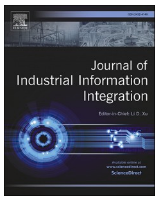
Journal of Industrial Information Integration 28 (2022) 100347

Contents lists available at ScienceDirect

Journal of Industrial Information Integration

journal homepage: www.sciencedirect.com/journal/journal-of-industrial-information-integration



An Integrated Intelligent Modeling and Simulation Language for Model-based Systems Engineering

Lin Zhang [a], Fei Ye [a,1,*], Kunyu Xie [a,1], Pengfei Gu [a,1], Xiaohan Wang [a,1], Yuanjun Laili [a], Chun Zhao [b], Xuesong Zhang [c], Minjie Chen [d], Tingyu Lin [e], Zhen Chen [a]

[a] *Beihang University; Xueyuan Road No.37, Beijing, China*
[b] *Beijing Information Science and Technology University; North Ring No.35, Chaoyang District, Beijing, China*
[c] *Jilin University; Qianjin Street No.2699, Chaoyang District Changchun, Jilin Province, China*
[d] *Beijing Huaru Technology Co., Ltd; Dongbeiwangxi Road No.10 Haidian District, Beijing, China*
[e] *Beijing Institute of Electronic System Engineering; Yongding Road No.52, Haidian District Beijing, China*

A R T I C L E I N F O

Keywords:
Modeling and simulation language
Model-based systems engineering
Integration
Intelligent
Verification

A B S T R A C T

Modeling and simulation are now leading the way in supporting analysis and development of system of systems. At present, to support the unified modeling formalism and dynamic simulation for different domain specific models across MBSE process, system modeling languages (such as SysML) are often required to cooperate with multi-physics modeling languages and simulation platforms (such as Modelica, Simulink), which makes it challenging to ensure the true unity of the whole system models, the consistency between the various system layers and the traceability of the modeling and simulation processes. In response to the above problems, this paper develops a new integrated intelligent modeling and simulation language, which can uniformly describe the system-level architecture and physical behavior models as a whole. On this basis, models can be simulated directly to support system verification for MBSE. Compiler and simulation engine are developed to enable X language to support the simulation of continuous, discrete event and agent models. Finally, an intelligent car system is taken as a case to verify the modeling and simulation capabilities of X language.

1. Introduction

In recent years, Model-based Systems Engineering (MBSE) has become an essential means to support the development of complex systems, especially system of systems (SoS) [1–3]. Taking complex products as an example, MBSE transforms the traditional research and development (R&D) method based on documents and physical models into a model-driven R&D method. This formal description method renders MBSE reusable, unambiguous, intelligible, and easy to spread. MBSE employs Systems Modeling Language (SysML) to realize the model-based integrated management and optimization of the whole product development process [4]. Since a SysML model cannot be directly simulated, it is necessary to use other simulation methods to verify the correctness and completeness.

One of the mainstream approaches is to uniformly describe system components of different domains based on a unified modeling language to achieve seamless integration and data exchange of multi-domain models [5]. For complex products that incorporate mechanical, electrical, hydraulic, and control engineering, the requirement description and architecture design are first conducted based on system modeling languages (such as SysML, IDEF, etc.). Then the physical models are developed and integrated in accordance with the physical modeling languages (such as Modelica, etc.) and the integration standards (FMI, HLA, etc.). Finally, different stages of product development are uniformly managed through mappings and transformations between the system models and the physical models for full system modeling and simulation.

However, due to the disconnection between the system modeling languages and the physical domain simulation languages, the connection needs to be realized through a transformation. Therefore, this

* Corresponding author.
E-mail addresses: zhanglin@buaa.edu.cn (L. Zhang), yefei@buaa.edu.cn (F. Ye), zy1903114@buaa.edu.cn (K. Xie), by2003151@buaa.edu.cn (P. Gu), by1903042@buaa.edu.cn (X. Wang), lailiyuanjun@buaa.edu.cn (Y. Laili), zhao_chun@189.cn (C. Zhao), xs_zhang@126.com (X. Zhang), jimi_chen@163.com (M. Chen), lintingyu2003@foxmail.com (T. Lin), czhen@buaa.edu.cn (Z. Chen).
[1] These authors contribute equally to this work.

https://doi.org/10.1016/j.jii.2022.100347

Available online 23 April 2022

approach falls behind in ensuring the consistency and traceability of the whole system modeling and simulation process. Worse still, this method lacks the ability of intelligent modeling and simulation, which is of vital importance for intelligent actions in development processes and/or intelligent functions of complex products.

Thus, this paper develops a new integrated intelligent modeling and simulation language, which can uniformly describe the system-level architecture and physical behavior models as a whole. On this basis, models can be simulated directly to support system verification for MBSE. At the system modeling level, six parts of the definition, requirement, connection, equation, action, and state machine are designed based on the object-oriented approach to represent the architecture and behavior [6]. At the level of simulation and verification, the continuous, discrete event, and agent models are incorporated into the couple models of DEVS (Discrete Event System Specification) [7]. A specific tool XLab is developed to realize the modeling and simulation of the whole system.

The remainder of this paper is organized as follows. Section 2 presents related work from the perspective of system modeling, physical modeling, agent modeling, as well as the integration of system design and verification. In Section 3, we elaborate on the overall structure of X language, including the hierarchical structure of X language, the correspondence between graphics and text, and the design of X language classes. Section 4 gives an introduction to the essential elements and grammatical structure of X language, mainly in the form of classes. In Section 5, we introduce the compiler of X language and the framework of the simulation engine. In Section 6, we demonstrate the modeling and simulation capabilities of X language using an intelligent car system. Finally, we summarize this paper and briefly outline the further research work.

2. Related Work

When it comes to system modeling, typical practices include, to cite but a few, the modeling methods based on the DEVS, system modeling methods based on the SysML language, multidisciplinary unified modeling methods based on the Modelica language, Bond diagram-based system dynamic structure modeling method, European simulation language (ESL)-based software and hardware coordination modeling method, Dymola language-based system dynamics modeling method, and high-level architecture (HLA)-based distributed simulation system modeling method. These languages and methods have been studied and applied to varying degrees in industry and academia.

2.1. System Modeling Language

Before the advent of SysML, many modeling languages and tools were used in systems engineering, such as IDEF, N2 diagrams, behavior diagrams, etc. These modeling languages employed different symbols and semantics, which cannot be interoperated and reused, thus restricting the effective communication between systems engineers and those of other disciplines on system requirements and design, and affecting the quality and efficiency of systems engineering. To meet the actual needs of systems engineering, the INCOSE (International Council of Systems Engineering) and the OMG (Object Management Organization) decided to propose a new system modeling language—SysML on the basis of reusing and extending a subset of UML 2.0 as a standard modeling language for systems engineering [5]. SysML supports the specification, analysis, design, verification, and validation of a broad range of complex systems. These systems may include hardware, software, information, processes, personnel, and facilities [8]. It supports multiple structured and object-oriented methods and multiple procedures. However, native SysML models are static and cannot be directly used to verify the correctness and completeness. Thus, the SysML models should be transformed into domain-specific models, such as Modelica models [9]. There has been a lot of work in this direction [10–15]. Generally, for systems with different characteristics, the types of SysML diagrams adopted are also different. Literature [16] proposes a method of using SysML parameter diagrams to describe the behavior of continuous systems. Literature [12] and [17] elaborate on the description of discrete event systems based on SysML action, sequence, or state machine diagrams. Although SysML models can be extracted and used by the transformation methods, system engineers have to add a large number of simulation codes, especially those related to system behavior, to obtain executable simulation models, which is a tedious process and not very versatile [18].

2.2. Physical Property Modeling Language

A complex system generally contains different domains, such as mechanics, electronics, control, hydraulics, and pneumatics. As a result, the cost is surging to verify and optimize characteristics of a complex system through physical experiments, and modeling and simulation of multiple domain physical properties prove to be a more efficient way.

For modeling and simulation of physical systems in a single domain, traditional software generally establishes system state equations for a single energy domain based on fundamental physical laws (such as Newton's laws of mechanics, Kirchhoff's laws, etc.), solves the equations through computer programming and finally obtains the system response characteristics. According to the known mathematical models, the typical software, such as Matlab and Simulink of MathWorks, is formed on the block diagram modeling methods. Other software such as PowerDEVS and Modelica also employs similar modeling methods.

Although most simple systems can be simulated by the above methods of a single domain, the actual engineering systems often incorporate the couple of multiple energy forms, such as mechanical, electromagnetic, hydraulic, and chemical energy. In this case, only interface-based co-simulation is feasible. It is difficult to obtain a mathematical model of a multi-domain system using a single-domain method or to realize the automatic generation and simulation analysis of the model in a unified form on the computer. To address this problem, in 1961, Professor Herry Paynter of the Massachusetts Institute of Technology, from the viewpoint of energy system dynamics, proposed a theoretical framework for a unified modeling method of system dynamics suitable for the coexistence of multiple energy domains [19]. It lays the foundation for dynamic analysis, modeling, and simulation of multi-energy domain couple systems. Since then, various bond graph technologies have been proposed for modeling and simulation applications of different systems [20].

The modeling method based on bond graphs can solve the modeling problems of multi-domain physical systems; however, the construction of model details based on graphs is not convenient enough. In 1997, the theoretical framework of energy conservation based on the bond graph and the equation-based Modelica language was proposed to support object-oriented modeling, declarative modeling, non-causal modeling, and multi-domain unified modeling as well as modeling of hybrid, i.e., both continuous and discrete, systems [21]. The Modelica language has then risen to be the mainstream language for multi-domain physical system modeling due to its high model reusability, ease of use, no symbol processing, and many other advantages. At the same time, the system standard library of Modelica language also provides essential components and typical system models in many fields, including electricity, fluids, thermodynamics, machinery, etc. [22], which offers great convenience in model development and simulation of physical systems.

Despite the fact that Modelica is able to model and simulate hybrid models, the equation-based language characteristics make it inept to support discrete event simulation well, resulting in to inconvenient description and low simulation efficiency [23–25]. Therefore, the application of Modelica lies more suitable in multi-domain physical system modeling rather than in modeling large-scale discrete systems.

2.3. Intelligent Extension of Modeling Languages

To meet the requirements of modeling intelligent activities and behaviors of complex systems, modeling languages need to be extended with intelligent factors to better describe the complex agent and neural network models and to strengthen the capability in interaction, sensing, and learning. As an essential method to achieve complex system modeling and simulation, agent-based models and multi-agent systems have seen ever-widening applications in the fields of simulation, artificial intelligence, and control. In order to efficiently construct various agent models, some modeling languages supporting agent models were proposed in different aspects.

As one of the most commonly used languages in modeling and simulation, Modelica has also been applied in the research of agent models. However, due to the insufficient support for discrete system modeling, it is not suitable for agent modeling. Therefore, the research on agent modeling based on Modelica mainly focuses on the support of third-party libraries. Bünning [26] pointed out that native Modelica is unsuitable for modeling agents and built a third-party library for multi-agent systems based on Modelica. Sanz [27] designed the ABMlib library rfor the type, behavior, communication, and environment of agents to improve the performance of Modelica based on agent models. In addition, some studies focus on applying Modelica to describe the continuous behavior or action of an agent rather than modeling the agent or the overall system agent [28,29].

