Modeling of Natural Language Requirements based on States and Modes

Yinling LIU and Jean-Michel BRUEL
IRIT-CNRS, Université de Toulouse
Toulouse, France
{firstName}.{lastName}@irit.fr

***Abstract*—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 requirements can clarify the relationship between states and modes, making the system safe.**

To this end, the MoSt language (a Domain Specific Language implemented on the Xtext framework) is proposed to modeling requirements based on states and modes. In this article, the relationship between states and modes is firstly provided. The metamodel and grammar of the language are then proposed. Finally, a validator is implemented to realise static checks of the MoSt model. The grammar and the validator are integrated into a publicly available Eclipse-based tool. A case study on requirements for designing cars has been conducted to illustrate the feasibility of the MoSt language. In this case study, we injected 9 errors. The results show that all the errors were detected in the static analysis.

***Keywords*—States and Modes, Requirements Modeling, Domain Specific Language.**

I. 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 [1]. 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. In other words, modes can actively influence system behaviors. Modes show more capabilities. While states are changed when conditions are satisfied.

Various aspects have been emphasized to analyse requirements, including context [2], the failures and successes of other requirements [3], and requirements evolution [4]. 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 [5]. They also aid in translating the user's version into the physical realization of the system [1]. They can be used as a medium to reduce misunderstandings between stakeholders such as users, acquirers, and developers as well [6].

Domain-Specific Languages (DSLs) are programming languages or specification languages that target a specific problem domain [7]. When the domain of one problem is covered by a particular DSL, we will solve that problem in an easier and faster way via using that DSL rather than a general-purpose language like Java or C, etc. In our case, we aim to create a new DSL to 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 rules 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. Thus, in this paper, a DSL - MoSt (stands for Modes and States) has been proposed to model requirements based on states and modes. The MoSt language was implemented using the Xtext framework [7].

Furthermore, the MoSt language has been used to model the requirements of designing a car. All the injected errors in the MoSt model have been successfully detected. Requirements engineers can use the MoSt modeling language to formalize requirements, so as to better organize, accurately express, and effectively manage requirements. The extracted information on states and modes can serve as "standard" terms when team members communicate with each other. So, unnecessary conflicts on the system description can be avoided.

The remainder of this paper is structured as follows. Section II briefly reviews the main related work. Section III presents all the elements about how to design the MoSt modeling language. Section IV gives a systematic

evaluation of the language to illustrates its feasibility. Section V concludes the paper with future perspectives.

II. Literature Review

A. Requirements Modeling

A majority of work focuses on how to design a new language to facilitate requirements analysis. The new languages proposed include EARS [8], OCL_{TM} [3], FRET [9], etc. Most of the languages are dedicated to better requirements eliciting and verification. Different viewpoints of analyzing requirements have been considered, including developers [10], uncertainty [4], [2], awareness requirements [3], machine learning requirements [11], etc. 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.

III. The MoSt Modeling Language

A. Relationship between States and Modes

In this paper, we propose our proper definitions of modes and states. We argue modes are the abstraction of use cases as [1] mentioned. Modes transitions happen only when the corresponding signals 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.

B. MoSt Metamodel

The MoSt metamodel highlights the properties of the MoSt Modeling Language. As shown in Fig. 1, MoSt is capable of describing NLRs (Natural Language Requirements) 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.

MoSt focuses on describing functional and 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 demands that the MoSt model must be self-included to make it verifiable. The concepts of *Mode*, *State*, and *Constraint* describe functional requirements. Non-functional requirements can only be expressed via *Constraint* concept. More specifically, concepts *PropertyConstraint* depicts functional and non-function requirements to be checked. On the other hand, *EnvironmentRequirement* concept initializes values and ranges of system attributes.

Concept *Condition* is one of the most important concepts in this meta-model, which includes 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 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 former is because the trigger conditions of modes transitions are only signal conditions. The latter is because environment requirements are only used to describe initial status of variables.

C. MoSt Grammar

The MoSt grammar illustrates the rules describing how to write different types of requirements. 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.

