X Language Modeling Specification 3.0

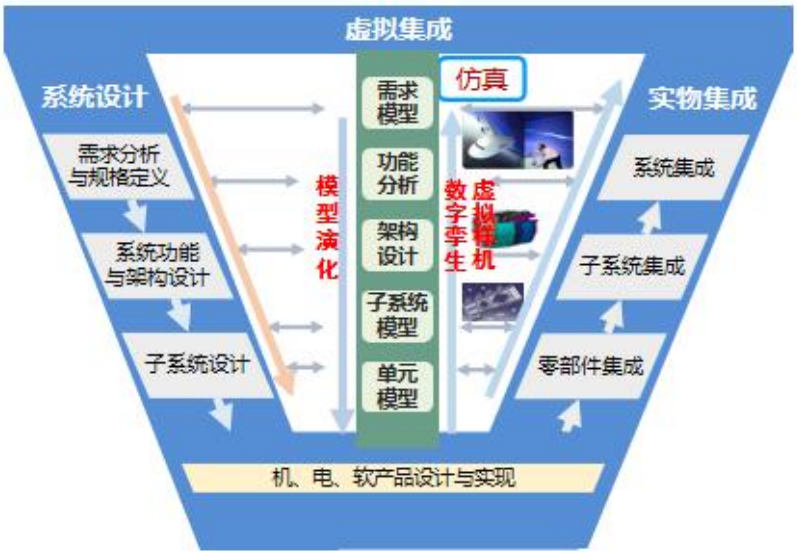
September 2023

CONTENTS

1 X Language Overview

1.1 Background of X language development

In recent years, Model-based Systems Engineering (MBSE) has become an important tool to support system modeling and development.The core idea of MBSE is to support the system from conceptual design, analysis, validation to the development of the whole life cycle of all phases of the system by means of a unified, formalized, and standardized model, so that the information exchange between engineers can be changed from the traditional document-based and physical model-driven R&D mode to a model-driven R&D mode. The information exchange between engineers is transformed from the traditional R&D mode driven by documents and physical models to a model-driven R&D mode. Currently, the mainstream modeling language supporting MBSE is SysML (System Modeling Language), which is developed by INCOSE in collaboration with OMG on the basis of Unified Modeling Language (UML) for describing engineering systems. However, since SysML lacks the ability to describe the physical model of a product. The system integration stage still adopts the physical system integration approach, which will lead to inefficient development, high cost, and great limitations. To completely change the traditional R&D mode and realize the maximum value of MBSE, it is also necessary to transform the integration verification process based on physical prototypes into a process based on digital prototypes, i.e., based on modeling and simulation-based simulation system engineering (MBSE) with the help of simulation technology, which can really shorten the R&D cycle, reduce costs, improve efficiency, make the whole R&D process easy to trace and easy to maintain; realize the complex product The development of complex products required by the success of a manufacturing.



At present, there are two mainstream implementation methods, method one is for the integration of mechanical, electrical, hydraulic, control type of complex products, first based on the system modeling language (such as SysML, IDEF, etc.) to carry out the requirements of the modeling and architectural design, and then based on the physical modeling language (such as Modelica, Matlab/Simulink, etc.) and with the integration of standards (FMI, HLA, etc.). The second method is to realize the integration of complex product system design and simulation by establishing the mapping relationship between system modeling languages (e.g., SysML, IDEF, etc.) and physical modeling languages (e.g., Modelica, Matlab/Simulink, etc.) to realize the automatic conversion of system models and physical models.

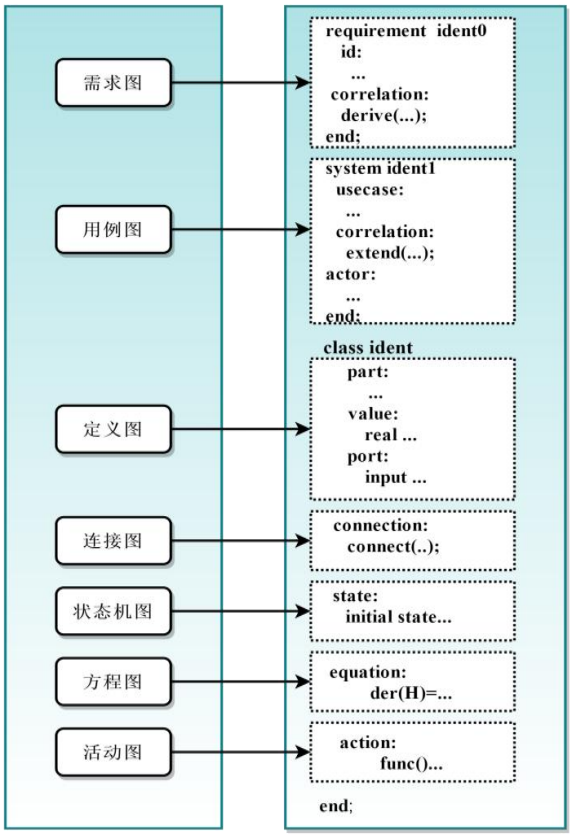
However, due to the different development backgrounds and object orientedness of system modeling languages (e.g., SysML, IDEF, etc.) and physical modeling languages (e.g., Modelica, Matlab/Simulink, etc.), the syntax and semantics of the two heterogeneous languages cannot be maintained consistently, and thus the mapping and conversion of the two languages are not completely carried out. In addition, the realization of a complete MBSE development process requires learning multiple languages, using multiple modeling platforms, establishing conversion rules between specific languages, etc., which greatly increases the learning cost of modelers.

Based on this, we propose a new generation of integrated modeling and simulation

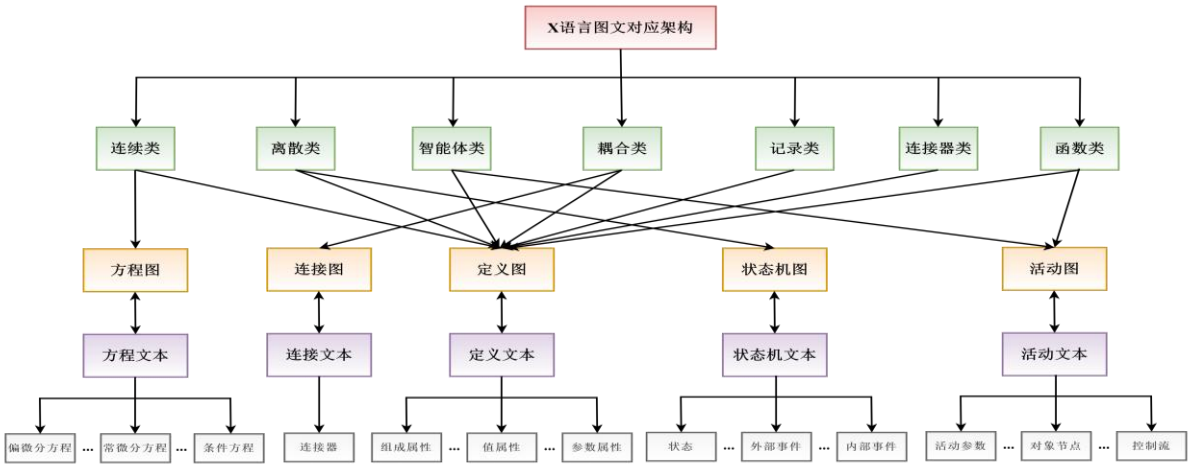
language for complex systems to support MBSE, X. X fully supports Model-Based Systems Engineering (MBSE), provides standardized graphical modeling descriptions in the conceptual design phase of a product, and automatically compiles the standardized graphical models into textual simulation models, which support full-system MBSE driven by a simulation engine. Driven by the simulation engine, it supports seamless and integrated simulation of the whole system, the whole process, and multiple perspectives, and realizes unified and integrated description and simulation from conceptual model design, system architecture design, multi-physical domain model to simulation model.

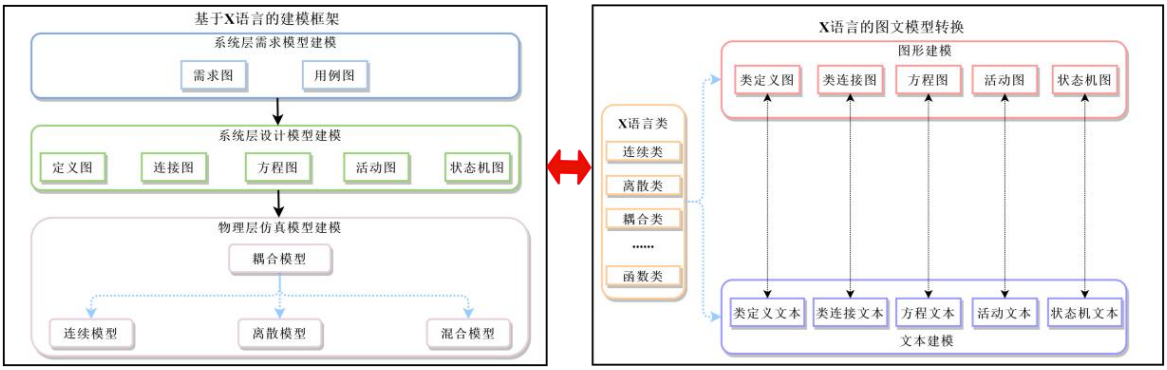
1.2 X-language core architecture

X-language is a new generation of integrated modeling and simulation language for complex systems to support MBSE, and its design goal is to provide an integrated modeling and simulation language that can realize the whole process of complex systems (requirements, design, validation, etc.), multi-domain (mechanical, electrical, hydraulic, control, etc.), multi-granularity (parts, components, equipment, subsystems, systems, and even systems), and multi-characteristics (continuous, discrete, hybrid, etc.).

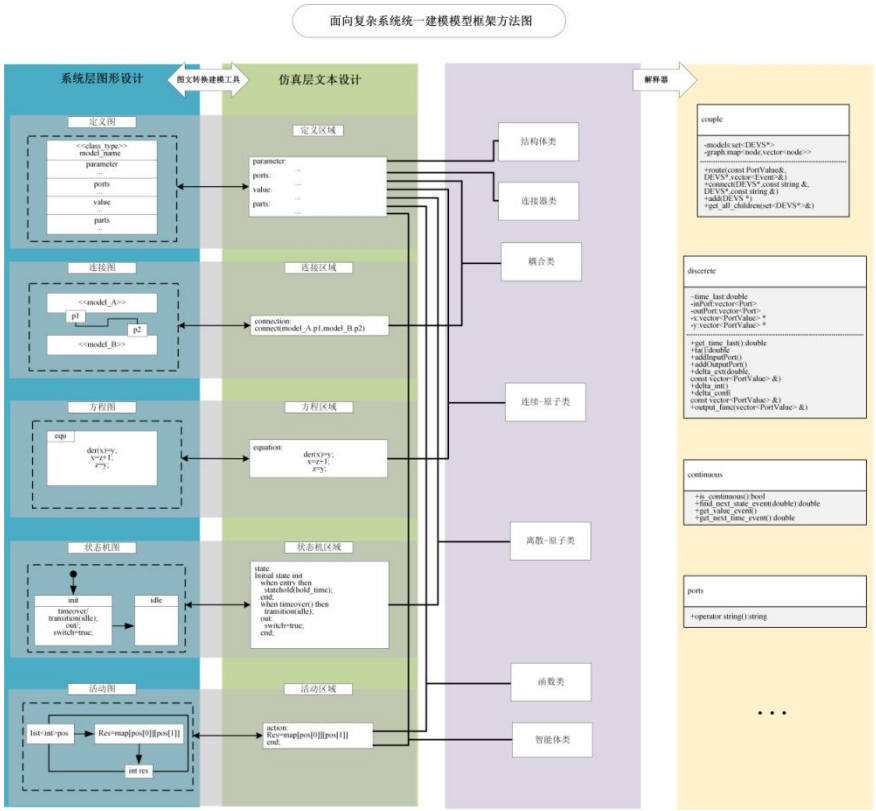


X is an object-oriented language that defines seven specific classes (continuous, discrete, coupled, etc.) for the description of multi-featured, multi-granular models. etc.), which can realize the description of multi-featured and multi-granularity models. Each class has two modeling forms, graphical and textual, which correspond to each other and can be converted to each other.