SysML plays a vital role in the system-level model design. Some works revolve around SysML to provide support for the construction of agent models. Sha [30] introduced the concept representation of agents with SysML, supported and verified the early conceptual model of agents, and provided a case to demonstrate the method proposed in the article. Literature [31] proposed a set of dynamic modeling methods based on SysML to establish agent models and emphasized the important influence of the environment, including the fact that the environment determines the action and updates the parameters of agents. As a system-level language, SysML can describe the event relationship among agents and can easily establish an agent conceptual model. However, as the native SysML model is static, the simulation of agent models needs to rely on third-party simulation tools. Besides, SysML is also insufficient in describing the detailed behaviors and actions of agents, as well as some specific multi-agent algorithms.

As a kind of discrete models, agents can be modeled and simulated by DEVS. In the literature [32], a complex agent perception architecture is constructed based on multiple types of atomic models with the BDI (Belief-Desire-Intention) model as its component. However, the entire model is too bulky and redundant as many parts are uncommon to most agent models. In addition, the BDI part in the paper is only represented in the form of symbols rather than in the atomic model. Akplogan [33] used the DEVS couple model to build a BDI agent model to solve the problem of agent decision-making in agricultural applications and provided a specific application to prove its feasibility of the overall architecture. From the perspective of multi-agents, Müller [34] built a set of system models using DEVS and modified the expression of the original DEVS atomic model to adapt to the multi-agent characteristics. However, this method is not applicable when facing a single complex agent, as the specification of DEVS is rudimentary compared with the agent models.

In summary, although several representative modeling languages may support the construction of agent models and multi-agent systems, they only apply agent models at the level of basic methods or integration with third-party libraries, lacking the ability of autonomy. In other words, these languages themselves are not designed for modeling multi-agent models, which can make intelligent and dynamic decisions by reasoning and learning, so it is difficult to avoid problems such as poor interpretability, poor algorithm representation, low autonomy, and low modeling efficiency.

2.4. Integration of System Modeling and Simulation

SysML has the ability to describe the system model but cannot be directly used to verify its correctness and completeness. Therefore, it needs to be transformed into an executable one to make up for the deficiency, which is the common practice used by current engineers and researchers.

Schamai et al. [35,36] proposed a mapping method ModelicaML based on the UML extension method for Modelica transformation. It uses state machine diagrams as the carrier for hybrid modeling of discrete and continuous behavior and adds annotations to state transformation to describe continuous behavior, thereby providing a more complete solution for integrating design and simulation behavior. However, this method lacks formal model expression. The description based on plain text can not effectively express and manage models, and the parameter correspondence between the state machine diagrams and structure models is not attained, which makes ModelicaML unable to fully support all grammar standards of Modelica. Gauthier et al. [37] used the ATL (Atlas Transformation Language) to map a SysML model to a Modelica model based on the SysML4Modelica extension package [38] proposed by the OMG to verify the accuracy and completeness of a design model. Compared with the QVT (Query/View/Transformation) mapping method applied by OMG, Gauthier et al. have some innovations in the mapping method. However, due to the incomplete definition of the SysML4Modelica extension package, the description of the Modelica syntax is not perfect. Cao et al. [39] proposed a unified behavior model extension method based on SysML. This method is combined with Matlab/Simulink to realize the automatic transformation between the design and simulation models by establishing a supplementary simulation model. On the flip side, this method is more focused on the simulation of the control system field and is weak in supporting the multi-domain modeling and simulation of the physical system, so it is not applicable to the simulation of complex systems engineering. Li et al. [40] developed a modeling language that supports modeling and simulation of the continuous and discrete systems, providing parallel solutions for the simulation optimization problem, but not good at system-level modeling. Li et al. [41] proposed a SysML-based visualization model transformation method from the perspective of a meta-model. They determined the transformation relationships between the SysML source and Modelica target models by hierarchical instantiation modeling of the transformation rule and transformation activity meta-model, thereby implementing the dynamic transformation activities. Despite that, this method does not extend SysML but establishes a mapping relationship with Modelica based on the existing model elements of SysML. Therefore, the specific description of complex products is insufficient, making the transformation between the two languages incomplete. Zhou et al. [42] constructed the SysML extension package M-Design for Modelica based on the Modelica meta-models, and then defined the mapping rules between the two according to the extended SysML and Modelica meta-models, thus implementing the automatic transformation from the SysML design models to Modelica simulation models. Even so, the extension package only defines the basic meta-model of Modelica, and some advanced features escape the description. Besides, the ATL-based mapping method only implements the one-way model transformation from SysML to Modelica and fails to implement the bidirectional model transformation.

In addition, complex systems involve models in multiple domains. These models use different formalisms, modeling languages, and tools to solve specific problems, bringing significant challenges to consistency management. Different techniques were proposed to alleviate the consistency problem in systems engineering studies. MBSE tools like SCADE Architect can be directly integrated into SCADE Suite, providing system and software teams with the same environment to synchronize requirements, avoiding duplication and inconsistency [43]. Herzig et al. [44] presented a conceptual basis for inconsistency management in MBSE that a model can be represented by a graph and that

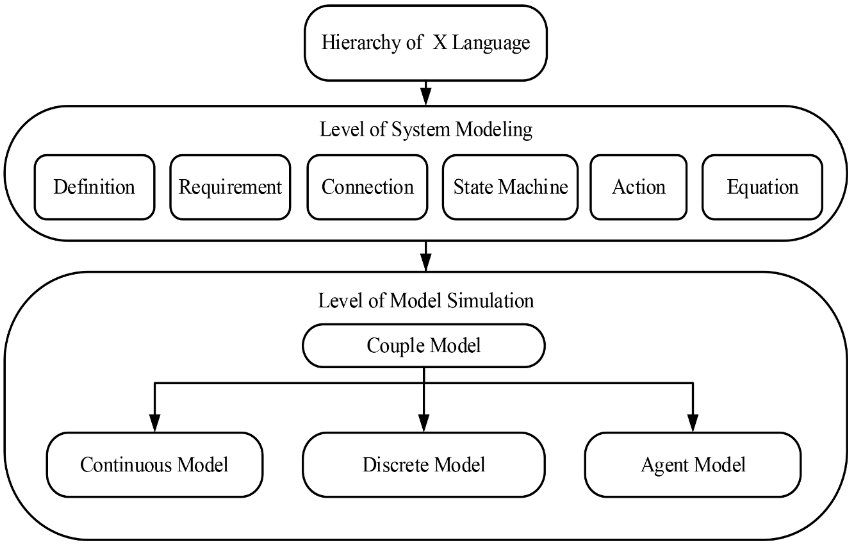


Figure 1. Hierarchy of X language

inconsistencies manifest as subgraphs. To identify inconsistencies, graphs can be queried using partially defined graph patterns. However, the authors did not provide proof of the technical viability and practicability. Feldmann et al. [45] introduced a conceptual approach based on semantic web standards allowing for identifying inconsistencies in heterogeneous models and demonstrating its technical viability. A possible disadvantage is that the evaluation of the conceptual framework's viability is preliminary since only a small system was used as a demonstration case. Jongeling et al. [46] proposed the idea of extending the OpenMBEE platform to include code as a view. They believe that this extension will allow simple structural consistency checks between the SysML system model and C/C++ code and provide engineers and managers with insight into model-code consistency. Berriche et al. [47] proposed a model synchronization approach to actively check for model consistency in a continuous way during the multidisciplinary design process. However, this method has some limitations: the model synchronization method is only applicable to structural and hierarchical models. In addition, the classification and resolution of differences is a manual process that relies on the activities of the project manager.

In summary, the existing modeling languages are mainly aimed at a specific part of modeling and simulation. They lack the ability of full-process (the entire lifecycle) collaborative design and verification. Although the integration of system design and verification can realize the unified management of different stages during product development, it is still achieved through the mapping and transformation between languages. It may be effortless to deal with a single domain model but challenging to support the modeling and simulation of complex systems that contain the continuous, discrete event and intelligent properties.