1) Natural Language Requirements: The natural language requirement rule is illustrated as follows:

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

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

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

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

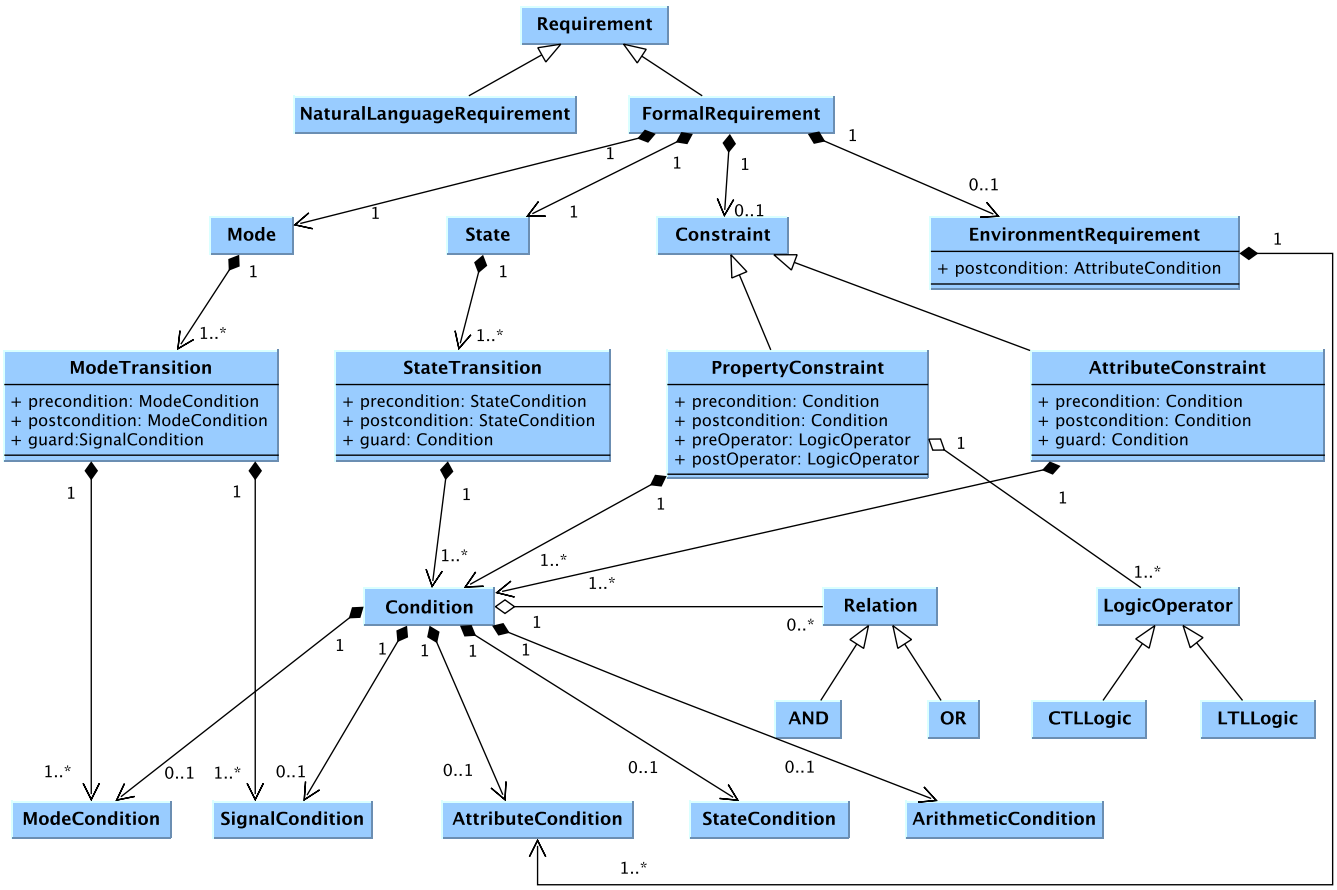

Figure 1. The MoSt metamodel

Two generic templates are used including "when..., then..." and "... should be". This first template applies to mode, state, property, and 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 [12]. That's why we would like to use different kinds of specific conditions to define pre-conditions, post-conditions. Our template is similar to one of EARS template "WHEN <optional preconditions> <trigger> the <system name> shall <system response>" [8]. 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 behaviours, states, and optional features. Ubiquitousness can be expressed by "mode = normal". Optional features can be described by "mode = A" (A is related to optional features). 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. To save space, we choose to just discuss four rules - ENVIRNOMENT, MODE, STATE, and PROPERTY.