X language graphical modeling includes seven types of diagrams, namely: requirement diagram, use case diagram, definition diagram, connection diagram, equation diagram, activity diagram, and state machine diagram, in which the requirement diagram and use case diagram realize the establishment of the system requirement model and clarify the system functions; the definition diagram and the connection diagram realize the structural description of the model; and the equation diagram, the activity diagram, and the state machine diagram realize the description of the model's continuous, discrete, and the mixed behaviors of the continuous and discrete. X language completes the architectural modeling of the system model through definition diagrams and connection diagrams, and completes the description of the system functions and behaviors through equation diagrams, activity diagrams, and state machine diagrams, which can be converted into corresponding simulation texts, and then the model is interpreted into a simulatable file through the X language interpreter, and the performance of the system is verified by the X language simulation engine to achieve the integrated modeling and simulation of the complex system with full process, multi-domain, multi-granularity, multi-characteristics and multi-dimensionality. This enables the integrated modeling and simulation of complex systems with full process, multi-domain, multi-granularity and multi-features.



The modeling and simulation process of complex system in X language is shown in the figure above. After the modeler conducts the requirement analysis, he establishes the top-level architectural model of the system based on the definition diagram and connection diagram, and establishes the functional and behavioral models of different subsystems based on the equation diagram, state-machine diagram, and activity diagram; through the independently-developed XLab tool, the graphical model of the complex system will be converted into the simulation text, which can be sent to the interpreter directly. The generated simulation text can be directly sent to the interpreter, which, after receiving the simulation text, interprets and generates C++ code for different class files, and finally compiles and executes it on the DEVS-based simulator. Therefore, based on X-language, the integrated modeling and simulation of the whole process, multi-domain, multi-granularity and multi-characteristics of complex systems can be realized, making the whole R&D process easy to trace and easy to maintain, and really shortening the R&D cycle of complex products, reducing costs and improving efficiency.

2 X language keywords, operators, data types and built-in functions

2.1 Keywords

agentover	Intelligent Body Simulation End Marker	time	Current moment simulation time stamp
infinite	Infinity Labeling in Discrete Event Simulation	flow	Streaming variable identification
event	event identifier	input	Input port identification
output	Output Port Identification	elapsetime	Identifies how long the state has currently lasted
int	integer type	real	real type
bool	Boolean type	string	character type
while	Logic Loop Conditional Judgment Marker	then	Conditional Enforcement Flag
end	End qualification of chunks	send	Send Event/Message Functions
transition	Status Event Transfer Marker	couple	coupled class identifier
discrete	Discrete Class Marking	continuous	Continuous category marking
agent	real number identifier	when	Event Trigger Judgment Marker
der	Time Differential Marking	for	for loop identifier
connect	model port connection identifier	function	Function Definition Identifier
record	Special Data Structure	receive	Received event

	Definition Identification		occurrence marker
true	True	false	false
extend	Inheritance marking	import	External reference mark
state	Status Definition Identifier	timeover	State internal event trigger identifier
entry	Entry state event marker	statehold	Status duration setting
physical	Physical Communication Identifier	intelligent	Intelligent Algorithmic Communication Marker

2.2 Operators

+	plus sign	^	Multiplication	( )	Parentheses
-	minus sign	=	Assignment	[ ]	Access subscripts
*	Multiplication sign	,	Commas	.	Fetch field
/	Division sign	;	Semicolon	>、<、<=、>=	Comparator
++	Self-incrementing 1	--	Self-decreasing by 1	==	Constant Equation
.*	Matrix multiplication	./	Matrix division		

2.3 Data types

2.3.1 int

Description: Integer variable, e.g. -1, 0, 1

Declaration: int a = 1;

Methods: +, -, *, /.

2.3.2 real

Description: real (floating-point variables), such as: -0.01, 0.1, 7.996

Declaration: real a = 1.1;

Methods: +, -, *, /.

2.3.3 string

Description: string type, such as "a bsfsfsfs", "string"

Declaration: string a = "temp";

Methods:

1)determine the length a.size(), the return value is an unsigned integer;

2)string addition a = a + "eve", return value is string;

3)string index a[3], the return value is the corresponding position string.

2.3.4 bool

Description: Variables of true and false type, such as true, false (initial lowercase)

Declaration: bool a = true;

2.3.5 Arrays

Description: fixed-length array, consistent with the c language array

Declaration: int a[5] = [1,2,3,4,5], real b[1,1,2] = [[[1,1]]] ;)

Methods: access a[3] ;

2.3.6 list

Description: Indefinite-length data structure for storing variables of a given type, supporting more complex operations.

Declare: list<int> a = {1,2,3,4,5};

Methods:

1)add element at the end a.append();

2)list length (empty can be replaced by 0) a.size(), return value is unsigned integer;

3)list access a[3], the return value is the corresponding position element;

4) list insertion a.insert (int pos, int insertVarriable).

2.3.7 map

Description: A dictionary format for storing key-value pairs, e.g. {"a":1, "b":2}.

Declaration: map<string, int> a = {"a" :1}

Methods:

1)Dictionary access a["a"] = 1;

2)Dictionary length a.size(), return value is unsigned integer.

3)Dictionary add element a.add();

4)Dictionary delete element a.remove();

2.4 Built-in Functions

2.4.1 run(plan planName, …)

Used to run the plan of an intelligent body, the incoming plan will be executed sequentially.

2.4.2 send(message msg)/ send(outputport o1, outvalue v1)

1) Implement the message sending in the intelligent body's PLAN;
2) Implementing the output of the message in DEVS and sending the output value to the output port.

2.4.3 receive()/receive(inputport1,inputport2,inputport3...)

1) Implement message acceptance in intelligences;
2) Implementation of acceptance of events with defined ports in discrete classes, which may include multiple with that port parameters.

2.4.4 statehold(real time1)

The discrete class defines the duration of the state.

2.4.5 entry()

Defines the behavior that needs to be performed before entering the state.

2.4.6 timeover()

Defines the behavior that needs to be performed after the duration of the entry state has expired.

2.4.7 trasition(state s1)

Defines the target state for state transitions after the end of the entry state.

2.4.8 real random(real a,real b， int randtype)

Returns a randomized value between a, b in an as-you-go fashion for the chosen randtype.

2.4.9 void seed(real a)

Sets the random number seed, which must be used before using the random function.

2.4.10 connect(connector 1,connector2)/ connect(output,input)

1) When connecting two connectors the connection relationship is not sequential;
2) When connecting two discrete ports the output port is the first parameter and the target input port is the second parameter.

2.4.11 mathlib（function library）

real der(), the derivative function;

real sin(), the sine function;

real cos(), the cosine function;

real tan(), the tangent function;

real arccos(), the arcsine function;

real arcsin(), the inverse cosine function;

real arctan(), the inverse tangent function;

real sqrt(), the square root function;

int abs(), absolute value function;

int mod(), the remainder function;

real log(real a, real b), logarithmic function, returns the logarithm of b with a as the base;

real ln(real a), logarithmic function, returns the logarithm of a with e as the base.

3 Description specification of X language classes and external file imports

3.1 Continuous class

3.1.1 Introduction

The continuous atom class continuous models can be modeled. The continuous atom class itself is not re-splittable and is the smallest structural unit in the simulation.

3.1.2 Basic structure

Continuous classes often contain a header, a definition section, and an equation section.

The header section consists of two parts: the import part (structural keyword import) and the inheritance part (structural keyword extends). The import section imports the external class and the extends section inherits from the parent class.

The definition section describes the attributes of the continuous class, including the sections led by the parameter, value, and port structural keywords. The parameter part defines the constant attribute of the continuous model, and the value of the attribute after the equal sign is the value of the attribute, which will not be changed during the simulation and is a constant, such as the resistance value of a thermostatic resistor; the value part defines the variable of the model during the simulation, and if there is an equal sign after the variable, the initial value of the variable is the variable after the equal sign, and if the variable is constant, it can be preceded by the qualifier constant; the port part defines the port part of the model, i.e., the input/output part, which has two main forms of expression, one is the quantity modified by input/output as a qualifier, and the other is the instantiation of the connector class (the connector class will be described in more detail later).

The equation section describes the behavior of the continuous class. This section uses equation as the keyword, and the variables and constants used are those defined in the definition section. der() is used as a built-in differentiation function to represent the differentiation of variables, and if, for, and other constructs can be applied to this section to describe the behavior of the model. Note that the equal sign in the equation section indicates an equal constraint relationship in the equation and does not represent an assignment.

A definition template for the continuous class is given below:

```
continuous continuousName
    import out_continuous1;
    import out_function1;
    import out_record1;
    import out_connector1;
    extends out_continuous1;
parameter:
    datatype argname1;
    datatype argname2;
    out_record1 argname3; //Instantiate the imported out_record1 named argname3
    …
value:
    datatype varname1;
    datatype varname2;
    out_record1 varname3; //Instantiate the imported out_record1 named varname3
    …
port:
    input datatype argname1;
    output datatype argname2;
    out_connector1 argname3;//Instantiate the imported out_connector1 named argname3
    out_connector1 argname4;
    …
equation:
    out_function1();//Using the imported out_function1 function
    //This section describes the relationship between the variables of the model
```

3.1.3 Usage

In continuous classes, the model classes after import are often connector, function,

record, and continuous, where the imported continuous class can only be used for inheritance, the connector class is used for instantiation in the latter part of the port, the function class is used for equation building in the equation part, and the record class is used for parameter definition. extends can be used for inheritance of a model, and in continuous classes, the objects inherited can only be continuous classes. The continuous class can only be used for inheritance, the connector class is used for instantiation in the later port section, the function class is used for equation building in the equation section, and the record class is used for parameter definition. extends can be used for model inheritance, and in the case of the continuous class, the inherited objects can only be of the continuous class.

There are two types of calls to the continuous class. One is called and inherited by the continuous class and the other is called and instantiated as a component of the couple class.

3.1.4 Sample Code

The following example models a first-order system:

```
continuous Firstordersys
parameter:
    real k=8;
value:
    real z;
port:
    input real x=0;
    output real y;
equation:
    der(x)=z;
    y=z+k;
end;
```

3.2 Discrete classes

3.2.1 Introduction

The discrete atom class discrete class models discrete models. The discrete atom class is itself indivisible and is the smallest simulation unit for discrete models. Similar to the continuous atom class, the smallest simulation unit does not imply that the discrete atom class is only capable of building "small" models in terms of descriptive power. For example, for the modeling of a complex event, either the complex event can be split into simple events and

then connected, or the complex event can be directly modeled as a whole, as long as the logical relationship can be satisfied.