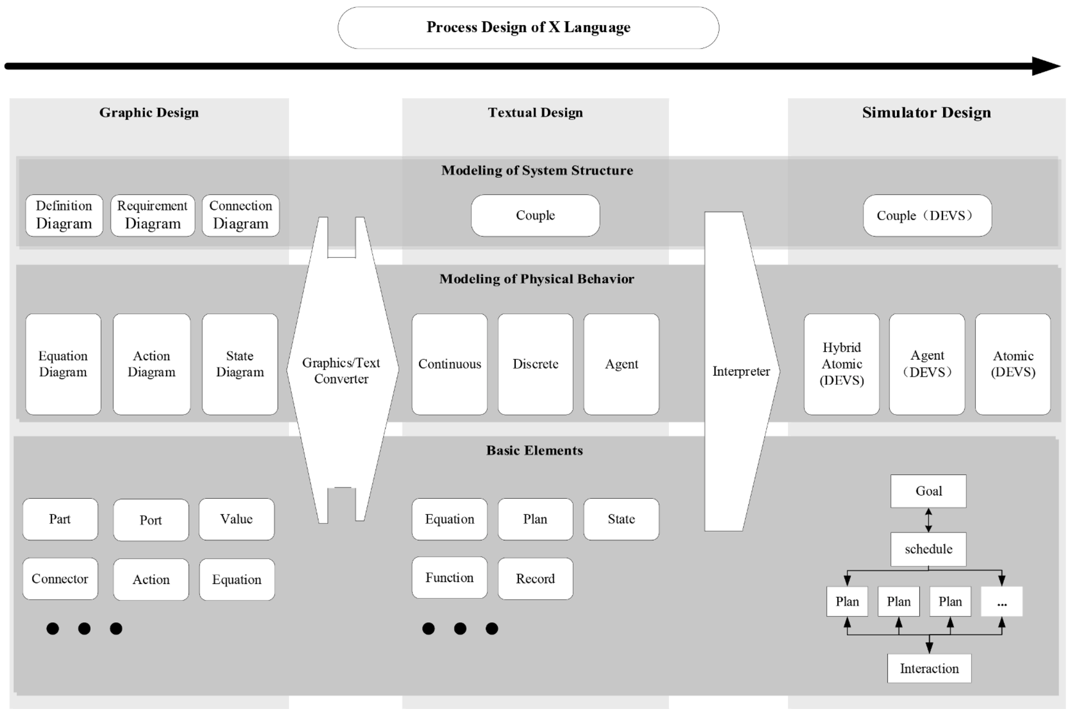


Figure 2. Design process and elements of X language

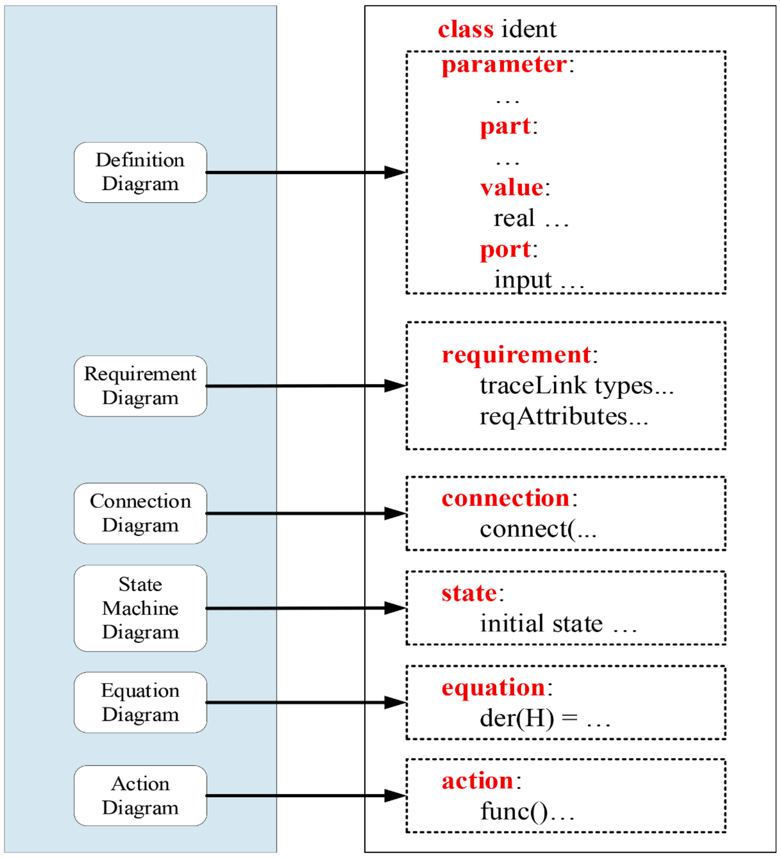


Figure 3. Correspondence between graphical and textual syntax of X language

3. Overall Structure of X Language

As can be seen from above, the existing modeling languages cannot achieve full system modeling and simulation. Aiming at this problem, this paper proposes a new integrated intelligent modeling and simulation language-X language. As shown in Figure 1, at the level of system modeling, six parts of the definition, requirement, connection, equation, action, and state machine are designed to represent structure and behavior. At the model construction and simulation level, the continuous, discrete event, and agent models are regarded as part of the couple model of DEVS, which supports the simulation verification of physical behavior. Based on the design, X language is endowed with the capabilities to: 1) support two modeling forms of graphics and text, and based on XLab, the two forms of models can be converted to each other. 2) support system-level architecture and physical behavior modeling and simulation verification. 3) support modeling for various complex agent models, including agent learning, communication, and multi-agent parallel simulation. 4) support continuous, discrete event, and hybrid simulation.

X language is a modeling language that supports model-based systems engineering. As shown in Figure 2, from the perspective of modeling process, it includes two modeling forms of graphics and text, as well as an engine for simulation. As for language elements, there are structural elements, behavioral elements, and other essential elements. Figure 3 shows the one-to-one correspondence between the graphic and textual forms of X language and its six types of diagrams: definition diagram, requirement diagram, connection diagram, state machine diagram, equation diagram, and action diagram. Each diagram corresponds to the language text and exists as a part of the class. Therefore, a single part can only describe an aspect of the class rather than the complete one. That is, a class is a collection of contents described by multiple parts.

The six parts of X language have their respective focuses. The definition part and the connection part define the system model from the system structure level, explaining which components the system contains and the connection relationships among them. The requirement class is defined for requirement description, analysis, and tracking. Modelers generally use a variety of relationships to establish the traceability between requirements, and the traceability from requirements to the structure and behavior of the system models.

The equation part, action part, and state machine part describe the system behavior from the perspective of the mathematical equation, assignment process, and state machine, respectively. Different modeling elements are selected depending on the characteristics of the model; for example, continuous models can be modeled by equations, while discrete models by state machines. Therefore, different forms are required to describe the behavior of different types of system models or components. On this basis, the restricted classes of X language are classified: the continuous class is used to describe continuous multi-domain physical systems, the discrete class describes discrete systems, and the agent class describes agent systems.

X language supports model-based systems engineering and has the ability to verify the entire process of system design. The models built in either graphics or text can be directly translated into executable DEVS codes via the compiler. X language simulation engine is a multi-domain engine designed based on DEVS, which can support cross-domain modeling in multiple domains of continuous, discrete events, and agents. The results obtained by the simulation can be directly fed back to the engineer to realize the functional verification of the system design.

4. Essential Elements and Grammatical Structure of X Language

Class is the basic unit of a model in X language, which can be further divided into basic classes and restricted classes. The basic classes are

Table 1
Functions of X language classes

Types	Functions
Class	Supporting the description of any types of model entities
Continuous	Supporting the description of model entities with continuous behavior
Discrete	Supporting the description of model entities triggered by events
Agent	Supporting the description of model entities with intelligent behavior
Couple	Supporting the description of system-level model entities with multiple components
Requirement	Supporting the description, analysis and tracking of requirements
Record	Supporting the description of complex data structures in various model entities
Function	Supporting the description of algorithms required for various model entities to solve procedural modeling
Connector	Supporting the description of connectors that follow Kirchhoff's law in various model entities

modified by the keyword *class* and applied to describe the structural and behavioral characteristics of any entities. The restricted classes are mainly used to describe models of different characteristics more accurately and improve the readability. They are modified by specific keywords, including *continuous, discrete, couple, agent, requirement, record, function,* and *connector*. The types and functions of the classes are shown in Table 1.

X language provides two modeling forms: graphics and text. The concept of class is for the entire language; that is, it is applicable to both text and graphic model forms. Diagrams refer to the elements of the graphical models, including definition diagram, requirement diagram, connection diagram, state machine diagram, equation diagram, and action diagram. Correspondingly, the text models also contain 6 parts, namely the definition part, requirement part, connection part, state part, equation part, and action part. A class is often a combination of multiple components, as shown in Table 2, which lists the components contained in each class.

The following content will specify the essential elements and grammatical structure of each class, and the extended BNF [6] of each component is demonstrated in Appendix A.

Table 2
Composition of the two forms of X language classes

Class	Composition of diagrams	Composition of text
Class	Definition diagram, Requirement diagram, connection diagram, equation diagram, state machine diagram, action diagram	Definition part, Requirement part, connection part, equation part, state machine part, action part
Continuous	Definition diagram, equation diagram	Definition part, equation part
Discrete	Definition diagram, state machine diagram	Definition part, state machine part
Agent	Definition diagram, action diagram	Definition part, action part
Requirement	Requirement diagram	Requirement part
Couple	Definition diagram, connection diagram	Definition part, connection part
Record	Definition diagram	Definition part
Function	Definition diagram, action diagram	Definition part, action part
Connector	Definition diagram	Definition part

Table 3
Definition of continuous class

Continuous class	Related properties of continuous class	Descriptive objects of different properties
Definition diagram/ Definition part	Parameter	Definition of instantiation parameters and their types
	Value	Definition of state variables
	Port	Definition of ports
Equation diagram/ Equation part	Equation	Definition of behavior described based on mathematical equations

4.1. Continuous Class

According to the behavior characteristics of models, they can be divided into continuous models, discrete models, and hybrid models. The continuous class is defined for models with continuous behavior that constantly changes over time and can be abstracted by mathematical equations. The structural properties of entities with continuous behavior contain the definitions of parameters and their types, state variables, and ports. At the same time, the behavior properties can be defined by mathematical equations. Based on this, in graphics and text modeling, the continuous class in X language describes the structural attributes by the definition diagram and the definition part, and the behavior characteristics by the equation diagram and the equation part, respectively. The details are shown in Table 3.

4.2. Discrete Class

The discrete class is defined for discrete models. The behavior of discrete models are triggered by events and can be regarded as an abstraction of a series of states. The structural properties of discrete models generally include the definitions of the parameters and their types, state variables, and ports. The behavior parameters are defined based on the state machine theory. Accordingly, in graphics and text modeling, the discrete class in X language describes the structural characteristics by the definition diagram and the definition part, and the behavior characteristics by the state machine diagram and the state machine part, respectively. The details are presented in Table 4.

4.3. Couple Class

The couple class is defined to endow X language with the ability to describe couple models. Its principal function is to describe the components included in the couple model and the connections among them. Based on this, the couple class in X language adopts the definition diagram and the definition part to describe the system composition, and the

Table 4
Definition of discrete class

Discrete class	Related properties of discrete class	Descriptive objects of different properties
Definition diagram/ Definition part	Parameter	Definition of instantiation parameters and their types
	Value	Definition of state variables
	Port	Definition of ports
State machine diagram/ State machine part	State	Definition of behavior based on state machine description

Table 5
Definition of couple class

Couple class	Related properties of couple class	Descriptive objects of different properties
Definition diagram/ Definition part	Port	Definition of ports
	Part	Definition of components
Connection diagram/ Connection part	Connection	Definition of the connection relationship between components

connection diagram and the connection part to describe the component connection relationship, respectively. The details are shown in Table 5.

4.4. Agent Class

The agent class is defined for multi-agent models that emphasize more on the autonomy of agents, which can make intelligent and dynamic decisions by reasoning and learning. The design of the agent class in X language follows the BDI architecture and has been simplified. Among them, the most critical content is retained; that is, goals guide the execution of plans. A series of content such as environmental perception is left to users to define, thereby improving the extensibility of the language. For modeling with the agent class, the entire structure can be divided into two parts, the definition part and the action part. The former is used to initialize the values of parameters and variables, as well as the declaration of functions and plans. The latter is to control the execution of the plan and set the start and end conditions of the agent simulation.

The architectural correspondence between the agent class and the BDI is illustrated in Figure 4. The left side is the architecture of the agent class, and the BDI architecture is on the right. The plan corresponds to the intention part of the BDI architecture. The content defined in the execution part is associated with the goal part and the belief part is integrated into the entire process of interaction process between the agent and the environment. On the basis of retaining the characteristics of the original BDI architecture, the agent class is integrated with the syntax and semantics of X language, giving X language intelligent characteristics.

The agent models generally contain the behavior of communication and interaction with other entities, the agent class in X language describes the structure characteristics by the definition diagram and the definition part, and the behavior characteristics by the action diagram and the action part, respectively. The details are shown in Table 6.

Table 6
Definition of agent class

Agent class	Related properties of agent class	Descriptive objects of different properties
Definition diagram/ Definition part	Parameter	Definition of instantiation parameters and their types
	Value	Definition of state variables
	Plan	Definition of Agent behavior sequence
Action diagram/ Action part	Active	Definition of plan execution logic

4.5. Requirement Class

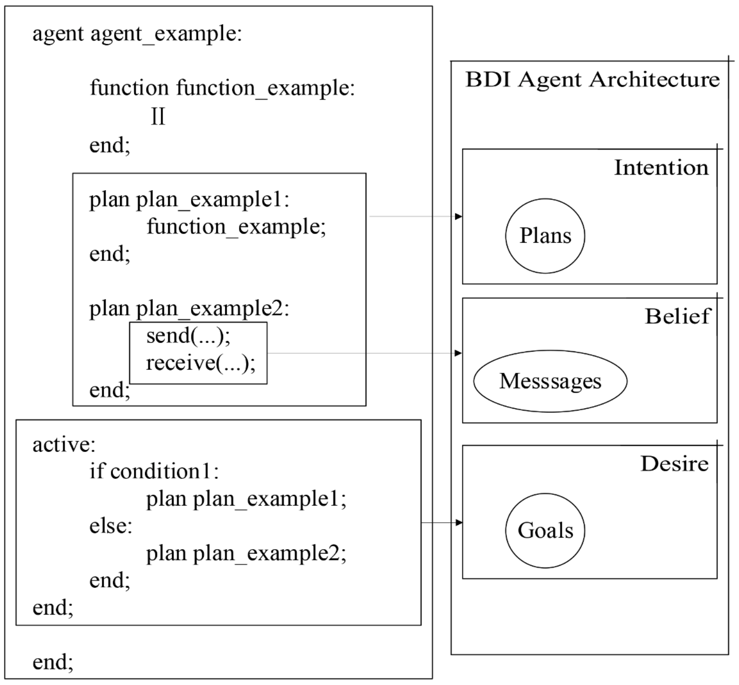
The requirement class is defined for requirement description, analysis, and tracking. It contains two parts, namely *Requirement Attributes* and *TraceLink Types*. The former describes the relevant attributes of requirements to illustrate the inherent characteristics of a specific requirement, such as ID, Name, Level, Type, etc. The latter is used to manage links between requirements and requirements and other modeling elements (including stakeholders, sources, and system development elements), forming links for demand tracking, such as Compose, Satisfy, Verify, etc. The details are shown in Table 7.