1). Environment Requirements

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

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

An example of environment requirements can be written as follows: *[1] The accSpeed should be initialised to 0 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. Signal

conditions imply the condition for triggering mode transitions. The rules of mode requirements are listed as follows:

MODE: modeReqID=ReqID 'when' preModeCondition =MODECONDITION relation= RELATION guard= SIGNALCONDITION ',' 'then' postModeCondition = MODECONDITION'.' ;

An example of mode requirements can be written as follows: *[2] when the car is in mode economic and it receives Ac signal, then it is in mode sportive.*

3). State Requirements

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

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

An example of state requirements can be written as follows: *[3] 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). Property Requirements

Property requirements aid in describing functional requirements and non-functional requirements. They will be used as properties that need to be checked. Classic temporal logics (CTL and LTL) are considered in our language. The rules of property requirements are expressed as follows:

PROPERTY:
propertyReqID=ReqID 'when' preOperator= (CTLOperator | LTLOperator) prePropertyConditions += (STATECONDITON | ATTRIBUTECONTION | MODECONDITION) (preRelations+=RELATION prePropertyConditions += (STATECONDITON | ATTRIBUTECONTION | MODECONDITION))* ',' 'then' postOperator = (CTLOperator | LTLOperator) postPropertyConditions += (STATECONDITON | ATTRIBUTECONTION | MODECONDITION)* (postRelations+=RELATION postPropertyConditions += (STATECONDITON | ATTRIBUTECONTION | MODECONDITION))* '.';*

An example of property requirements can be written as follows: *[4] when all globally the car is in state autonomy and it is in mode economic, then all next it is not in state accelerate.*

IV. Evaluation

A. System description

This example is based on a UML state machine diagram for a car[1]. The car has five states including *Parking*, *Ignition*, *Start*, *Accelerate*, and *Autonomy*. We assume that this car has the function of the autonomous driving. The details of the car's state transitions are shown in Fig. 2.

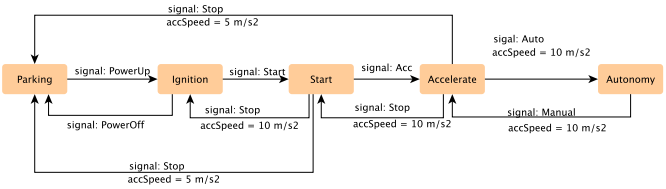

Figure 2. The State Machine Diagram of a Car

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 accelerate speed to $5m/s^2$ and $10m/s^2$, respectively.

B. Requirements formalization

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. 2. If we start from requirements documents, we will also need to extract the information of state transitions. This project is publicly available on GitHub[2].

In the following, we provide examples for states, attributes, modes, and property requirements, 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.
- 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.

C. Requirements Static Checking

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

[1]https://www.softwareideas.net/a/1539/Car-States--UML-State-Machine-Diagram- (accessed in May 2022)

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

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. Consistency checks analyze the legality of requirements, regarding user-defined rules. For example, one of our rules is "Non-integer variables should only be initialised once". If two requirements are written as "[1.1] The doorIsOpen should be initialised to FALSE." and "[1.2] The doorIsOpen should be initialised to TRUE.", then error "Non-integer variables should only be initialised once! 'doorIsOpen'" will be shown on these two requirements. In the MoSt model, the 9 injected errors have been detected by the static analysis, shown in Fig. 3.