3.2.2 Basic structure

The discrete class is used to describe the atomic model (discrete behavior) in complex products. In general, a discrete class consists of a header, a definition section, and a state section. The header is used to import or inherit from external classes, the definition section is used to initialize the values of parameters and variables, as well as related components and input/output ports, and the state section is used to define the state of the atomic model and the transfer logic between states.

The following sample code illustrates the basic structure and composition of the discrete class.

The header mainly carries out the import and extend operations of the external classes. import means that the external classes need to be instantiated below, extends can be used for the inheritance of the model, and in the discrete class, the object of inheritance can only be the discrete class.

The Definition section describes the attributes of the discrete class, including the parameter, value, and port attributes. Where parameter and value are similar to the functionality in the continuous class, port no longer contains the instantiation of the connector connection class and requires the qualifier event before input/output to indicate the input/output of an event. The value of the input port is controlled by the external model and the value stored in the input port remains the same between accepting inputs and can be treated as a constant, while the output variable is similar to a variable. The variables declared in the VALUE module, on the other hand, can be defined with initial values, and the variables assigned initial values during the initialization phase of the simulation will be initialized to their initial values.

The state part is unique to the discrete atom class, and the keyword state is used to lead this part to indicate the transfer situation and transfer conditions between different states in the discrete event. state part consists of several state machines, and each state machine is started by the state and the name of this state machine, and ended by end. The initial state is composed with the keyword inital state plus the initial state machine name. Relatively speaking, states are categorized into two kinds of states: composite states and simple states. Composite state means that the state can include other simple states and composite states. For

example, for the following example, there are two states in the discrete model daodanshiti, namely work and send, where work is the initial state, i.e., led by the keyword inital state.

As for some state transfer behaviors in the state, some fixed functions can be used to represent them, the main functions are as follows:

行为名称	用途
Entry statement	Define the behavior that needs to be performed before entering the state (typically including a declaration of the duration of the state)
Receives tatement	Define the behavior and state transitions of the state when it receives specific inputs.
State-event statement	Define the behavior and state transitions that occur when a state variable satisfies certain conditions.
Time-event statement	Define behavior and transitions at the end of a state's duration.
Catch-equation statement	Define the continuous behavior of the state during its duration, using equations

A definition template for the discrete class is given below:

```
discrete DiscreteName
    import out_discrete1;
            extends out_discrete1;
parameter:
        datatype argname1;
    out_ discrete1 argname2; //Instantiate the imported out_record1 named argname2
value:
    datatype varname1;
    out_ discrete1 varname2; //Instantiate the imported out_record1 named varname2
port：
    event input datatype argname1;
    event output datatype argname2;
state：
    initial state StateName1
        when entry() then
            statehold(infinite);
        end;
        when receive(ReceiveEvent1) then
```

```
                ......
                transition(StateName2);
            end;
        end;
        state StateName2
            when entry() then
                statehold(0);
            end;
            when timeover() then
                ......
                transition(StateName1);
            out:
                send(port1,value);
            end;
        end;
    end;
```

3.2.3 composite state

Other composite states or simple states can be defined inside the composite state. The behavioral definition of a composite state has the following points to note compared to a simple state:

Only one state can be active in a composite state at the same time, not multiple states at the same time.

1. The behavior defined by a composite state is also valid for the states defined within it, e.g., if a receive statement is used in a composite state, then all states defined within the composite state can trigger the statement. For example, the following composite state StateName1 includes the initial state StateName2. Although there is no receive statement defined in StateName2, if ReceiveEvent1 is received in StateName2, the receive behavior defined in StateName1 is also triggered. receive behavior defined in StateName1. Meanwhile, if a behavior defined in a composite state is redefined in its internal state, the behavior defined in the internal state prevails. For example, if the behavior receive (ReceiveEvent1) is redefined in the StateName3 state, then when the event is triggered, the definition in StateName3 prevails.

```
state StateName1
        when entry() then
            statehold(infinite);
        end;
        when receive(ReceiveEvent1) then
                ......
            transition(StateName4);
        end;
        initial state StateName2
            when entry() then
                statehold(0);
            end;
            when timeover() then
                ......
                transition(StateName1);
            out
                send(port1,value);
            end;
        end;
       state StateName3
            when entry() then
                statehold(0);
            end;
            when receive(ReceiveEvent1) then
                ......
                transition(StateName5);
            end;
        end;

    end;
```

3.2.4 Usage

3.2.4.1 Entry statement

This function indicates the behavior to be performed before entering the state. Typical behaviors include the duration of the state, which is defined using the statehold statement. Note that states that do not have a duration declared in the entry statement, or where the entry statement is not defined, have a default duration of infinite. The entry statement can iterate

over declarative statements such as if. As in the case above the text program for the discrete model:

```
when entry() then
    statehold(infinite);
end;
```

That is to say, the duration of this state is infinite, that is, there is no time to control the state transfer and continue to run the situation, the state transfer and the subsequent operation of the function is only triggered by the input of the relevant events. (This part is usually realized with the timeover statement)

3.2.4.2 Receive statement

The receive statement is an external event, and its core statement, receive, defines which ports receive the message when the event behavior will be triggered.The receive statement can include one or more parameters, each of which must correspond to an input interface declared in the discrete class.

The external event will directly cause a transfer of state, which is indicated by the transition statement. The argument to the transition statement must be one of the multiple states defined in the State module, and the state can be self-referential, allowing it to transfer from one state to another. This statement ends with end.

As in the case above the text program for the discrete model:

```
when receive(ReceiveEvent1) then
        ......
    transition(StateName2);
end;
```

This program means that after receiving the ReceiveEvent1 event, the transfer process from the current state to the StateName2 state is completed by the transition statement after the operation.

3.2.4.3 state event and time event statement

These two languages define two other cases of state transfer, namely continuous behavior caused by changes in equations or state variables and the effect of time constraints on state.

The state-event statement is generally used in conjunction with the catch-equation statement because in ordinary states, the state variables will remain unchanged from the time

of entry into the state, so defining a state event lacks practical significance. A catch-equation describes continuous equation-based behavior over the duration of the state, i.e., the state variables will continue to change over the duration of the state as the equations are solved, and a state event can be triggered once the conditions declared in the State-event statement are met.

The Time-event statement defines an internal event that is common in DEVS, i.e., a time event is triggered when the state expires at the end of the state duration defined in the entry statement, which is denoted by timeover.

In both statements, the output can be additionally described. When the internal behavior has been described, in the out section of each of the two rules, outputs can be defined, and outputs to the corresponding ports are expressed using the send statement.The send function expresses the outputs to the corresponding ports, and the send function consists of two parameters, the first parameter is the output port of the model, and the second parameter is the value of the output to the port.

```
when timeover() then
    ......
    transition(StateName1);
out:
    send(port1,value);
end;
```

This segment of the program indicates that this state, after the time defined in the ENTRY (0 in this case), performs the state transfer written in the TRANSITION, i.e., from the SEND state to the StateName1 state. The value is also output to the corresponding port port1.

Similarly, a state transfer can be signaled by a change in a consecutive event, as in the following case:

```
when h < 10 then  //x ⊆ V_state
  ...
  transition(state2);
out
  send(port1, value);//port1 ⊆ X
end;
```

This case represents a state transfer when h<10 and outputs the value value to port port1.

3.2.4.4 for-equation

The for equation is used to define array equations, i.e., equations that can be satisfied by each element of an array. The number of for equation loops is defined by the range of the for loop variable, and when calculating the number of equations, the number of for equations will be counted as the number of its loops.

```
for i in 1:10 loop
    ...
end;
```

In this case, the variable i is looped from 1 to 10.

3.2.4.5 if-equation

The if equation consists of an if clause, a number of else if clauses, and either 0 or 1 else clause, where the expression must be able to be transformed into a bool expression. During the solving process, the judgment conditions of if and else if are evaluated sequentially, and if one of the conditions is true then the equation under that condition is selected for solving, otherwise the equation under the else statement block is selected for solving if the else statement block exists, and if it doesn't exist then the judgment is over and no equation is executed.

In addition to the above conditions, in order to ensure that the number of equations is fixed under each condition, that is, in order to ensure that the system of equations must be solvable, the number of equations included in each branch of the if-equation must be the same.

```
if expression then
    ...
else if expression2 then
    ...
end;
```

3.2.5 Modeling cases

The following example shows the X language modeling form of the water tank model:

```
discrete LiquidSource
    parameter:
        real flowLevel = 0.02;
```

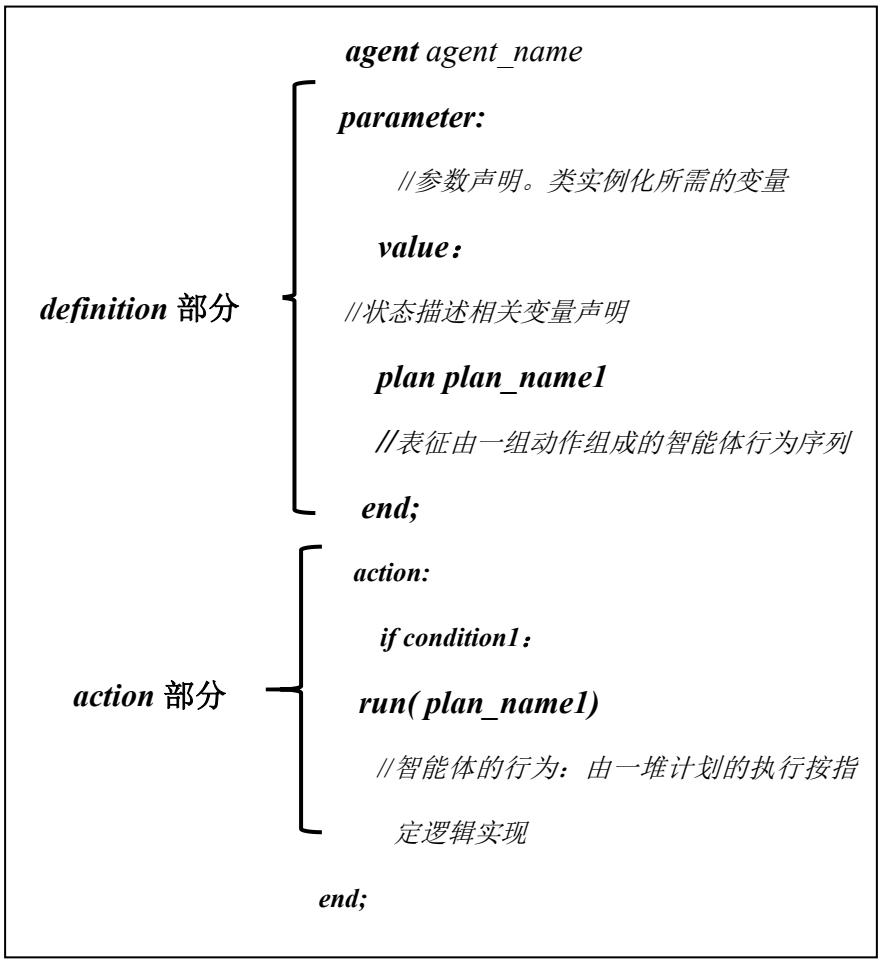
```
port:
    event output real qOut;
state:
    initial state init
        when entry() then
            statehold(0);
        end;
        when timeover() then
            transition(pass);
        out：
            send(qOut,flowLevel);
        end;
    end;
    state pass
        when entry() then
            statehold(150);
        end;
        when timeover() then
            transition(idle);
        out：
            send(qOut,3*flowLevel);
        end;
    end;
    state idle
        when entry() then
            statehold(infinity);
        end;
    end;
end;
```

3.3 Intelligent Body Class

3.3.1 Introduction

The agent class is used to describe the model of an intelligent body in a complex product. Generally, the agent class is composed of two parts: the definition part, which is used to initialize the values of parameters and variables, and the declaration of functions and plans, and the action part, which is used to control the execution of the plans and to set the start and termination conditions for the simulation of an intelligent body.