4.6. Record Class

The record class is defined for models with different data types. Accordingly, the record class in X language uses definition diagrams and definition parts to describe the types of data contained in graphics and text modeling, as shown in Table 8.

4.7. Function Class

The function class is defined for models with complex functional



```
agent agent_example:

        function function_example:
                []
        end;

        plan plan_example1:
                function_example;
        end;

        plan plan_example2:
                send(...);
                receive(...);
        end;

active:
        if condition1:
                plan plan_example1;
        else:
                plan plan_example2;
        end;
end;

end;
```

Figure 4. The architectural correspondence between the agent class and the BDI

Table 7
Definition of requirement class

Requirement class	Related properties of requirement class		Descriptive objects of different properties
Requirement diagram/ Requirement part	Requirement Attributes [48]	Identifier	Unique identifier of the requirement
		Name	Name of the requirement
		Level	Level of the requirement (Stakeholder/System/ Component)
		Type	Type of the requirement (General/Functional/ NonFonctional/ Physical/Design)
		Risk	Security level of the requirement (High/ Medium/Low)
		Source	Where the requirement originated from
		Stakehodler	Stakeholder who is in charge
		Text	Description text defined by the modeler
	TraceLink Types [48]	Compose	Relates requirement to its parent requirement
		Copy	The same requirement that appears in a different level
		Derive	Relates requirement to its derived requirement
		Refine	Relates requirement to its refined requirement
		Satisfy	Relates requirement to the block that fulfills it
		Verify	Relates requirement to test cases
		MappedTo	Relates requirement to a particular attribute, operation, state or value of the artifact
		OriginatedFrom	Relates requirement to its source
		ResponsibleOf	Relates requirement to its stakeholder

Table 8
Definition of record class

Record class	Related properties of record class	Descriptive objects of different properties
Definition diagram /Definition part	Value	Definition of different data types

Table 9
Definition of agent class

Function class	Related attributes of record class	Descriptive objects of different properties
Definition diagram/ definition part	Value	Definition of input/output parameters
Action diagram/action part	Active	Definition of function realization process

Table 10
Definition of connector class

Connector class	Related attributes of record class	Descriptive objects of different properties
Definition diagram/ definition part	Value	Definition of input/output data types

behavior. Generally, the function class is composed of the definition part and the action part. The former is used to define input and output parameters, while the latter is to specify specific functions of the function class. The details are shown in Table 9.

4.8. Connector Class

The connector class is defined for models with connectors that follow Kirchhoff’s law. The connectors define the types of data transferred between the entity and other entities. On this basis, the connector class is defined to describe non-causal connector ports as well as the types of data transferred between models or components. Generally, the connector class in X language describes the types of data transmitted through the definition diagram and the definition part when graphics and text are modeled, as shown in Table 10.

A class is a collection of contents described by multiple parts, and X language modeling framework consists of 6 parts. Therefore, a single part can only describe an aspect of the class rather than the complete one.

5. Compiler and Simulation Engine Design

5.1. Framework of X Language Compiler

X language is an object-oriented multi-domain system modeling and simulation language. It includes not only the characteristics of object-oriented programming languages but also those of equation-based modeling languages. These two kinds of languages usually employ different interpretation routes, leading to different technical routes in X language interpretation.

The framework of X language compiler is shown in Figure 5. The interpretation process is divided into three stages, namely the pre-processing stage, the intermediate processing stage, and the post-processing stage.

The pre-processing stage is responsible for lexical analysis and grammatical analysis of the source code to obtain the abstract syntax tree and collect the information of the elements in the model to make a symbol table.

In the intermediate processing stage, the compiler processes the abstract syntax tree that has been obtained by the pre-processing stage. Different technical methods are applied according to the types of models included in the interpreted document. For continuous models based on equations, the first step is to flatten the model to obtain a flattened equation set. Then in the stage of casualization, the equations are transformed into a form suitable for being solved by the differential equation solver. For assignment-based models, i.e., the agent models and the DEVS-models, the static type checking is first performed for each sentence contained in the model with the help of the symbol table constructed in the previous stage. Then different technical methods are selected according to different model types. The reason to adopt different compilation routes here is that the interpretation and code generation of the DEVS model can correspond to the simulation code one by one, while the representation of the upper model of an agent model differs drastically from the lower one, thus requiring the parameter correspondence and the integration of generated files to ensure that the necessary information is not lost. Therefore, the complexity of the two compilation routes is also quite different.

In the last stage, the processed data structures obtained in the different technical routes are traversed and the simulation files of the respective models are obtained. If the model has a hierarchical structure, the file level integration is required to obtain simulation files of the entire model.

