 'F2' for focus

CC5:

```
[1.1]  when the machine is in state idle and it receives Put signal, then it will be in state weighting.
[1.11]  when the machine is in state idle and it receives Put signal, then it will be in state weighting.
```
You have written the same state requirements.
Press 'F2' for focus

CC6:

```
[1.1]  when the machine is in state idle and it receives Put signal, then it will be in state weighting.
[1.111]  when the machine is in state idle and it receives Put signal, then it will be in state locking.
```
You have written different state postconditions with the same preconditions and guards.
Press 'F2' for focus

CC7:

```
[1.8]  when the machine is in state heating and it receives Prewash signal and its temperature1 is equal to 95, then it will be in state hTPrewashing.
```
The requirement of attribute temperature1 is missing.
Press 'F2' for focus

Fig. 18. Results of Washing Machine Requirements Static Checks

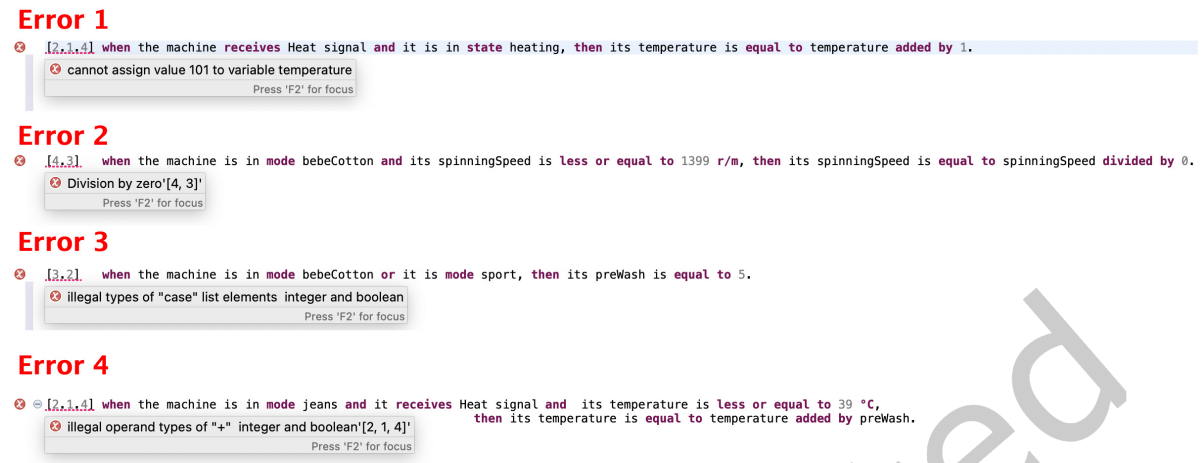

Fig. 19. The Screenshots of Errors Detected by the Traceability Analysis for the Washing Machine

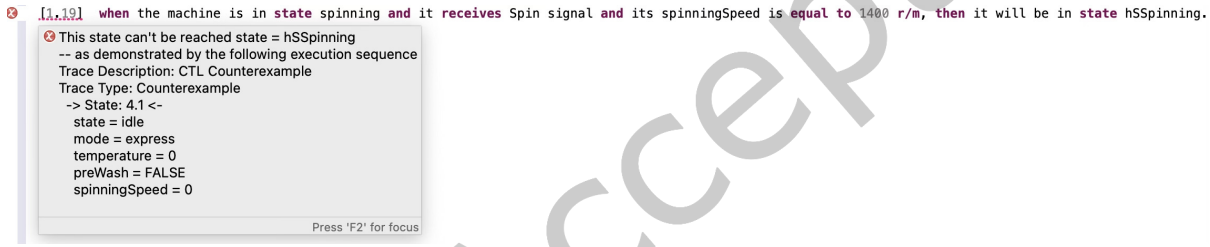

Fig. 20. Verification Result for Requirement [1.19]

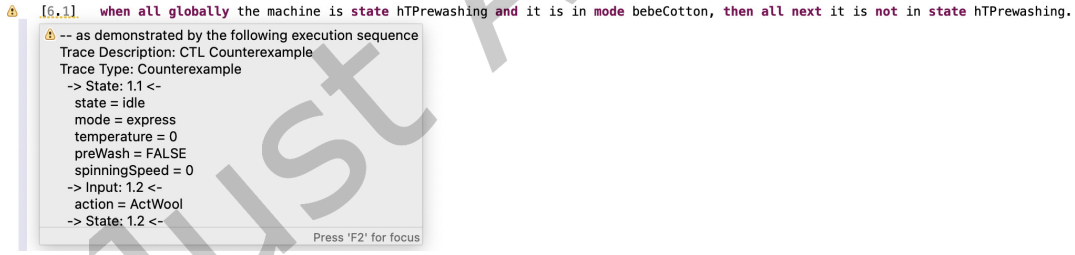

Fig. 21. Verification Result for Requirement [6.1]