The agent class describes the architecture as follows:



3.3.2 Basic structure

The creation of the intelligent body model consists of the agent class, where the upper half of the agent is the part of the definition of the intelligent body, i.e., the part of the definition of the variables, functions, and plans, and the action is the part of the invocation of the plans and the control of the life cycle of the intelligent body. The entire agent class is equivalent to being used in an inherited manner.

Example:

```
agent agent_example:
value:
    ……
action:
    ……
end;
```

3.3.3 Usage

The overall intelligences are instantiated and run in the same way as devs' atomic model join coupled model simulator, with variables of type AGENT base class.

Example:

```
import BaseAgent;
agent base1 = BaseAgent.baseagent("base1", [50, 0, 10], "imissile1");
```

Represents the definition of an Intelligence with a complete life cycle process, followed by an Intelligence name that is globally unique.

Used to define variables, functions, plans

The instantiation parameters of the Intelligence are passed in using parameter.

```
agent agent_1
parameter real b; //  实例化参数
end
```

The part of an action connection or invocation of an intelligent body, the

run denotes the order of execution of the connectionplan, understood as a chain of plan execution.

Example:

```
agent agentname:
......
action:
run(interaction_with_env,interaction_with_center,interaction_with_imissile)
end;
```

一个专门用于定义消息格式与内容的数据结构体，结构体中有四个基本的内容，来自谁 from，发送给谁 to，消息内容 content，协议 protocol。

例如：

A special data structure used to define the format and content of the message, the structure has

four basic contents, from whom from, to whom to send to, the message content, protocol protocol.

For example:

A special data structure used to define the format and content of the message, the structure has four basic contents, from whom from, to whom to send to, the message content, protocol protocol.

For example:

```
plan pl_1
    message msg = [from=”a”, to=”b”,content="hello", protocol="inform"];
end;
```

The action of sending a message, which is a function that links a message storage variable in the smartbody class, once declared represents a message being sent to the specified smartbody object.

```
plan pl_1
    message msg = [from=”a”, to=”b”,content="hello", protocol="inform"];
    send(msg);
end;
```

When the intelligent body does not receive receive, the whole will be in the hanging state, not execute other statements until the arrival of the message.

Accepts the content of the message from the intelligent body agt.mailbox, and performs the relevant operations according to the specific protocol content of the message. The important role is to interpret the message to the intelligence! It is also necessary to send feedback. receive() parses the content of the message, understands the semantics through the protocol, and interprets the message in message format.

```
plan pl_1
    message msg = receive();
end;
```

The receive action will only proceed to completion when a message is received. If the agent never receives a message, the receive action will remain in the pending state until a message enters the agt.mailbox.

plan in the upper text is similar to a function, input and output rely on global variables to control, send and receive methods can only be called by the plan module.

For example:

```
plan interaction_with_center:
value:
message msg2 = [];
message msg3 = []
   map<string, list<float>> content = {};
action:
   msg2 = …
   send(msg2);
   msg3_content = receive();
end;
```

3.4 Coupled class

3.4.1 Introduction

The coupling class is an important class of X language, and it is also the key for X language to realize the connection between simulation-level modeling and system modeling. The coupled model in X language undertakes the task of connecting multi-domain model simulation, in the coupled model, not only can connect the same domain (such as discrete model), but also can connect the models of different domains, only need the ports between the two can match each other. It is the main class of tools for modeling and simulation of complex systems in X language.

3.4.2 Basic structure

The coupling class is organized by the keyword couple. A coupled class consists of three parts: a header part, an attribute part, and a linking part.

The head part includes two parts: importing external model (structural keyword import) and inheriting external model (structural keyword extends); the class in the extends part can only be a coupled class, and the import part imports the external class as needed for inheritance or instantiation. The attributes section describes the basic attributes of the model,

and the connections section describes the connections between different submodels, indicating the directed connections between two ports of the submodels.

3.4.3 Usage

The properties section includes three parts: parameter, port, and part. Parameter represents the intrinsic property of the couple class, which is a constant. Port represents the interface of the couple class. Part represents the submodule contained in the couple, which contains continuous, discrete, and the instantiation of the couple class. Instantiation.

The connection part is led by the connection keyword and internally connects the ports in the form of connect. the attributes of the quantities connected at both ends of the connection must be the same, they can be the same as a single variable, but they need to be of the same numeric value type, and they can be the same as a connector class. The connection relationship has a direction, and the two ports connected by connect represent a directed connection from the first port to the second port.

For example, in the following example, the four models connector, continuous, discrete, and couple are imported, and couple_1 is inherited. part, instantiate the continuous_1 and discrete classes. port part, define your own port using the imported connector_1. The connection part, connects the p-port of name_2 to the n-port of name_3 (from the p-port of name_2 to the n-port of name_3), and connects the n-port of name_2 to the port of name_1 of this couple model, and, since name_1 is a CONNECTOR class, it can be assumed that the p-port of name_2 is also a CONNECTOR class. p-port is also of the CONNECTOR class.

The sample code is as follows:

```
couple Model_1
import connector_1;
    import continuous_1;
    import discrete_1;
    import couple_1;
extends couple_1;
```

```
parameter:
real a=1;
port:
connector_1 name_1;
part:
    continuous_1 name_2(am=a);
    discrete_1 name_3;
connection:
    connect(name_2.p,name_3.n);
    connect(name_2.n,name_1);
end;
```

3.4.4 Modeling case (water tank model)

```
couple topmodel
import Tank;
import LiquidSourse;
import PIcontinuousController;
part:
    Tank tp;
    LiquidSource ls;
    PIcontinuousController pc;
connection:
    connect(ls.qOut,tp.qIn);
    connect(tp.tSensor,pc.qin);
end;
```

3.5 Connector class

3.5.1 Introduction

The connector class is a key class for implementing component-based modeling. connector is a method for allowing models to exchange information with models. connector focuses on non-causal modeling of physical systems. The non-causal physical modeling approach distinguishes between two different types of variables: flow variables and potential variables. A connector is generally used to denote a binding semantics between two component ports that have or satisfy Kirchhoff's theorem, i.e., the potential variables are equal and the sum of the flow variables is equal to zero.

3.5.2 Basic structure

The connector class is organized by the keyword connector. It contains only one part, value, which can contain several quantities of different data types. Generally in the modeling process, the continuous class calls the connector class for non-causal modeling. The data type used is real. the following is a template for defining the connector class:

```
connector connectorname
value:
    real varname1;
    flow real varname2;
end;
```

3.5.3 Usage

The connector class can only be called by the continuous and discrete classes. Instantiation is defined in the port section of the calling body.

3.5.4 Example Code

The following example defines the positron in the circuit.

```
connector PositivePin
value:
    real v;
    flow real i;
end;
```

3.6 Record class

3.6.1 Introduction

Record classes are a class of aggregated data types in the X language. Record classes can have their own variables for implementing more complex data structures.

3.6.2 Basic structure

The record class is organized by the keyword record. It contains only one part, value,

which can contain multiple quantities of different data types and is used to express complex data structures. Specific description of the form of the template is as follows:

```
record RecordName
   value:
      //declarations for record variables
end;
```

3.6.3 Usage (constructor of record)

Now, we know how to define a record type. How should we create a record type? If we want to declare a variable that happens to be of the record type, the variable declaration itself creates an instance of the record type, and we can also specify the value of the variable within the record type through a modification statement, for example:

```
parameter Vector v(x=1.0, y=2.0, z=0.0);
```

In some cases, however, we may wish to create a record type instead of a variable (e.g., to use it in an expression, to pass as a parameter to a function or to use it in a modifier function). For each record type definition, a function name is automatically generated that is identical to the name of the record type. This function is called the "record constructor". The record constructor takes a variable that matches the internal definition of the record type and returns an instance of the record type. So, in the above example of Vector definition, we can also initialize the parameter variable by record constructor as follows:

```
parameter Vector v = Vector(x=1.0, y=2.0, z=0.0);
```

In this case, the value of variable v is assigned by calling the record constructor through the expression Vector(x=1.0, y=2.0, z=0.0).

3.6.4 sample code (computing)

```
record Vector
value:
    real x;
    real y;
    real z;
```

end;

3.7 Function Class

3.7.1 Introduction

Function classes make it easy to describe the mathematical properties of objects, and sometimes it is necessary to create new functions for special purposes, using function classes.

3.7.2 Basic structure

The function class is led by the keyword function. It contains a header, a definition section, and a behavior section. The header part describes the imported external functions. Definition part includes two parts: port and value, port defines the input and output of the function: the input parameter and return value of the function are defined in port, the input parameter is declared with the qualifier input, and the return value is declared with output. The behavior part is led by the keyword action, and the action part can be used to describe the relationship between input and output.

Considering all the above, the template for function class definition is given below:

```
function functionname
import out_functionname1;
import out_functionname2;
…//导入的外部函数
port:
input inputtype argname1;
input inputtype argname2;
output outputtype argname1;
…
value：
vartype varname1；
vartypr varname2；
…//中间变量
action：
//描述输入输出关系
end；
```

3.7.3 Usage

The function class can be called by the continuous, discrete, and function classes. When called, they need to be imported in the header.

3.7.4 Sample Code

A segmented function is modeled below.

```
function fal :
port:
input real x;
input real a1;
input real delta1;
output real y;
action:
if abs(x) > delta1 then :
    y = (abs(x)) ^ a1 * sgn(x);
else :
    y = x /( delta1 ^ (1 - a1));
end;
```

3.8 Module class

3.8.1 Introduction

Module classes are a descriptive architecture used in the X language for describing simulation models like causal module graphs with specific causal relationships. Module classes differ from function classes in that module classes are capable of describing simulation time-dependent operations such as integration and differentiation.

3.8.2 Basic structure

The module class is organized by the keyword block. It contains a header section, a definition section, and a behavior section. The header section describes the imported external functions. Definition part includes two parts, port and value, port defines the input and output of the function: the input parameters and outputs of the module are in port, input parameters

are declared with the qualifier input, and outputs are declared with output. The behavior part is led by the keyword action, and the action part can be used to describe the relationship between inputs and outputs, as well as to define operations related to the simulation time.

Considering all the above, the template for function class definition is given below:

block blockname

import out_functionname1;

import out_functionname2;

…//导入的外部函数

port:

input inputtype argname1;

input inputtype argname2;

output outputtype argname1;

…

value：

vartype varname1；

vartypr varname2；

…//中间变量

action：

//描述输入输出关系

end；

3.8.3 Usage

Module classes can participate as modules in the construction of COUPLE classes and enable modeling and simulation of multi-domain models such as control models through causality-determined connections.

3.8.4 Sample Code

A segmented function is modeled below.

block differential:

port:

input real x;

output real y;

action:

y = der(x);

end;

3.9 X-language import of external files

3.9.1 Introduction

When importing external files, you need to use the external import function. The external function currently defines three types of methods for importing external models: DLL connection library models, FMU models and Julia models.