5.2. Framework for X Language Simulation Engine

As X language is an object-oriented modeling language for multi-

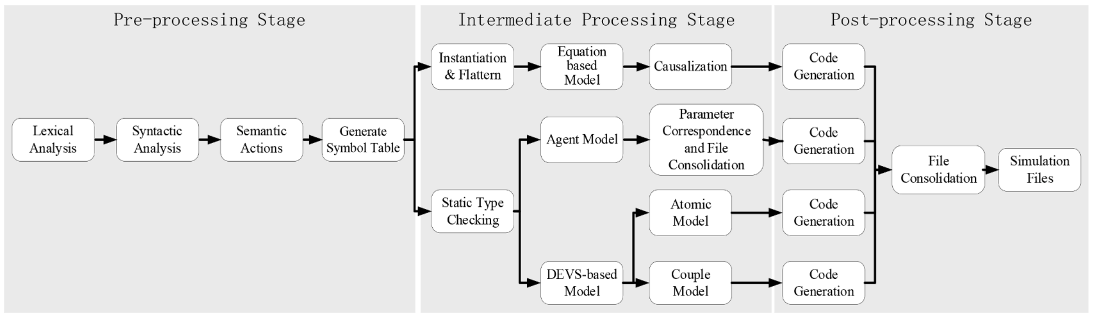


Figure 5. Framework of X language compiler

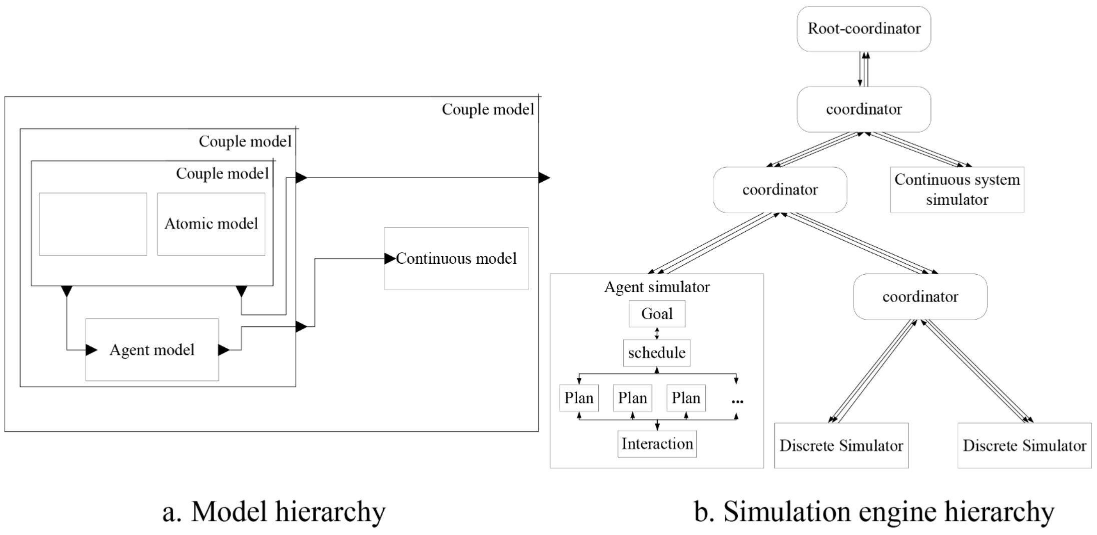


Figure 6. Simulation engine architecture

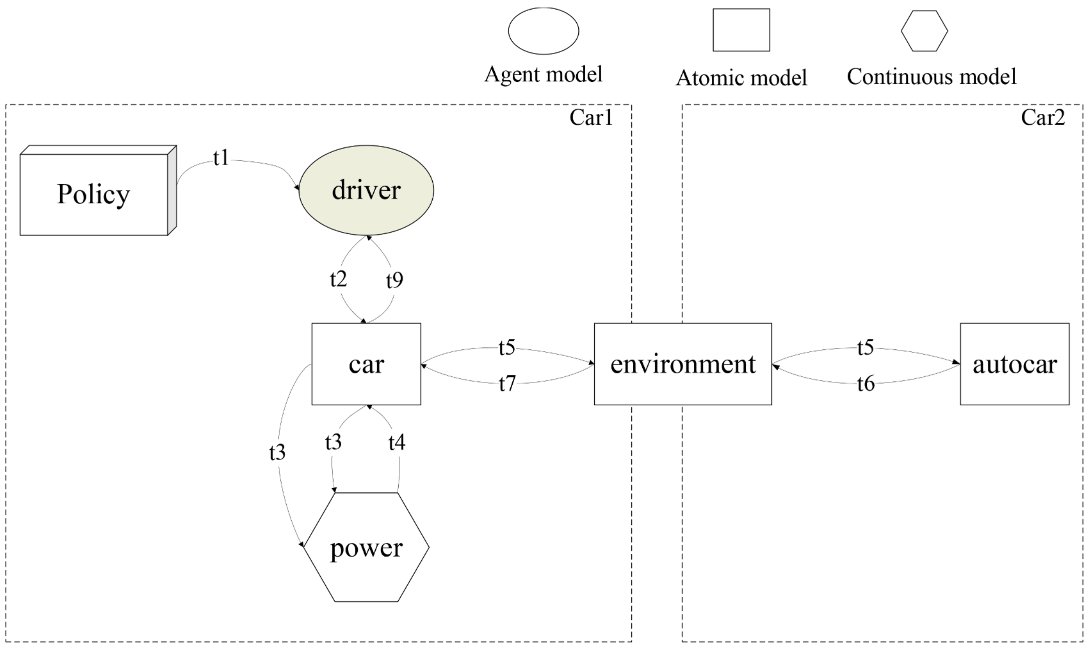


Figure 7. An intelligent car driving system

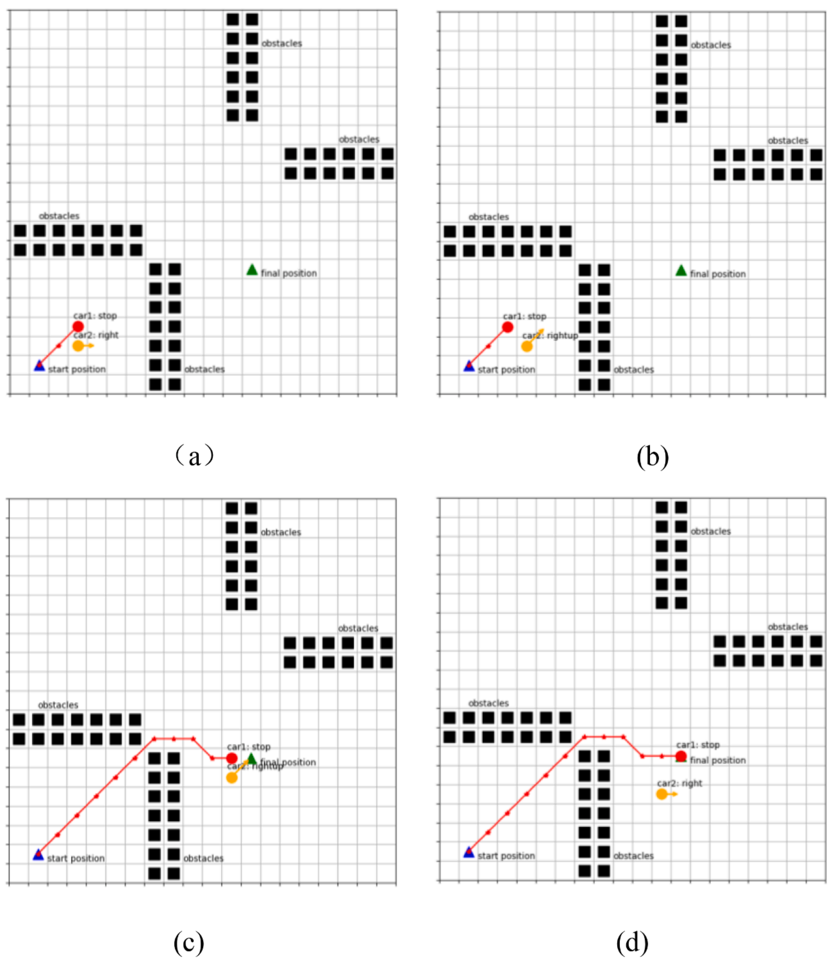


Figure 8. The dynamic diagram of the system

domain systems, it is necessary to consider the simulation capabilities for specific domains and the interaction between different domains to build a simulation engine. As a specification that supports multi-domain modeling, DEVS can simultaneously support continuous, discrete event, and agent simulation, which fully meets the needs of X language and provides the language with support for multi-domain model verification.

The most effective implementation of the DEVS simulation algorithm is the hierarchical recursive program developed by Zeigler. As shown in Figure 6, for the DEVS simulation structure, each atomic model corresponds to a *simulator* that executes the internal and external events of the atomic model. The couple model corresponds to the *coordinator*, which schedules the events of atomic models inside the model. The top level of the entire model corresponds to the *root-coordinator*, which manages the event advancement of the whole model.

X language still keeps the architecture of the DEVS engine, and several extensions are made to fit its particularity:

1) Firstly, the continuous model exists in the form of an atomic model in the engine, which is the extension of the continuous model based on DEV&DEVS. The state events and time events defined in the continuous model are further abstracted into the internal events of the atomic models. The equation is solved at the time nodes on which internal events and external events are triggered.
2) Secondly, the agent model exists as a couple model. According to DEVS, the couple model can be regarded as a particular atomic model, so the agent model is used as an atomic model in the engine structure. The agent model of X-language is implemented based on the BDI architecture and divided into four modules: *goal, schedule, plan*, and *interaction*. In actual design, the agent model can interact with other atomic models or agent models based on *interaction*.
3) Thirdly, based on the above two extensions, the agent and continuous models are presented as DEVS models so that the connection among the agent, discrete event, and continuous models is abstracted as an event-based connection.

In X language simulation engine, there is message-based communication between the connected *simulator* and *coordinator*. Each time an event occurs (internal or external), the *coordinator* sends a transformation message to its child nodes, informing them to perform the transformation. When the *simulator* executes the internal or external events, it calculates its next state. If an internal transformation is being performed, it needs to send the output to its parent *coordinator*. Therefore, there are three types of communication between the *simulator* and the *coordinator*, as shown in Figure 6: internal and external event notifications from the *coordinator* to the *simulator*, and output from the *simulator* to the *coordinator*.

Each *simulator* and *coordinator* define the time when the next internal transformation occurs. In the *simulator*, the time is calculated by the time advance function of the atomic model. In the *coordinator*, the next event occurrence time of all child nodes is obtained and the minimum value is taken as its next event occurrence time. Therefore, the next event time of the *root-coordinator* is the minimum of the following event times of all models in the system. The *root-coordinator* continuously advances the global time to correspond to the event occurrence time and sends event messages to its child nodes to notify them to execute the event. Then this

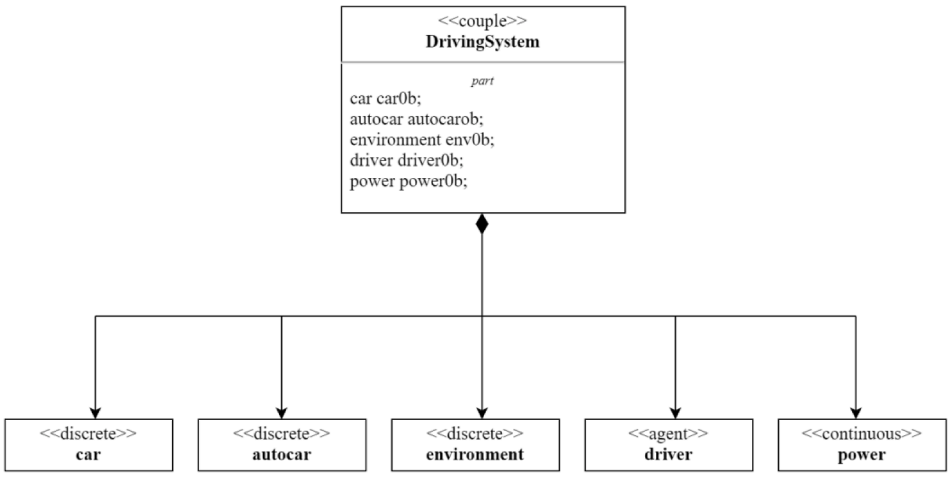


Figure 9. Definition diagram of the top-level model

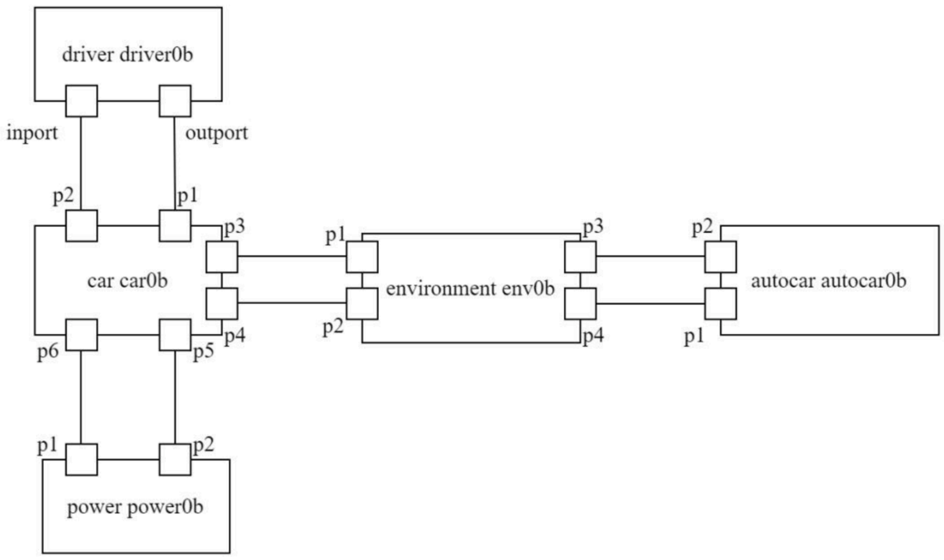


Figure 10. Connection diagram of the top-level model

```
couple simulation
      import car;
      import autocar;
      import environment;
      import driver;
      import power;

part:
      car carOb;
      autocar autocarOb;
      environment envOb;
      driver driverOb;
      power powerOb;

connection:
      connect(driverOb.outport, carOb.p1);
      connect(carOb.p2, driverOb.inport);

      connect(carOb.p3, envOb.p1);
      connect(envOb.p2, carOb.p4);

      connect(carOb.p6, powerOb.p1);
      connect(powerOb.p2, carOb.p5);

      connect(envOb.p4, autocarOb.p1);
      connect(autocarOb.p2, envOb.p3);

end;
```

Figure 11. Modeling text of the top-level model

cycle repeats until the simulation ends.