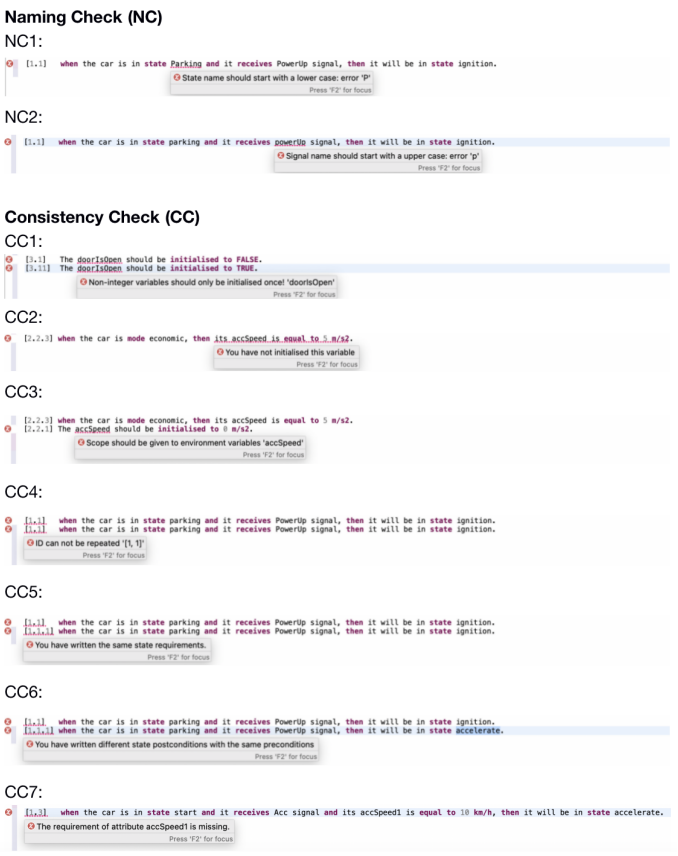

Figure 3. Results of Requirements Static Checks

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.

V. Conclusion and future work

NLRs modeling is never an easy thing. In this paper, the MoSt language is designed and implemented to model NLRs from the viewpoint of states and modes. To design the language, we provide the clear relationship between states and modes, the metamodel of the language, and its grammar. To implement the language, we use the Xtext framework to implement the designed grammar. The static analysis of the MoSt modeling is achieved via implementing user-defined rules.

The next step of our work is to study requirements verification. Requirements verification can investigate the impacts of the activation of one mode on system beahviors. The future work will focus on implementing a code generator to realise the transformation of the MoSt model into the formal model so that the MoSt model can be formally verified. We hope that the MoSt modeling tool can be directly connected to external model checkers to make the formal analysis transparent to users.

Acknowledgements

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

[2] M. Ahmad, N. Belloir, and J.-M. Bruel, "Modeling and verification of functional and non-functional requirements of ambient self-adaptive systems," *Journal of Systems and Software*, vol. 107, pp. 50–70, 2015.

[3] V. E. Silva Souza, A. Lapouchnian, W. N. Robinson, and J. Mylopoulos, "Awareness requirements for adaptive systems," in *Proceedings of the 6th international symposium on Software engineering for adaptive and self-managing systems*, pp. 60–69, 2011.

[4] J. Whittle, P. Sawyer, N. Bencomo, B. H. Cheng, and J.-M. Bruel, "Relax: Incorporating uncertainty into the specification of self-adaptive systems," in *2009 17th IEEE International Requirements Engineering Conference*, pp. 79–88, IEEE, 2009.

[5] A. M. Olver and M. J. Ryan, "On a useful taxonomy of phases, modes, and states in systems engineering," in *Systems Engineering/Test and Evaluation Conference, Adelaïde, Australia*, 2014.

[6] M. Edwards, "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*, 2003.

[7] L. Bettini, *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.

[8] A. Mavin, P. Wilkinson, A. Harwood, and M. Novak, "Easy approach to requirements syntax (ears)," in *2009 17th IEEE International Requirements Engineering Conference*, pp. 317–322, IEEE, 2009.

[9] D. Giannakopoulou, T. Pressburger, A. Mavridou, and J. Schumann, "Automated formalization of structured natural language requirements," *Information and Software Technology*, p. 106590, 2021.

[10] H. J. Goldsby, P. Sawyer, N. Bencomo, B. H. Cheng, and D. Hughes, "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)*, pp. 36–45, IEEE, 2008.

[11] S. Nalchigar, E. Yu, and K. Keshavjee, "Modeling machine learning requirements from three perspectives: a case report from the healthcare domain," *Requirements Engineering*, pp. 1–18, 2021.

[12] C. A. R. Hoare, "Procedures and parameters: An axiomatic approach," in *Symposium on Semantics of Algorithmic Languages* (E. Engeler, ed.), vol. 188 of *Lecture Notes in Mathematics*, pp. 102–116, Springer, 1971.