3.9.2 basic grammar

The keyword for externally importing a function is external (), and this statement can only be used in a function class, as shown in the example. The header of the function class is the same as a normal function. The port in the definition section needs to correspond to the input and output parameters of the externally imported function. If the corresponding parameters of the externally imported function need the assistance of intermediate variables, define the intermediate variables in the value section. In the action part of the external function can be called to import external files, the first parameter of the external function to identify the type of file imported, such as Julia, the subsequent parameters based on different types of different definitions, such as Julia file import only need to add the storage location of Julia file can be. It should be noted that a function can only import a unique external file. And external import only means that the file is imported, it is not the same as the function is called, only when the imported file of the X language function is called the function will be called to

perform the behavior.

Considering all of the above, a template for the external function import format is given below:

```
function externanlFunctionName
    import out_functionname1;
    import out_functionname2;
    …//导入的外部函数
port:
    input inputtype argname1;
    input inputtype argname2;
    output outputtype argname1;
    …//需要和外部函数的输入输出参数进行对应
value：
    vartype varname1；
    vartypr varname2；
    …//中间变量
action：
    //描述文件的调用格式
    external ("Julia","//文件地址");//导入 Julia 文件
    //external ("DLL","dll 声明式","//DLL 编译所需 lib 地址","//DLL 文件存储位置" );
//导入 DLL 文件
    //external ("FMU","//文件地址","FMU 文件名称");//导入 FMU 文件
end；
```

4 Summary of the detailed specification of the X language

4.1 The X-Language Bacchus Paradigm

```
//整体存储框架
stored_definition   ::= ['within' name ';'] {['final'] class_definition}
//类定义
class_definition   ::= ('encapsuate')?   class_prefixes class_specifier
//类前缀
class_prefixes   ::=   ['partial'] ('class'|'continuous'|'discrete'|'agent'|'couple'|'connector'|
'enum'|'record'|'function')
//类结构定义
class_specifier   ::=   (IDENT   definition_section   {composition}   'end')|
enumeration_definition
//类组成部分
composition ::= connection_section | stm_section | equation_section   | act_section
//定义模块
definition_section ::={(import_clause
    | extends_clause
    | class_definition
    | parameter_component_clause)';'}
    {port_section
    |part_section
    |value_section
|plan_definition}
//连接部分
connection_section ::= 'connection:'{connect_clause';'}
//状态机部分
```

```
stm_section ::= 'state:' {state_definition}
//方程部分
equation_section ::= 'equation:' {equation ';'}
//活动部分
act_section ::= 'action:' {statement';'}
//组件部分
part_section ::= 'part:'{component_clause';'}
//值变量部分
value_section ::= 'value:'{component_clause';'}
//端口部分
port_section ::= 'port:'{port_component_clause}
//端口组件声明
port_component_clause
    : (port_prefix)?  type_specifier  component_list';'
//端口前缀
port_prefix  ::= 'input' |'output'|'event''input' |'event''output'
//计划定义
plan_definition ::= 'plan' IDENT definition_section act_section 'end';'
//状态定义
state_definition ::= 'initial' 'state' IDENT  (state_statement| state_definition)*  'end';'
|'state' IDENT  (state_statement| state_definition)*  'end';'
|'state'  IDENT  catch_clause  ((when_receive_clause';')|(when_statement';'))*
'end';'
//状态行为语句
state_statement ::= (when_entry_clause
    | when_receive_clause
| when_goto_out_clause)';'
//状态持续时间定义规则
```

```
statehold_clause ::= 'statehold' '('expression')'
 //离开状态执行行为
when_goto_out_clause ::= 'when'timeover_clouse'then'
        {statement ';'}
        [out_with_satement]
        'end'
 //输出行为
out_with_satement ::= 'out' (statement ';')*
 //接受消息执行行为
when_receive_clause ::= 'when' receive_clause 'then'{statement ';'}
      [out_with_satement]
      'end'
 //进入状态执行行为
when_entry_clause ::= 'when''entry' 'then' statehold_clause ';' {statement';'} 'end'
//状态内方程规则描述
catch_clause ::=
        'catch'   {simple_statement';'}
        'equation' {equation';'} 'end'';'
//枚举类定义
enumeration_definition ::= IDENT'{'enum_list'}'
//枚举列表
enum_list ::= IDENT { ',' IDENT }
//变量实例化规则定义
component_clause ::=['replaceable'] type_prefix type_specifier component_list
//元素声明列表
component_list ::= component_declaration {','component_declaration}
//元素声明
component_declaration ::= IDENT [array_subscripts] [modification]
```

```
//实例化格式
modification ::= class_modification
    | '=' initializer
//初始化格式
initializer ::=   array_literal
    | expression
//类实例化列表
class_modification   ::= '(' [argument_list] ')'
//参数列表
argument_list ::= argument {',' argument}
 //名称参数列表
argument ::= (name   [modification])|expression
//import 规则
import_clause ::= 'import' (IDENT '=' name | name ('.' ( '*' | '{' import_list '}' ) )? )
//import 列表
import_list ::= IDENT {',' IDENT}
//继承规则
extends_clause ::= 'extends' type_specifier [class_modification]
//参数声明规则
parameter_component_clause ::= 'parameter' type_specifier component_list
//类型前缀
type_prefix ::= ['flow'] [is_discrete | is_constant]
//是否是离散变量
is_discrete ::= 'discrete'
//是否是常量
is_constant ::= 'constant'
//接受规则
receive_clause ::= 'receive('component_reference')'
```

```
        | 'receive('expression{','expression}')'
    //发送规则
    send_clause ::= 'send"('[component_reference ','expression]')'
        |'send"('component_reference','component_reference')'
    //状态持续时间结束规则
    timeover_clouse ::= 'timeover"(")'
    //连接规则
    connect_clause ::= 'connect' '(' component_reference ',' component_reference ')'
    //方程定义
    equation ::= simple_equation
        | if_equation
        | for_equation
        | when_receive_equation
        | when_equation
    //等式方程
    simple_equation ::= expression '=' expression
    //if 方程
    if_equation ::= 'if' expression 'then'    {equation ';' }   {elseif_equation}   [else_equation]
'end'
    //elseif 方程
    elseif_equation ::= 'elseif' expression 'then'    {equation ';' }
    //else 方程
    else_equation ::= 'else' {equation ';' }
    //for 方程
    for_equation ::= 'for' for_index 'loop'   {equation ';' }    'end'
    //for 循环参数
    for_index ::= IDENT ['in' expression ]
    //when 方程
```

when_equation ::= 'when' expression 'then' {equation ';' } [out_with_equation] {elsewhen_equation} 'end'

//输出方程

out_with_equation ::= 'out' {equation ';'}

//when 方程

elsewhen_equation ::= 'elsewhen' expression 'then' {equation ';'}

[out_with_equation]

//when receive 方程

when_receive_equation ::= 'when' receive_clause 'then'{equation ';'}

[out_with_equation]

'end'

//过程描述语句定义

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

//赋值语句

simple_statement ::= component_reference '=' expression

| '('output_expression_list')' '=' function_call

//brake 语句

break_statement ::= 'break'

//continue 语句

continue_statement ::= 'continue'

//运行语句

run_statement ::= 'run''('component_reference {','component_reference}')'

//智能体仿真结束语句

agentover_statement ::= 'agentover'

//返回语句

return_statement ::= 'return'

//if 语句

if_statement ::= 'if' expression 'then' {statement ';'} {elseif_statement} [else_statement] 'end'

//elseif 语句

elseif_statement ::= 'elseif' expression 'then' {statement ';'}

//else 语句

else_statement ::= 'else'{statement ';'}

//for 语句

for_statement ::= 'for' for_index 'loop' {statement ';'} 'end'

//while 语句

while_statement ::= 'while' expression 'loop' {statement ';'} 'end'

//when 语句

when_statement ::= 'when' expression 'then' {statement ';'} {elsewhen_statement} 'end'

//elsewhen 语句

elsewhen_statement ::= 'elsewhen' expression 'then' {statement ';'}

//transition 语句

```
transition_clause ::= 'transition' '('IDENT')'
//表达式树定义
expression ::= logical_expression
    |colon_literal
    |receive_clause
colon_literal ::=   logical_expression ':' logical_expression
    | logical_expression ':' logical_expression ':' logical_expression
logical_expression ::= logical_term ['or' logical_expression]
logical_term ::= logical_factor('and' logical_term)?
logical_factor ::= ['not'] relation
relation ::= arithmetic_expression [relational_operator arithmetic_expression]
relational_operator ::=   '<'| '<=' | '>' | '>=' | '==' |'!='
arithmetic_expression ::= add_operator arithmetic_expression
    | term [add_operator arithmetic_expression]
add_operator ::= '+' |'-'| '.+' | '.-'
term ::= factor [mul_operator term]
mul_operator ::= '*' | '/' | '.*' | './'
factor::= primary [index_operator primary]
index_operator ::='^' | '.^'
//表达式基础元素定义
primary ::= bracket_exp
    | interger_literal
    | real_literal
    | bool_literal
    | string_literal
    | list_literal
    | map_literal
    | component_reference
```

```
    | component_inistant
    | function_call
    | '(' output_expression_list ')'
    | array_literal
    | infinite_clause
//括号表达式
bracket_exp ::='('expression')'
//变量名称
name   ::=   IDENT {'.' IDENT}
//无穷大表达式
infinite_clause ::= 'infinite'
//类型
type_specifier ::= ['.'] name
    |list_type
    |map_type
//列表类型
list_type ::= 'list<'type_specifier'>'
//map 类型
map_type ::= 'map<'type_specifier','type_specifier'>'
//数组表达式
array_literal ::='[' array_arguments ']'
//字符串表达式
string_literal ::= STRING
//实数表达式
real_literal ::= UNSIGNED_NUMBER | NUMBER
//整形表达式
interger_literal ::= UNSIGNED_INTEGER
//bool 表达式
```

```
bool_literal ::='true'|'false'
//列表表达式
list_literal ::= empty_args|('{'list_args'}')
//列表元素表达式
list_args ::= expression(','expression)*
//map 表达式
map_literal ::= empty_args|('{'map_args{','map_args}'}')
//map 元素表达式
map_args ::= '{'expression':'expression'}'
//列表和 map 空表达式
empty_args ::='{''}'
//函数调用表达式
function_call ::= (component_reference | 'der'|'pre' ) function_call_args
//变量引用
component_reference ::= ['.']   component_reference_args
//变量引用元素
component_reference_args ::= IDENT [array_subscripts]['.'component_reference_args]
//函数调用元素
function_call_args   ::= '(' [function_arguments] ')'
//函数调用元素
function_arguments ::=   expression {',' expression}
    | named_arguments
//数组元素
array_arguments ::= expression {',' expression}
//名称元素列表
named_arguments ::=   named_argument {',' named_argument}
//名称元素
named_argument ::= IDENT '=' expression
```

```
//函数返回值列表
output_expression_list ::= [expression] {',' (expression)? }
//数组下标
array_subscripts ::= '[' subscript {',' subscript} ']'
//下标
subscript ::= select_all | expression
select_all ::= ':'

//词法元素定义
//标识符定义
IDENT ::=  NONDIGIT { DIGIT|NONDIGIT}
//字母元素定义
NONDIGIT ::= 'a'..'z'|'A'..'Z'|'_'
//无符号整型
UNSIGNED_INTEGER ::= DIGIT {DIGIT}
//无符号实数
UNSIGNED_NUMBER ::= UNSIGNED_INTEGER EXP  | UNSIGNED_INTEGER '.' DIGIT {DIGIT} EXP?
//无符号数定义
NUMBER ::= UNSIGNED_INTEGER EXP
    |  UNSIGNED_INTEGER '.' DIGIT{DIGIT} EXP?
//数字字符定义
DIGIT ::= '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
//e 为底的指数定义
EXP ::= ('E' | 'e') ['+' | '-'] UNSIGNED_INTEGER ;
//字符串定义
STRING ::= "" {EscapeSequence | ~('\\'|"") } ""
EscapeSequence ::= '\\' ('b'|'t'|'n'|'f'|'r'|'\"'|'\''|'\\')
```

```
//注释定义
COMMENT ::=   '/*' .*? '*/'
COMMENT1 ::=   '//' .*?   '\n'
```

5 X Language Graphics Description Specification

5.1 Requirements diagram

5.1.1 Purpose

Text-based requirements have traditionally been an important product of systems engineering. This does not mean that all approaches require text-based requirements. An increasingly widely used technique is to create use cases to replace text-based functional requirements and constraint expressions to replace text-based non-functional requirements. However, when we need to show these requirements and how they relate to other model elements, requirement diagrams can be created.