[6.5] **when all globally** the machine is **state** mSSpinning **and** it is in **mode** jeans, **then all next** it is **not** in **state** mSSpinning.

```
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
 -> State: 2.1 <-
   state = idle
   mode = express
   temperature = 0
   preWash = FALSE
   spinningSpeed = 0
 -> Input: 2.2 <-
   action = ActWool
 -> State: 2.2 <-
```

Press 'F2' for focus

Fig. 22. Verification Result for Requirement [6.5]

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

3.  Requirement:
4.  MODE | STATE | ATTRIBUTECONSTRAINT | PROPERTYCONSTRAINT | 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' preModeCondition=MODECONDITION
11. (relations+=RELATION guards+=SIGNALCONDITION)* ',' 'then'
12. postModeCondition = MODECONDITION '.';

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

17. ATTRIBUTECONSTRAINT:
18. attributeReqID=ReqID 'when' preAttributeCondition=(STATECONDITON
19. |ATTRIBUTECONTION | SIGNALCONDITION)  (relations+= RELATION
20. guards+=(STATECONDITON|ATTRIBUTECONTION| SIGNALCONDITION))*',' 'then'
21. postAttributeCondition = (ATTRIBUTECONTION | ARITHMETICCONDITION)'.';

22. PROPERTYCONSTRAINT:
23. propertyReqID=ReqID 'when' preOperator= (CTLOperator | LTLOperator)
24. prePropertyCondition=(STATECONDITON | ATTRIBUTECONTION |
25. MODECONDITION )  (preRelations+=RELATION guards +=(STATECONDITON |
26. ATTRIBUTECONTION | MODECONDITION))* ',' 'then'
27. postOperator=(CTLOperator | LTLOperator) postPropertyConditions
28. =(STATECONDITON | ATTRIBUTECONTION | MODECONDITION )
29. (postRelations+=RELATION postPropertyConditions +=(STATECONDITON | ATTRIBUTECONTION |
30. MODECONDITION))* '.';
```

Fig. 23. The Grammar of the Types of Formal Requirements

```
STATECONDITON:
((ID (ID)* 'state' stateName=ID) | ((ID)* compOperator=COMPARISONOPERATOR) (ID)*
'state' stateName=ID);

MODECONDITION:
ID (ID)* 'mode' modeName=ID;

SIGNALCONDITION:
ID (ID)* 'receives' signalName=ID ID;
ReqID: '[' reqID+=INT ('.'reqID+=INT)* ']';

ATTRIBUTECONTION:
ID (ID)* attributeName=ID ID operator=COMPARISONOPERATOR attributeValue =
ATTRIBUTEVALUE (unit+=UNIT)*;

ARITHMETICCONDITION:
ID result=ID (ID)* compcondition=COMPARISONOPERATOR var1=ID arithmeticOperator
= ARITHMETICOPERATOR (var2=ID | var3=INT);
```

Fig. 24. The Grammar of the Conditions of Formal Requirements

```
ARITHMETICOPERATOR:
(ADD | SUBTRACTION | MULTIPLICATION | DIVISION | MODULE);

MODULE: add= 'moduled' 'divided';
DIVISION: division='divided' 'by';
MULTIPLICATION:multiplication='multiplied' 'by';
SUBTRACTION: substraction='subtracted' 'by';
ADD: add='added' 'by';

COMPARISONOPERATOR:
(EQUAL | LESS | GREATER | NOTEQUAL | LESSEQUAL | GREATEQUAL | NOT);
NOT: not="not";
GREATEQUAL:
greateEqual+='greater' greateEqual+='or' greateEqual+='equal' greateEqual+='to';
LESSEQUAL: lessEqual='less' 'or' 'equal' 'to' ;
NOTEQUAL: notEqual='not' 'equal' 'to';
GREATER: greater='greater' 'than';
LESS:less+='less' less+='than';
EQUAL: equal='equal' 'to';
```

Fig. 25. The Grammar of the Arithmetic and Comparison Operators of Formal Requirements

```
CTLOperator:
AG | AF | EF | EG | AX;

AX:ax='all' 'next';
EG:eg='exist' 'globally';
EF:ef='exist' 'future';
AF:af='all' 'future';
AG:ag='all' 'globally' ;

LTLOperator:F | G | X;
F:f='future' ;
G:g='globally';
X:x='next';
```

Fig. 26. The Grammar of the Logic Operators of Formal Requirements

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

Fig. 27. The Grammar of miscellaneous elements