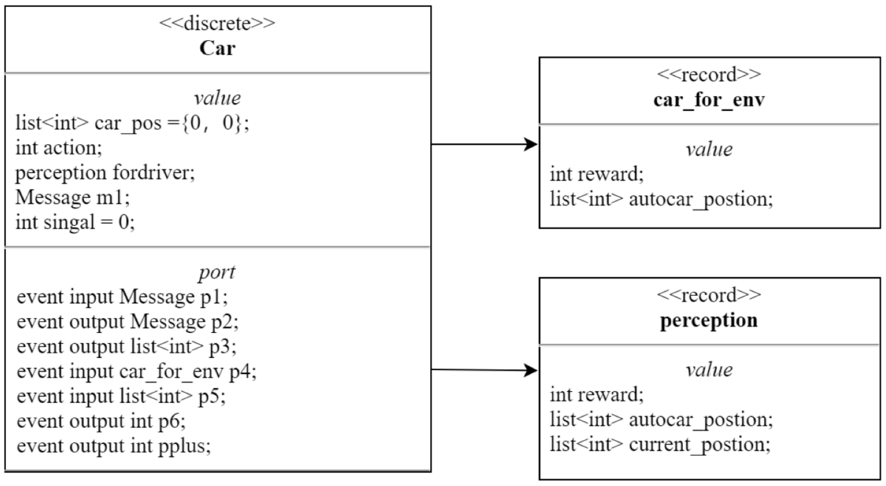


Figure 12. Definition diagram of the car

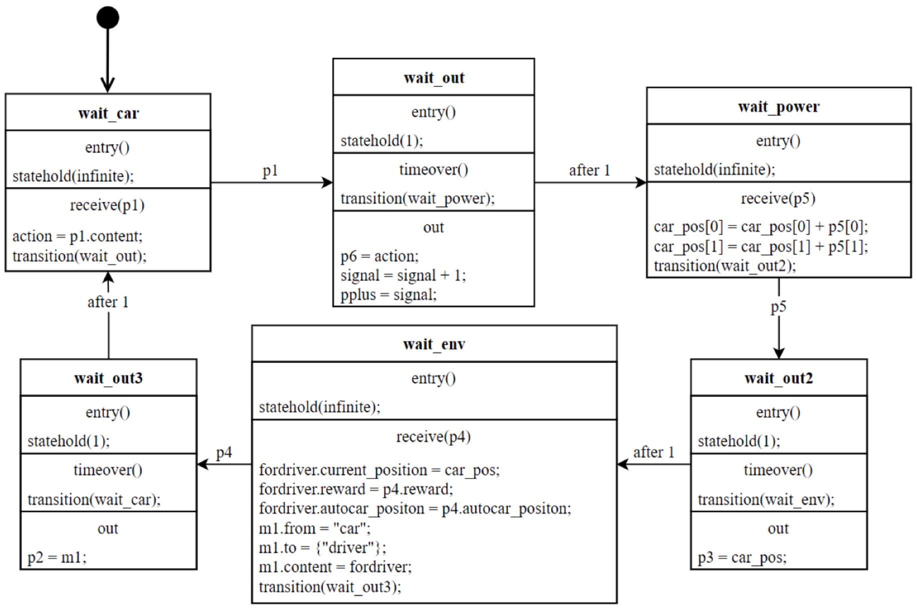


Figure 13. State Machine diagram of the car

6. Case study

An intelligent car driving system is illustrated here to prove the simulation capability of X-language for hybrid systems, which means the hybrid of a continuous system and discreet event system. The system is simplified into three modules: continuous module, discrete event module, and agent module. Figure 7 shows the structure of the system, where *Car1* is a manned vehicle, regarded as an intelligent car, and *Car2* is an unmanned vehicle, as a non-intelligent car. *Car1* contains three modules: *driver* as an agent model, *car* as a discrete event model, and *power* as a continuous model. *Car2* only has a discrete event model named *autocar*.

In order to show more clearly the entire modeling and simulation process based on X language, the connection relationship and action sequence among various models of the system are explicitly defined in Figure 7. The agent model (*driver*) first uses the trained policy to make decisions and passes the action results to the discrete event model (car). The *car* transmits the driver's actions to the *power* module to generate dynamic feedback based on the actions. Finally, the *car* changes its actual position according to the *power* feedback. During this process, the *environment* also keeps track of the positions of *Car2* and feeds them back to *Car1* so that *Car1* will make way for *Car2* to prevent accidents.

The dynamic diagram of the system is shown in Figure 8, where (a), (b), and (c) show the three interaction stages that *Car1* and *Car2* experience on the way to the destination. At each encounter, *Car1* will stop its current action and give way to Car2 until *Car2* leaves. The entire trajectory of *car1* from the initial position to the final position is illustrated in (d). In this process, *Car1* traverses the t1-t8 stages shown in Figure 7.

For this case, the top-level model of the entire system is firstly established, which is composed of the *car, autocar, environmental, driver*, and *power* module. The definition diagram and connection diagram of the system are shown in Figures 9 and 10, respectively defining the components and the connection relationship among the above 5 modules. Figure 11 indicates the modeling text of the top-level model. Obviously, Figure 9 and Figure 10 are the graphic forms of the couple class, and Figure 11 is the text forms; they all represent the same models. Similarly, Figures 12 and 13 are, respectively, the definition diagram and state machine diagram of the *car*. Figure 14 shows the modeling text of the *car*. In view of the fact that the modeling process is the same, the

```
discrete car
import car_for_env;
import perception;
value:
    list<int> car_pos = {0，0};
    int action;
    perception fordriver;
    Message m1;
    int signal = 0;

port:
    event input Message p1;
    event output Message p2;

    event output list<int> p3;
    event input car_for_env p4;

    event input list<int> p5;
    event output int p6;
    event output int pplus;

state:
    initial state wait_car
        when entry() then
            statehold(infinite);
        end;
        when receive(p1) then
            action = p1.content;
            transition(wait_out);
        end;
    end;

    state wait_out
        when entry() then
            statehold(1);
        end;
        when timeover() then
            transition(wait_power);
        out
            p6 = action;
            signal = signal + 1;
            pplus = signal;
        end;
    end;

    state wait_power
        when entry() then
            statehold(infinite);
        end;
        when receive(p5) then
            car_pos[0] = car_pos[0] + p5[0];
            car_pos[1] = car_pos[1] + p5[1];
            transition(wait_out2);
        end;
    end;

    state wait_out2
        when entry() then
            statehold(1);
        end;
        when timeover() then
            transition(wait_env);
        out
            p3 = car_pos;
        end;
    end;

    state wait_env
        when entry() then
            statehold(infinite);
        end;
        when receive(p4) then
            fordriver.current_position = car_pos;
            fordriver.reward = p4.reward;
            fordriver.autocar_positon=p4.autocar_positon;
            m1.from = "car";
            m1.to = {"driver"};
            m1.content = fordriver;
            transition(wait_out3);
        end;
    end;

    state wait_out3
        when entry() then
            statehold(1);
        end;
        when timeover() then
            transition(wait_car);
        out
            p2 = m1;
        end;
    end;
end;
```

Figure 14. Modeling text of the car model

other system models will not be described in detail. It should be noted that the modeler can use X language-specific tool XLab to realize the automatic transformation of the graphic model to the text model or can directly perform the text modeling and skip the graphic modeling according to the actual needs.

Since X language simulation engine is built based on DEVS, there is a time advancement process of discrete events during the simulation of the system, as shown in Figure 15, where a point represents an event occurrence of each DEVS-based atomic model. The abscissa represents the simulation advancing time, and the ordinate represents each atomic model. When time=0, the *driver goal* first output a signal to the *driver watchfor* plan. Then the *driver makedecision ext* model is activated to generate an agent action-decision for the car, thus activating the *car* model. Afterward, the *car* model outputs a signal to the *power* model, whose actions are continuous as shown in a line in Figure 15. The *power* model returns a signal to the *car*, and the *car* outputs a signal to the *environment* at the same time. The processes described above are the signal transition processes between different models in the car system.

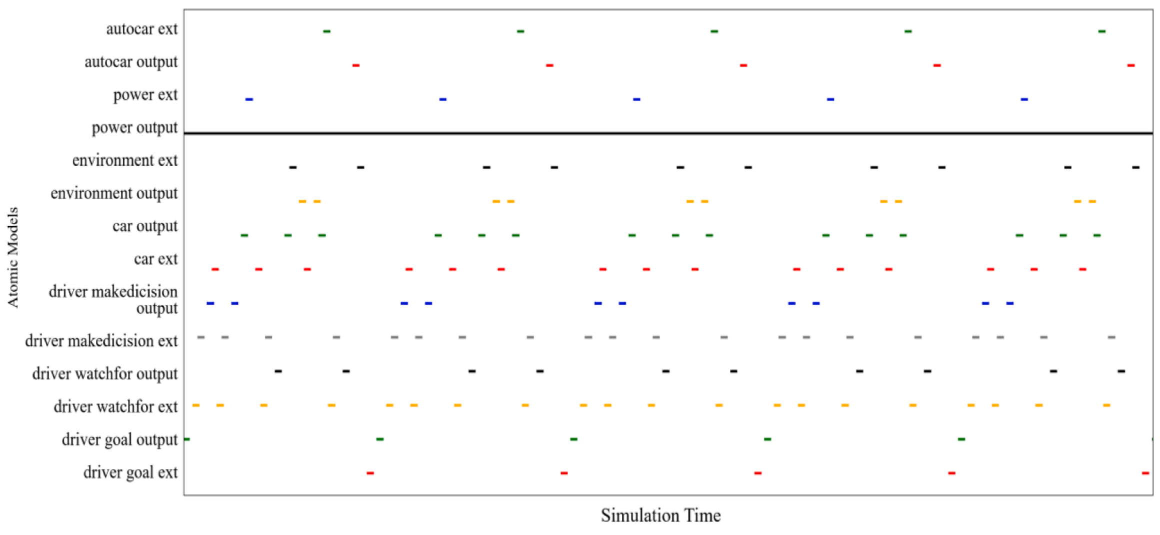


Figure 15. Simulation time series of each atomic model of the system

Table 11
Capability comparison of X language and other modeling languages

Capability	Languages XLaguage	SysML [49]	AADL [50]	Modelica [51, 52]	KARMA [53–56]	SIMAN [57]
Multi-view Modeling	★★①	★★	★	—	★★	—
Qualitative Modeling	★★	★★	★	★	★★	—
Simulation Execution	★★	—	★	★★	★	★
Integrated Modeling and Verification②	★★	—	★	★	—	—
Continuous System Modeling	★★	★	★	★★	★★	★
Discrete Event Modeling	★★	★	★★	★	★★	★★
Hybrid System Modeling	★★	★	★	★	★★	★
Multi-agent modeling	★★	★	★	★	★	—

① "-" means do not have this capability, "★" means have this capability but not enough, "★★" means good at this capability
② Models can be simulated directly to verify whether the performance meets stakeholder needs and objectives, without resorting to model transformation.

It can be found from Figure 15 that the running sequence of the entire model is consistent with the modeling sequence of Figure 7, presenting the simulation process of the couple model composed of continuous, discrete, and agent models. In this case, the *power output* is continuous throughout the simulations and interacts with the *car* when the interaction conditions are met. Due to the discrete nature, the agent model has a consistent simulation process with the operation of the discrete model.

This case shows the modeling processes of X language in multi-domain models and demonstrates the support of X language for the continuous, discrete event, and agent models from the perspective of modeling and simulation.

7. Conclusion

MBSE transforms the traditional R&D method based on documents and physical models into a model-driven R&D method, which renders MBSE reusable, unambiguous, intelligible, and easy to spread, enabling it to be an important tool in supporting system modeling and development. As the name suggests, models are the foundation of MBSE, and how to ensure the accuracy of the model has become an important research content. However, the modeling language SysML employed by MBSE needs to corporate with physical modeling languages rather than directly verifying the correctness and completeness of the model, which makes it challenging to ensure the consistency and traceability of the whole system modeling and simulation process. This method also lacks the support for intelligent products in modeling and simulation.

In response to this problem, this paper introduces a new integrated modeling and simulation language and the corresponding compiler and simulation engine developed by the authors' team that supports MBSE. X language has two modeling forms, namely graphics and text, and can be converted to each other through XLab. It is able to support both system modeling and physical simulation, ensuring that a unified language can run through the entire process of architecture design, multi-domain modeling, and simulation verification, thus realizing multi-disciplinary and cross-staged collaborative modeling and simulation of complex products. In addition, the agent class adding to the language enables X language not only to realize the modeling of continuous, discrete, and hybrid models, but also to support the modeling of various complex agent models, thereby giving the language the capability of intelligence modeling. Table 11 shows the comprehensive capabilities of X language through comparison with other mainstream modeling languages.

In Table 11, the mainstream modeling languages are compared from the perspectives of design, simulation, and verification, from which it can be seen that the existing languages have their own advantages in some aspects, but fails to independently complete the whole process of MBSE. For example, SysML[49] is good at top-level architecture modeling, Modelica[50] is dedicated to multi-domain physical system simulation, and AADL[50] focuses on safety analysis for embedded systems, each of which has its capability shortcomings. It is worth noting that a new modeling language recently proposed, KARMA[53, 54], can support the unified formalisms across MBSE models and simulations for different domain-specific models and performs well in each function listed in Table 11. However, as an architecture-driven technology-based language, the KARMA language still realizes verification through model transformation. In the follow-up study[55], the syntax of hybrid automata is integrated into KARMA to describe the behavior models more precisely and facilitate verification, which is an improvement. In contrast, models built in X language can be directly simulated to verify whether the performance associated with the system meets stakeholder needs and objectives without resorting to model transformation, which genuinely achieves the integration of modeling and verification by using a unified language.