This diagram is especially valuable when the viewer needs to see traceability from the requirement to the elements of the system model that depend on it.

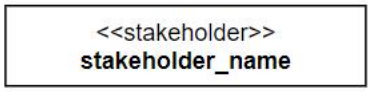
5.1.2 When to Create a Requirements Diagram

When adding new elements to the model, you create a relationship that points from those elements to the requirements that drove their creation. Establishing the traceability of requirements in this way is an activity that occurs throughout design and development. You may need to create requirements diagrams that show those relationships at any point in this work.

5.1.3 stakeholder

A stakeholder represents the demand side of the entire project or the party responsible for the demand, and is the person who proposes or practices the demand for the entire project development. The stakeholder identifier is a rectangle with the meta type <<stakeholder>> before the name.

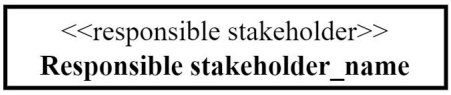
The exact form is as follows:



5.1.4 Responsible stakeholders

The Responsible Stakeholder represents the party responsible for the requirements of the entire project and is the practitioner of the entire project development. The identifier of the responsible stakeholder is a rectangle with the meta type <<responsible_stakeholder>> before the name.

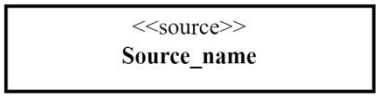
The specific form is as follows:



5.1.5 Sources of demand

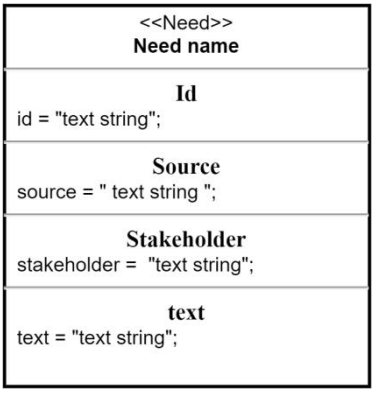
The requirements source represents the origin of the entire project requirements, which may be some official unit or requirements specification document, etc. The identifier of a requirement source is a rectangle with the meta type <<source>> before the name.

The specific form is as follows:



5.1.6 stakeholder needs

The identifier of a stakeholder requirement is a rectangle with the meta type <<need>> before the name. A stakeholder requirement has four attributes: id, source, stakeholder, and text, all of which are of type string. id indicates the number of the requirement, source indicates the source of the requirement, stakeholder indicates the stakeholder who made the requirement, and text describes the specifics of the requirement, generally through natural language (Stakeholder requirements often have no specific constraints).



5.1.6.1 Text Example

need Need_name:

Id : “text string”; 编号

Source: “text string”; 需求的来源

Stakeholder: “text string”; 提出需求的利益相关者

Text: “text string”; 需求的具体描述（无固定规则，收集于所有的利益相关者）

end;

5.1.7 requirement

The identification method of a requirement is a rectangle with the meta type <<requirement>> before the name. A requirement has seven attributes: id, responsible stakeholder, priority, type, level, role, text, all of which are of type string. id represents the number of the requirement, responsible stakeholder represents the party responsible for practicing the requirement, type represents the type of the requirement (generally including functional and non-functional requirements, non-functional requirements can be subdivided into performance requirements, design requirements, environmental requirements, applicability requirements), level represents the level of the requirement (generally including system-level requirements and component-level requirements), role represents the role of the MDAO-oriented requirement (generally including design variables, design variable boundaries, input parameters, constraints, target objects), text represents the role of the

requirement (generally including design variables, design variable boundaries, input parameters, constraints, target objects), and text represents the role of the MDAO-oriented requirement. The role represents the roles of MDAO-oriented requirements (generally including design variables, design variable boundaries, input parameters, constraints, and target objects). text generally describes the specific content of the requirements through natural language (for different types of requirements, the description of the requirements has its own constraint format).

The specific forms are as follows:

<<requirement>> **Requirement Name**
Id **id** = "text string";
Responsible stakeholder **responsible stakeholder**= " text string ";
Priority **priority** = High|Medium|Low;
Type **type** = Functional|Nonfunctional|Performance|Design(constraint)|Environmental|Suitability;
Level **level** = System requirement|Component requirement;
MDAO_Role **role** = Design variable|Design variable bound|Input parameter|Contraint|Objevtive;
Text **Functional:** "The SYSTEM shall [exhibit] FUNCTION [while in CONDITION]" **Performance:** "The SYSTEM shall FUNCTION with PERFORMANCE [and TIMING upon EVENT TRIGGER] while in CONDITION " **Design:** "The SYSTEM shall [exhibit] DESIGN CONSTRAINTS [in accordance with PERFORMANCE while in CONDITION]" **Environmental:** " The SYSTEM shall [exhibit] CHARACTERISTIC during/after exposure to ENVIRONMENT [for EXPOSURE DURATION] **Suitability:** "The SYSTEM shall exhibit CHARACTERISTIC with PERFORMANCE while CONDITION [for CONDITION DURATION]"

- **Functional requirements**: define what functions need to be performed to accomplish the objectives [22]
  Pattern: The SYSTEM shall [exhibit] FUNCTION [while in CONDITION]
  Example: "The aircraft shall provide propulsive power [during the entire mission]"

- **Performance requirements**: define how well the system needs to perform the functions [22]
  Pattern: The SYSTEM shall FUNCTION with PERFORMANCE [and TIMING upon EVENT TRIGGER] while in CONDITION
  Example: "The aircraft shall fly at min Mach 0.8 during cruise"

- **Design constraint requirements**: limit the options open to a designer of a solution by imposing immovable boundaries and limits [27]
  Pattern: The SYSTEM shall [exhibit] DESIGN CONSTRAINTS [in accordance with PERFORMANCE while in CONDITION]
  Example: "The aircraft shall have technologies with maturity TRL 9"

- **Environmental requirements**: define which characteristics the system should exhibit when exposed in specific environments (e.g. acoustic/thermal loads, atmospheric conditions) [27]
  Pattern: The SYSTEM shall [exhibit] CHARACTERISTIC during/after exposure to ENVIRONMENT [for EXPOSURE DURATION]
  Example: "The aircraft shall be maneuverable during exposure to ice conditions [for the entire flight]"

- **Suitability requirements**: include a number of the "-ilities" in requirements to include, e.g. transportability, survivability, flexibility, portability, reusability, reliability, maintainability, and security [27]
  Pattern: The SYSTEM shall exhibit CHARACTERISTIC with PERFORMANCE while CONDITION [for CONDITION DURATION]
  Example: "The aircraft shall exhibit a steady gradient of climb of minimum 2.4% while condition of one-engine-inoperative"

5.1.7.1 Text Example

requirement Requirement_name:

Id : "text string"; serial number

Responsible stakeholder: "text string"; Stakeholders responsible for realizing or implementing the requirement

Priority: High|Medium|Low; Prioritization of needs

Type:Functional|Nonfunctional|Performance|Design(constraint)|Environmental|Suitability; Specific types of requirements: functional and non-functional (including: performance requirements, design requirements, environmental requirements, suitability requirements)

Level: System requirement|Component requirement; Hierarchy of requirements: system-level requirements and component-level requirements

Role: Design variable|Design variable bound|Input parameter|Contraint|Objevtive; MDAO-oriented requirement roles: design variables, design variable boundaries, input parameters, constraints, objects

Text:

Functional: "The SYSTEM shall [exhibit] FUNCTION [while in CONDITION]"

Performance: "The SYSTEM shall FUNCTION with PERFORMANCE [and TIMING upon EVENT TRIGGER] while in CONDITION "

Design: "The SYSTEM shall [exhibit] DESIGN CONSTRAINTS [in accordance with PERFORMANCE while in CONDITION]"

Environmental: " The SYSTEM shall [exhibit] CHARACTERISTIC during/after exposure to ENVIRONMENT [for EXPOSURE DURATION]

Suitability: "The SYSTEM shall exhibit CHARACTERISTIC with PERFORMANCE while CONDITION [for CONDITION DURATION]"

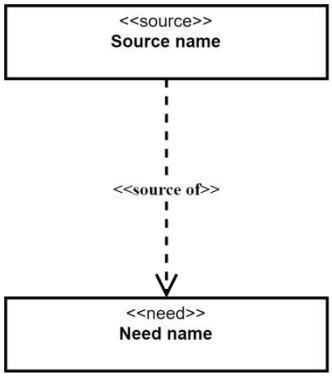
end;

5.1.8 Relationship of Requirement

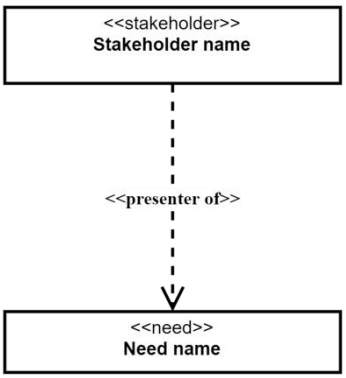
Documenting requirements in a system model is useful. However, direct relationships between requirements and other model elements have greater value. There are ten general types of requirements relationships that may be used in the modeling process: source, propose, responsible for, contain, track, inherit, improve, satisfy, validate, and map.

These relationships establish the traceability of requirements in the system model, which is generally a process requirement in systems engineering organizations. However, from a practical standpoint, documenting these relationships in the model allows you to use modeling tools to automatically generate requirements traceability and perform automated downstream impact analyses when requirements change. These features will save a lot of time, and that translates directly into savings on costs.

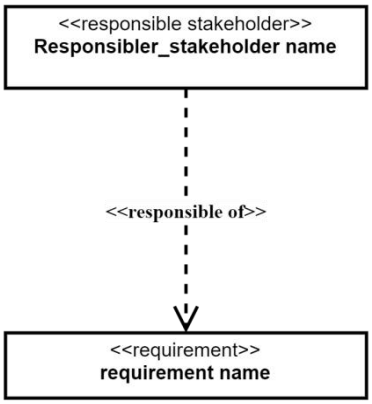
Requirements originate from an official unit or requirements specification document, and the source relationship describes the origin of the requirement. The specific form is as follows:



Requirements are generally formulated by all stakeholders, and the formulation relationship describes the stakeholders directly related to the original requirement. The specific form is as follows:



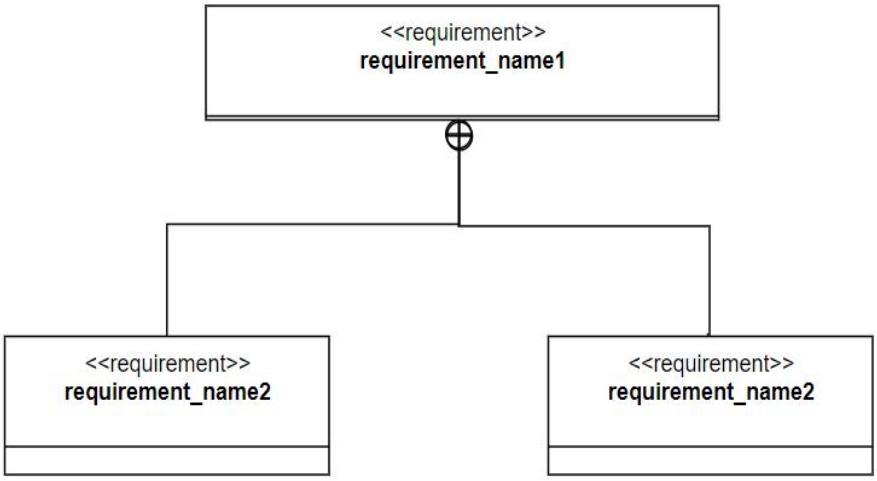
Requirements in the design process to the stakeholders who will be responsible for practicing accordingly. This takes the form of the following:



5.1.8.1 Include Relationship

There is a containment relationship between requirements, i.e., requirements can contain other requirements. Containment relationships between requirements are generally indicated by the crosshair marking method.

The specific form is as follows:

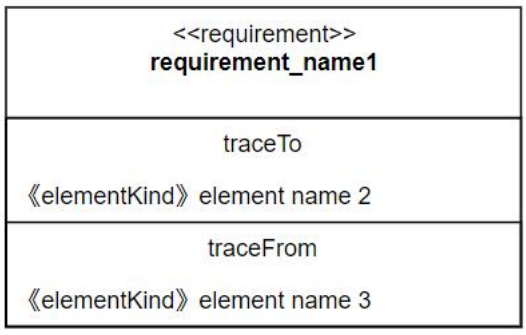


The figure indicates that Demand 1 contains Demand 2 and Demand 3.

5.1.8.2 Tracking relationships

Formally, a tracking relationship is a dependency relationship. A trace relationship is a weak relationship. It simply expresses a basic dependency: a modification to a provider element may result in the need for a modification to a client element. A trace relationship can generally be represented either by a requirement attribute field or by a dotted line with an open arrow (with the <<trace>> metatype above it).

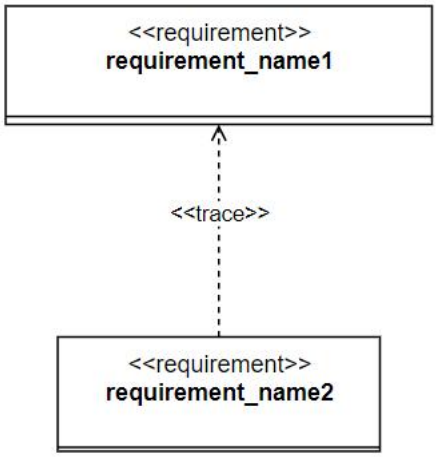
Approach 1: Requirements attribute column representation



The diagram traceTo indicates that requirement 1 will trace back to requirement 2 and

traceFrom indicates that requirement 3 will trace back to requirement 1.

Approach 2: Dashed line with open arrow (with <<trace>> metatype above) representation

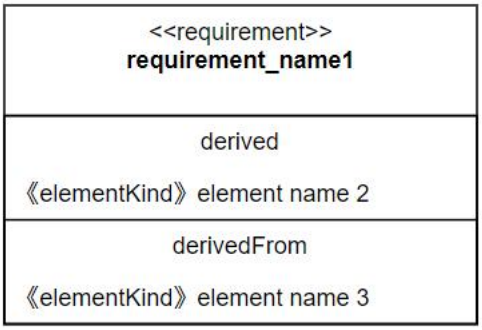


The figure indicates that demand 2 will track back to demand 1

5.1.8.3 Derive of requirement relationships

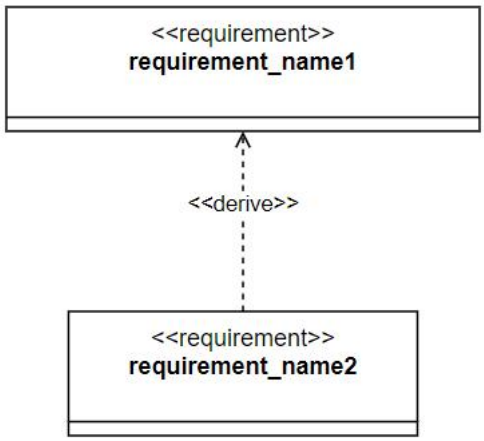
The derived requirements relationship is another type of dependency relationship. This relationship must have requirements on both the client and provider side. A derived requirement relationship means that the client's requirement derives from the provider's requirement. Derived requirement relationships can typically be represented by a requirement attribute field or by a dotted line with an open arrow (with the <<derive>> meta type above it). It is perfectly legal to have multiple levels of derived relationships, and dependencies are passable. Therefore, if the base requirement changes, then the downstream effects are felt throughout the chain of derived requirement relationships.

Approach 1: Requirement Attribute Column Representation



In the diagram derived indicates that requirement 2 inherits from requirement 1, and derivedFrom indicates that requirement 1 inherits from requirement 3.

Approach 2: Dashed line with open arrow (with <<derive>> meta type above) representation

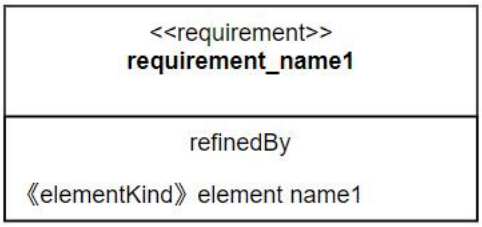


The figure indicates that requirement 2 is inherited from requirement 1

5.1.8.4 Refined relations

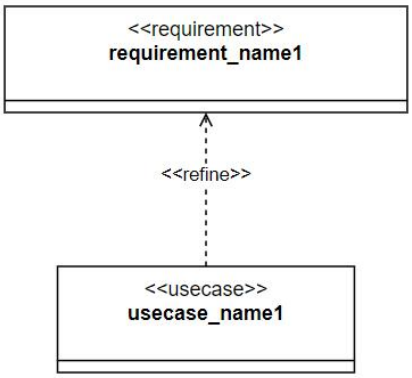
An improvement relation is another kind of dependency. The X language does not have any restrictions on the kinds of elements that can be displayed at either end of the improvement relationship. However, use cases are generally used to improve the functional requirements of the text. An improvement relationship indicates that the client-side elements are more specific than the provider-side elements. Improvement relationships can generally be represented either by a requirement attribute field or by a dotted line with an open arrow (with the <<refine>> metatype above it).

Approach 1: Requirement Attribute Column Representation



The diagram refinedBy indicates that element 1 improves requirement 1

Way 2: Dashed line with open arrow (with <<refine>> meta type above) representation

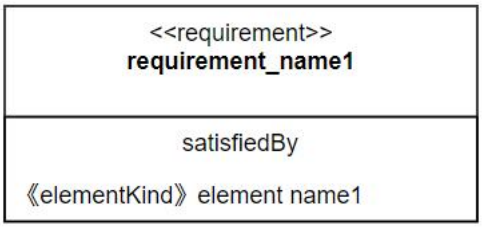


The figure indicates that use case 1 refined requirement 1

5.1.8.5 Satisfaction relationships

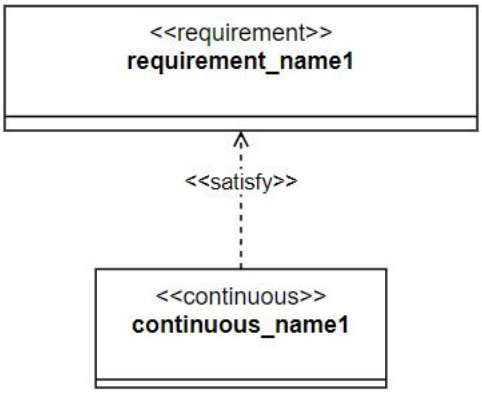
A fulfillment relationship is another type of dependency. This relationship must have a requirement on the provider side.The X language imposes no restrictions on the kinds of elements that can appear on the client side. However, client-side elements are usually classes (continuous classes, discrete classes, etc.). A fulfillment relationship is an assertion that an instance of a client class will satisfy a requirement on the provider side. Satisfaction relationships can typically be represented either by a requirement attribute field or by a dotted line with an open arrow (with the <<satisfy>> metatype above it).

Approach 1: Requirement Attribute Column Representation



The diagram satisfiedBy indicates that element 1 (which can be a continuous class, discrete class, etc.) will satisfy requirement 1.

Way 2: Dashed line with open arrow (with <<satisfy>> meta type above) representation

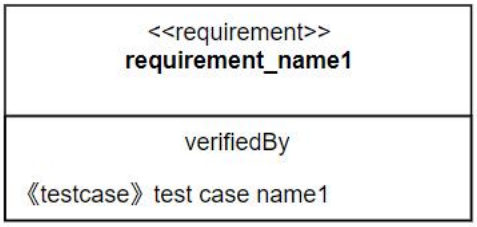


The figure indicates that continuous class 1 will satisfy requirement 1

5.1.8.6 verify relation

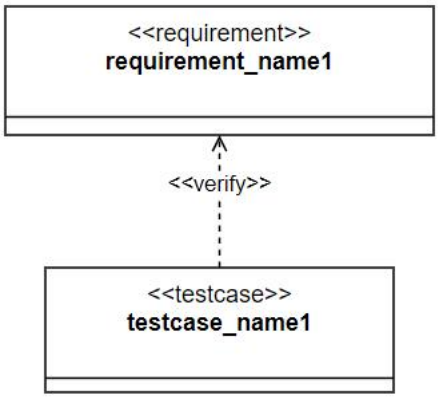
A verify relationship is another type of dependency relationship. Like a fulfillment relationship, a verify relationship must have a requirement on the provider side.The X language imposes no restrictions on the kinds of elements that can appear on the client side. However, client-side elements are usually test cases. A test case will typically be a system behavior that, when simulated, demonstrates whether the system actually satisfies the requirement. Validation relationships can generally be represented either through the Requirements Attributes column or through a dotted line with an open arrow (with the <<verify>> meta type above it).

Approach 1: Requirements attribute column representation



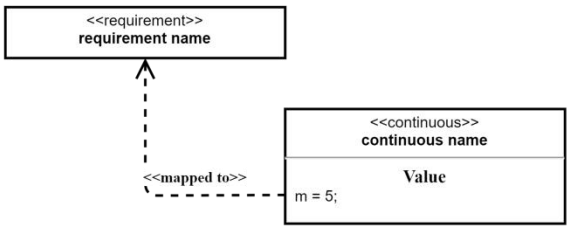
The diagram verifiedBy indicates that test case 1 verifies requirement 1

Approach 2: Dashed line with open arrow (with <<verify>> meta type above) representation



The figure indicates that test case 1 verifies requirement 1

A mapping relationship is another type of dependency, and like validation, a mapping relationship must have a requirement on the provider side. The client side generally corresponds to specific attributes or behaviors within the model element. The mapping relationship is set up to establish an association between the requirement and the specific attribute or behavior of the model element that corresponds to its realization. Mapping relationships can typically be represented by dotted lines with open arrows (with the <<mapped to>> metatype above).