So far, a large number of models and systems built in X language have been simulated and verified, just as the intelligent car system in this paper, fully validating the effectiveness of the proposed approach. To better support MBSE, for one thing, many development efforts still have to be done for XLab, including 1) developing model libraries for different industrial applications, 2) integrating with more software used in the lifecycle of product development, e. g. Computer-Aided Design (CAD), Computer-Aided Engineering (CAE), Software Engineering tools, 3) improving the compatibility with current modeling and/or simulation languages, 4) enriching software interfaces to improve compatibility with other software, such as support for FMI (Functional Mock-up Interface), 5) cloudification and servitization of XLab, etc. For another thing, A lot of research related to X language will be conducted, for example, X language-based comprehensive optimization for complex product design, digital twin construction with X language, X language-based multi-scale and multi-view modeling, model composition, and reuse with X language so on.

CRediT authorship contribution statement

Lin Zhang: Conceptualization, Methodology, Resources, Supervision, Funding acquisition, Writing – original draft, Writing – review & editing. **Fei Ye:** Conceptualization, Writing – original draft, Writing – review & editing, Project administration. **Kunyu Xie:** Software, Validation, Writing – original draft. **Pengfei Gu:** Software, Data curation, Writing – original draft, Visualization. **Xiaohan Wang:** Validation, Investigation, Writing – original draft. **Yuanjun Laili:** Conceptualization, Methodology, Formal analysis. **Chun Zhao:** Software, Validation. **Xuesong Zhang:** Conceptualization, Software. **Minjie Chen:** Methodology, Conceptualization. **Tingyu Lin:** Methodology, Conceptualization. **Zhen Chen:** Methodology, Software.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported by the National Key R&D Program of China, No.2018YFB1701600.

Appendix

Appendix A

BNFs for components of classes are shown in Figure 16, 17, 18, 19, 20, 21.

```
definition_section ::={(import_clause | extends_clause
        | class_definition
        | parameter_component_clause)';'}
        { port_section
        | part_section
        | value_section
        | plan_definition}
import_clause ::= 'import' (IDENT '=' name | name ('.' ( '*' | '{' import_list '}' ) )? )
extends_clause ::= 'extends' type_specifier [class_modification]
class_definition    ::= ('encapsuate')?    class_prefixes class_specifier
parameter_component_clause ::= 'parameter' type_specifier component_list
port_section ::= 'port:'{port_component_clause}
part_section ::= 'part:'{component_clause';'}
value_section ::= 'value:'{component_clause';'}
import_list ::= IDENT  {',' IDENT}
component_clause ::=['replaceable'] type_prefix type_specifier component_list
```

Figure 16. BNF of Definition Part

```
requirement_section ::= ('requirement'| 'traceble requirement') {statement';'}
statement ::= {traceLink types} {requirement name} {requirement contents}
traceLink types ::= compose|copy|derivedFrom|refinedBy|satisfiedBy|verifiedBy
        |mappedTo
        |originatedFrom
        |responsibleOf
requirement attributes::= identifier : id
        |name
        |level: Level
        |type: Type
        |risk: Risk
        |source
        |stakehodler
        |text: description
Level ::= Stakeholder|System|Component
Type ::= General|Functional|NonFonctional|Physical|Design
Risk ::= High|Medium|Low
```

Figure 17. BNF of Requirement Part

```
connection_section ::= 'connection:'{connect_clause';'}
connect_clause ::= 'connect' '(' component_reference ',' component_reference ')'
```

Figure 18. BNF of Connection Part

```
state_section ::= 'state:' {state_definition}
state_definition ::= 'initial' 'state' IDENT    (state_statement)*    'end'';'
        | 'state' IDENT    (state_statement)*    'end'';'
        | 'state' IDENT    catch_clause    ((when_receive_clause';')|(when_statement';'))*
'end'';'
state_statement ::= (when_entry_clause
        | when_receive_clause
        | when_goto_out_clause)';'
```

Figure 19. BNF of State Machine Part

```
equation_section ::= 'equation:' {equation ';'}
equation ::= simple_equation
        | if_equation
        | for_equation
        | when_receive_equation
        | when_equation
```

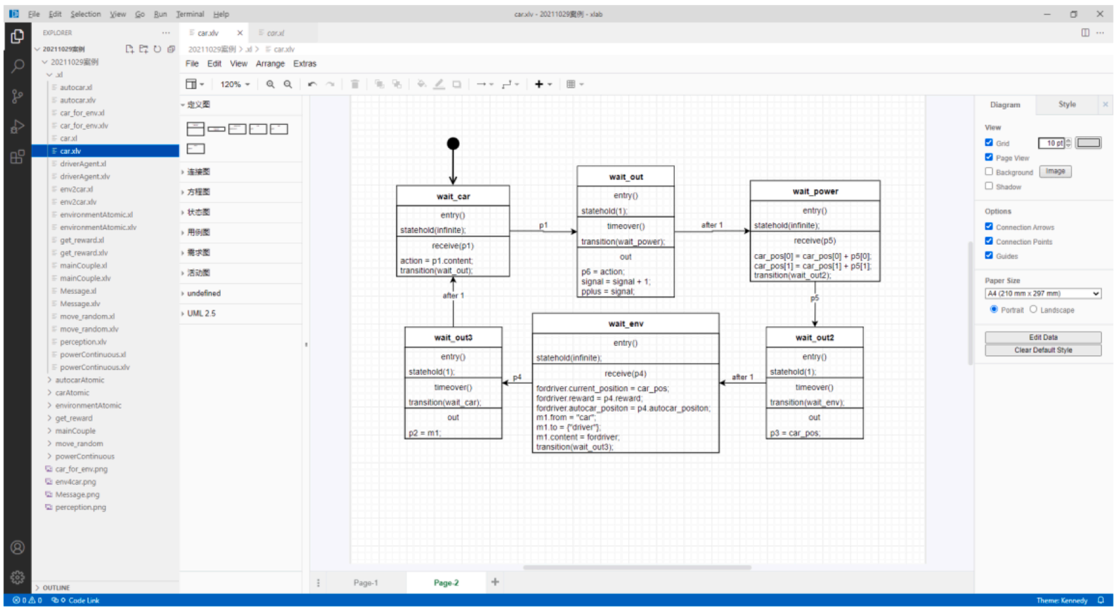
Figure 20. BNF of Equation Part

```
action_section ::= 'action:' {statement';'}
statement ::= send_clause
         | simple_statement
         | function_call
         | break_statement
         | continue_statement
         | return_statement
         | if_statement
         | for_statement
         | while_statement
         | when_statement
         | statehold_clause
         | run_statement
         | agentover_statement
         | transition_clause
```

Figure 21. BNF of Action Part

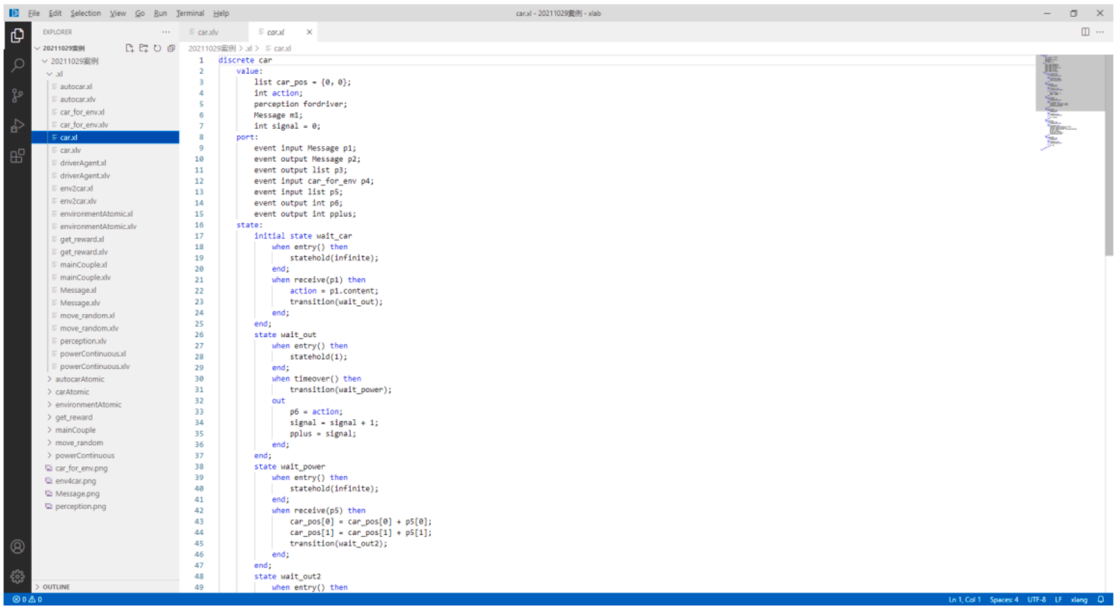
Appendix B

The processes of modeling and simulation using XLab are shown in Figure 22, 23.



```
wait_car
entry()
statehold(infinite);
receive(p1)
action = p1.content;
transition(wait_out);

wait_out
entry()
statehold(1);
timeover()
transition(wait_power);
out
p6 = action;
signal = signal + 1;
pplus = signal;

wait_power
entry()
statehold(infinite);
receive(p5)
car_pos[0] = car_pos[0] + p5[0];
car_pos[1] = car_pos[1] + p5[1];
transition(wait_out2);

wait_out2
entry()
statehold(1);
timeover()
transition(wait_env);
out
p3 = car_pos;

wait_env
entry()
statehold(infinite);
receive(p4)
fordriver.current_position = car_pos;
fordriver.reward = p4.reward;
fordriver.autocar_positon = p4.autocar_positon;
m1.from = "car";
m1.to = {"driver"};
m1.content = fordriver;
transition(wait_out3);

wait_out3
entry()
statehold(1);
timeover()
transition(wait_car);
out
p2 = m1;
```

Figure 22. State Machine diagram of the car in XLab



```
discrete car
    value:
        list car_pos = {0, 0};
        int action;
        perception fordriver;
        Message m1;
        int signal = 0;
    port:
        event input Message p1;
        event output Message p2;
        event output list p3;
        event input car_for_env p4;
        event input list p5;
        event output int p6;
        event output int pplus;
    state:
        initial state wait_car
            when entry() then
                statehold(infinite);
            end;
            when receive(p1) then
                action = p1.content;
                transition(wait_out);
            end;
        end;
        state wait_out
            when entry() then
                statehold(1);
            end;
            when timeover() then
                transition(wait_power);
            out
                p6 = action;
                signal = signal + 1;
                pplus = signal;
            end;
        end;
        state wait_power
            when entry() then
                statehold(infinite);
            end;
            when receive(p5) then
                car_pos[0] = car_pos[0] + p5[0];
                car_pos[1] = car_pos[1] + p5[1];
                transition(wait_out2);
            end;
        end;
        state wait_out2
            when entry() then
```

Figure 23. State Machine text of the car in XLab

References

[1] B P Zeigler, L. Zhang, Service-oriented model engineering and simulation for system of systems engineering[M]//Concepts and Methodologies for Modeling and Simulation, Springer, Cham, 2015, pp. 19–44.

[2] A L Ramos, J V Ferreira, J. Barceló, Model-based systems engineering: An emerging approach for modern systems[J], IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews) 42 (1) (2011) 101–111.

[3] L Zhang, Y Liu, Y Laili, et al., Model maturity towards modeling and simulation: Concepts, index system framework and evaluation method[J], International Journal of Modeling, Simulation, and Scientific Computing 11 (03) (2020), 2040001.

[4] M. Hause, OMG systems modeling language (OMG SysML™) tutorial[C]//INCOSE international symposium 19 (1) (2009) 1840–1972.

[5] G Shao, H Latif, C Martin-Villalba, et al., Standards-based integration of advanced process control and optimization[J], Journal of Industrial Information Integration 13 (2019) 1–12.

[6] L Zhang, F Ye, Y Laili, et al., X language: an integrated intelligent modeling and simulation language for complex products[C]. 2021 Annual Modeling and Simulation Conference (ANNSIM), IEEE, 2021, pp. 1–11.

[7] B P Zeigler, A Muzy, E. Kofman, Theory of modeling and simulation: discrete event & iterative system computational foundations[M], Academic Press, 2018, pp. 19–23.

[8] D D Walden, G J Roedler, K. Forsberg, INCOSE systems engineering handbook version 4: Updating the reference for practitioners[C], INCOSE International Symposium 25 (1) (2015) 678–686.

[9] A A Kerzhner, J M Jobe, CJJ. Paredis, A formal framework for capturing knowledge to transform structural models into analysis models[J], Journal of Simulation 5 (3) (2011) 202–216.

[10] E Huang, R Ramamurthy, L F McGinnis, System and simulation modeling using SysML[C]. 2007 Winter Simulation Conference, IEEE, 2007, pp. 796–803.

[11] O Schönherr, O. Rose, First steps towards a general SysML model for discrete processes in production systems[C]. Proceedings of the 2009 Winter Simulation Conference (WSC), IEEE, 2009, pp. 1711–1718.

[12] O Batarseh, LF. McGinnis, System modeling in SysML and system analysis in Arena [C]. Proceedings of the 2012 Winter Simulation Conference (WSC), IEEE, 2012, pp. 1–12.

[13] C J J Paredis, Y Bernard, R M Burkhart, et al., 5.5. 1 An overview of the SysML-Modelica transformation specification[C], INCOSE International Symposium 20 (1) (2010) 709–722.

[14] M Nikolaidou, G D Kapos, A Tsadimas, et al., Simulating SysML models: Overview and challenges[C]. 2015 10th System of Systems Engineering Conference (SoSE), IEEE, 2015, pp. 328–333.

[15] M Bajaj, D Zwemer, R Peak, et al., Slim: collaborative model-based systems engineering workspace for next-generation complex systems[C]. 2011 Aerospace Conference, IEEE, 2011, pp. 1–15.

[16] R S Peak, R M Burkhart, S A Friedenthal, et al., 9.3.2 Simulation-based design using SysML Part 1: A parametrics primer[C], INCOSE international symposium 17 (1) (2007) 1516–1535.

[17] R Wang, CH. Dagli, An executable system architecture approach to discrete events system modeling using SysML in conjunction with colored Petri Net[C]. 2008 2nd annual IEEE systems conference, IEEE, 2008, pp. 1–8.

[18] G D Kapos, V Dalakas, A Tsadimas, et al., Model-based system engineering using SysML: Deriving executable simulation models with QVT[C]. 2014 IEEE International Systems Conference Proceedings, IEEE, 2014, pp. 531–538.

[19] H M Paynter, Analysis and design of engineering systems[M], Cambridge MIT Press, 1961.

[20] M D Jenny, D Marisol, B Claude, A survey of bond graphs: theory, applications and programs[J], Journal of the Franklin Institute 328 (4) (1991) 565–606.

[21] P Fritzson, P. Bunus, Modelica-a general object-oriented language for continuous and discrete-event system modeling and simulation[C]. Proceedings 35th Annual Simulation Symposium. SS 2002, IEEE, 2002, pp. 365–380.

[22] P. Fritzson, Introduction to modeling and simulation of technical and physical systems with Modelica[M], John Wiley & Sons, 2011.

[23] J Nutaro, P T Kuruganti, V Protopopescu, et al., The split system approach to managing time in simulations of hybrid systems having continuous and discrete event components[J], Simulation 88 (3) (2012) 281–298.

[24] H Elmqvist, F Gaucher, S E Matsson, et al., State machines in Modelica[C], in: Proceedings of the 9th International Modelica Conference; September 3-5, 2012, Linköping University Electronic Press, Munich; Germany, 2012, pp. 37–46.

[25] Beltrame T, Cellier F E. Quantised state system simulation in Dymola/Modelica using the DEVS formalism[C] Proceedings 5th International Modelica Conference. The Modelica Association, 2006: 73-82.

[26] F Bünning, R Sangi, D. Müller, A Modelica library for the agent-based control of building energy systems[J], Applied energy 193 (2017) 52–59.

[27] V Sanz, F Bergero, A. Urquia, An approach to agent-based modeling with Modelica [J], Simulation Modelling Practice and Theory 83 (2018) 65–74.

[28] Aertgeerts A, Claessens B, De Coninck R, et al. Agent-based control of a neighborhood: a generic approach by coupling Modelica with Python[C] Proceedings of Building Simulation 2015. 2015.

[29] A Schaub, M Hellerer, T. Bodenmüller, Simulation of artificial intelligence agents using Modelica and the DLR visualization library[C]. 9th International Modelica Conference, Linköping University Electronic Press, Munich, Germany, 2012, pp. 339–346.

[30] Z Sha, Q Le, J H Panchal, Using SysML for conceptual representation of agent-based models[C], International Design Engineering Technical Conferences and Computers and Information in Engineering Conference 54792 (2011) 39–50.

[31] A Maheshwari, C R Kenley, D A DeLaurentis, Creating executable agent-based models using SysML[C], INCOSE international symposium 25 (1) (2015) 1263–1277.

[32] Zhang M. Constructing a cognitive agent model using DEVS framework for multi-agent simulation[J]. Proc. 15th Eur. Agent Syst. Summer School (EASSS), 2013: 1-5.

[33] Akplogan M, Quesnel G, Garcia F, et al. Towards a deliberative agent system based on DEVS formalism for application in agriculture[C] Proceedings of the 2010 Summer Computer Simulation Conference. 2010: 250-257.

[34] J P Müller, Towards a formal semantics of event-based multi-agent simulations[C]. International Workshop on Multi-Agent Systems and Agent-Based Simulation, Springer, Berlin, Heidelberg, 2008, pp. 110–126.

[35] W. Schamai, Modelica modeling language (ModelicaML): A UML profile for Modelica[M], Linköping University Electronic Press, 2009.

[36] Schamai W, Pohlmann U, Fritzson P, et al. Execution of umlstate machines using Modelica[C] 3rd International Workshop on Equation-Based Object-Oriented

Modeling Languages and Tools; Oslo; Norway; October 3. Linköping University Electronic Press, 2010 (047): 1-10.

[37] J M Gauthier, F Bouquet, A Hammad, et al., Tooled process for early validation of SysML models using Modelica simulation[C]. International Conference on Fundamentals of Software Engineering, Springer, Cham, 2015, pp. 230–237.

[38] Paredis C J J, Bernard Y, Burkhart R M, et al. 5.5. 1 An overview of the SysML-Modelica transformation specification[C] INCOSE International Symposium. 2010, 20(1): 709-722.

[39] Y Cao, Y Liu, H Fan, et al., SysML-based uniform behavior modeling and automated mapping of design and simulation model for complex mechatronics[J], Computer-Aided Design 45 (3) (2013) 764–776.

[40] B H Li, X Song, L Zhang, et al., CoSMSOL: Complex system modeling, simulation and optimization language[J], International Journal of Modeling, Simulation, and Scientific Computing 8 (02) (2017), 1741002.

[41] Li X G, Liu J H. A method of SysML-based visual transformation of system design-simulation models[J]. Journal of Computer-Aided Design & Computer Graphics, 2016: 11.

[42] S H Zhou, Y Cao, Z Zhang, et al., System design and simulation integration for complex mechatronic products based on SysML and Modelica[J], Journal of Computer-Aided Design & Computer Graphics. Beijing 30 (2014) 728–738.

[43] G. Berry, SCADE: Synchronous design and validation of embedded control software [M]. Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems, Springer, Dordrecht, 2007, pp. 19–33.

[44] SJI Herzig, CJJ. Paredis, A conceptual basis for inconsistency management in model-based systems engineering[J], Procedia Cirp 21 (2014) 52–57.

[45] S Feldmann, S J I Herzig, K Kernschmidt, et al., Towards effective management of inconsistencies in model-based engineering of automated production systems[J], IFAC-PapersOnLine 48 (3) (2015) 916–923.

[46] Jongeling R, Cicchetti A, Ciccozzi F, et al. Towards boosting the OpenMBEE platform with model-code consistency[C] Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings. 2020: 1-5.

[47] Berriche A, Mhenni F, Mlika A, et al. Towards model synchronization for consistency management of mechatronic systems[J]. Applied Sciences, 2020, 10 (10): 3577.

[48] S Haidrar, H Bencharqui, A Anwar, et al., REQDL: a requirements description language to support requirements traces generation[C]. 2017 IEEE 25th International Requirements Engineering Conference Workshops (REW), IEEE, 2017, pp. 26–35.

[49] M. Hause, The SysML modelling language[C], Fifteenth European Systems Engineering Conference 9 (2006) 1–12.

[50] P H Feiler, D P Gluch, J J Hudak, The architecture analysis & design language (AADL): An introduction[R], Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst, 2006.

[51] P Fritzson, V. Engelson, Modelica—A unified object-oriented language for system modeling and simulation[C]. European Conference on Object-Oriented Programming, Springer, Berlin, Heidelberg, 1998, pp. 67–90.

[52] F E Cellier, V. Sanz, Mixed quantitative and qualitative simulation in Modelica[C]. Proceedings of the 7th International Modelica Conference, Linköping University Electronic Press, Como; Italy, 2009, pp. 86–95.

[53] J Lu, G Wang, J Ma, et al., General modeling language to support model-based systems engineering formalisms (part 1)[C], INCOSE International Symposium 30 (1) (2020) 323–338.

[54] J Guo, G Wang, J Lu, et al., General modeling language supporting model transformations of mbse (part 2)[C], INCOSE International Symposium 30 (1) (2020) 1460–1473.

[55] Ding J, Reniers M, Lu J, et al. Integration of modeling and verification for system model based on KARMA language[C] Proceedings of the 18th ACM SIGPLAN International Workshop on Domain-Specific Modeling. 2021: 41-50.

[56] Chen J, Wang G, Lu J, et al. Model-based system engineering supporting production scheduling based on satisfiability modulo theory[J]. Journal of Industrial Information Integration, 2022: 100329.

[57] Pegden C D. Introduction to SIMAN[C] Proceedings of the 17th conference on Winter simulation. 1985: 66-72.