5.1.9 Requirements text

Req req_name

import model elements a;

import model elements b;

...

need Need_name1:

Id : “text string”;

Source: “text string”;

Stakeholder: “text string”;

Text: “text string”;

end;

need Need_name2:

Id : “text string”;

Source: “text string”;

Stakeholder: “text string”;

Text: “text string”;

end;

need Need_nameN:

Id : “text string”;

Source: “text string”;

Stakeholder: “text string”;

Text: “text string”;

end;

requirement Requirement_name1:

Id : “text string”;

Responsible stakeholder: “text string”;

Priority: High|Medium|Low;

Type:Functional|Nonfunctional|Performance|Design(constraint)|Environmental|Suitability;

Level: System requirement|Component requirement;

Role: Design variable|Design variable bound|Input parameter|Contraint|Objevtive;

Text：

Functional: "The SYSTEM shall [exhibit] FUNCTION [while in CONDITION]"

Performance: "The SYSTEM shall FUNCTION with PERFORMANCE [and TIMING upon EVENT TRIGGER] while in CONDITION "

Design: "The SYSTEM shall [exhibit] DESIGN CONSTRAINTS [in accordance with PERFORMANCE while in CONDITION]"

Environmental: " The SYSTEM shall [exhibit] CHARACTERISTIC during/after exposure to ENVIRONMENT [for EXPOSURE DURATION]

Suitability: "The SYSTEM shall exhibit CHARACTERISTIC with PERFORMANCE while CONDITION [for CONDITION DURATION]"

end;

requirement Requirement_name2:

Id : “text string”;

Responsible stakeholder: “text string”;

Priority: High|Medium|Low;

Type:Functional|Nonfunctional|Performance|Design(constraint)|Environmental|Suitability;

Level: System requirement|Component requirement;

Role: Design variable|Design variable bound|Input parameter|Contraint|Objevtive;

Text：

Functional: "The SYSTEM shall [exhibit] FUNCTION [while in CONDITION]"

Performance: "The SYSTEM shall FUNCTION with PERFORMANCE [and TIMING upon EVENT TRIGGER] while in CONDITION "

Design: "The SYSTEM shall [exhibit] DESIGN CONSTRAINTS [in accordance with PERFORMANCE while in CONDITION]"

Environmental: " The SYSTEM shall [exhibit] CHARACTERISTIC during/after exposure to ENVIRONMENT [for EXPOSURE DURATION]

Suitability: "The SYSTEM shall exhibit CHARACTERISTIC with PERFORMANCE while CONDITION [for CONDITION DURATION]"

end;

requirement Requirement_nameN:

Id : “text string”;

Responsible stakeholder: “text string”;

Priority: High|Medium|Low;

Type:Functional|Nonfunctional|Performance|Design(constraint)|Environmental|Suitabi

lity;

Level: System requirement|Component requirement;

Role: Design variable|Design variable bound|Input parameter|Contraint|Objevtive;

Text：

Functional: "The SYSTEM shall [exhibit] FUNCTION [while in CONDITION]"

Performance: "The SYSTEM shall FUNCTION with PERFORMANCE [and TIMING upon EVENT TRIGGER] while in CONDITION "

Design: "The SYSTEM shall [exhibit] DESIGN CONSTRAINTS [in accordance with PERFORMANCE while in CONDITION]"

Environmental: " The SYSTEM shall [exhibit] CHARACTERISTIC during/after exposure to ENVIRONMENT [for EXPOSURE DURATION]

Suitability: "The SYSTEM shall exhibit CHARACTERISTIC with PERFORMANCE while CONDITION [for CONDITION DURATION]"

end;

traciability：

componse(a1,b1);

trace(a2,b2);

derive(a3,b3);

verify(a4,b4);

satisfy(a5,b5);

refine(a6,b6);

map(a7,b7.c1);

source(a8,b8);

present(a9,b9);

responsible(a10,b10);

end;

5.2 Use case diagrams

5.2.1 Purpose

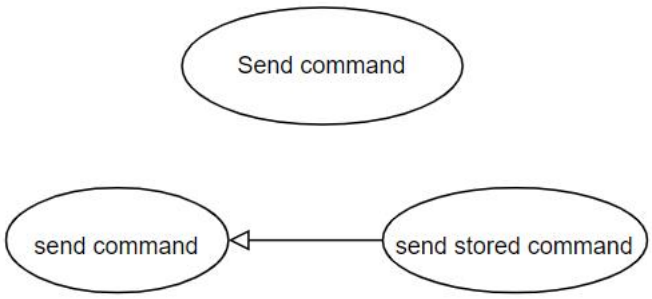
A use case diagram succinctly communicates a series of use cases - externally visible services provided by the system - and the executors who trigger and participate in the use cases. A use case diagram is a black-box view of the system, and as such is also well suited as a contextual diagram of the system.

5.2.2 When to create a use case diagram

A use case diagram is an analysis tool that is typically created early in the system life cycle. A system analyst may enumerate various use cases and then create use case diagrams during the conceptual and operational development phases of the system. In some approaches, the analyst creates use cases in a text-based, functional requirements manner during the requirements elicitation and specification phase of the system life cycle.

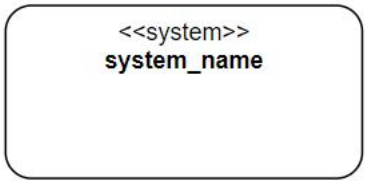
5.2.3 Use cases

The identification method of the use case diagram is an ellipse. The name of a generalized use case is a verb phrase (in the ellipse). Use cases can be generalized or specialized, which means you can create and display generalized relationships from one use case to another.



5.2.4 System boundaries

The system boundary represents the system that owns and executes the use case. The method of identifying the system boundary is a rectangular box around the use case. The name of the topic - shown at the top of the rectangle - must be a noun phrase.

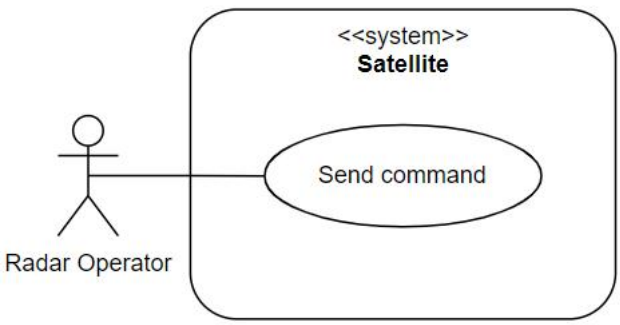


5.2.5 Implementers

There are two methods of identifying actors: matchstick miniatures, or rectangles with the <<actor>> keyword in front of their names. Generally modelers will use matchstick miniatures to represent people, and the rectangle identification method to represent systems.

5.2.6 Executor and use case association

An association is typically created between the executor and the use case. From there, it indicates that the executor interacts with the system to trigger and participate in the use case.

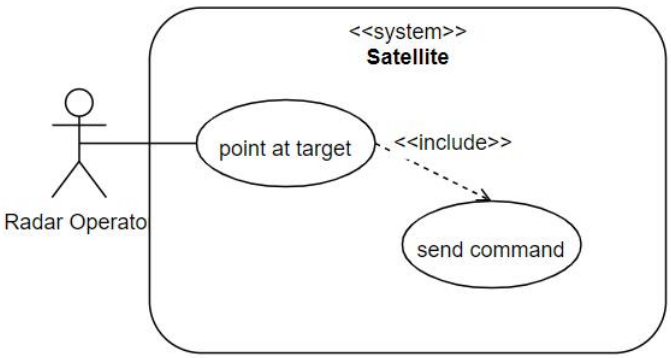


5.2.7 Base use cases

A base use case is an arbitrary use case that is connected to the main executor through an association relationship. This means that the base use case represents the goal of the main executor.

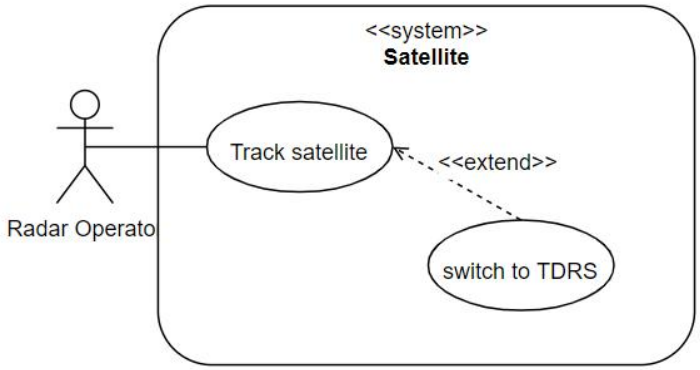
5.2.8 Embedded use cases

An include use case is any use case that is the target of an include relationship-that is, an element located at the end of an arrow. An include relationship is identified by a dotted line with an arrow and the keyword <<include>> next to it.



5.2.9 Extended use cases

An extension use case is any kind of use case. It is the source-located element at the end of the extension relationship. The extension relationship is identified by a dotted line with an arrow with the keyword <<extend>> nearby.



5.3 Definition diagrams

5.3.1 Purpose

Definitional diagrams are graphical representations of the structural attributes (input and output ports, parameters, state variables, etc.) of a particular class in the X language

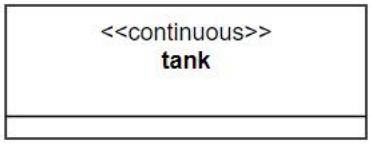
5.3.2 When to Create a Definition Map

When we model a system, we basically create definition diagrams. Definitional diagrams are usually combined with other diagrams to describe specific classes (e.g., continuous classes, discrete classes, etc.) of the X language.

5.3.3 Defining diagram elements and relationships

5.3.3.1 Class

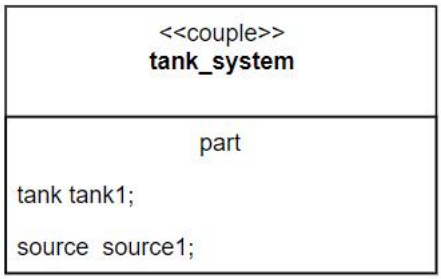
Classes (both general and specific) are the basic units in the structure of the X language. Classes can be used to create models for any type of entity of interest in the system or in an environment external to the system. Classes are generally rectangles with the metatype <<class|couple|continuous|discrete... >> rectangles.



The figure represents a continuous class model named tank

5.3.3.2 Component properties

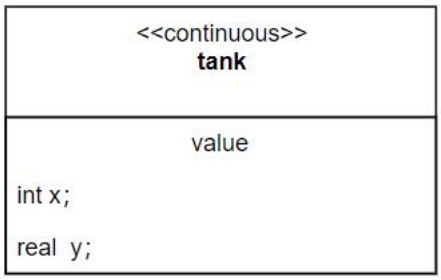
The component attribute represents the internal structure of the class (generally only the couple class has this attribute in the X language). In other words, classes are composed of component attributes. The relationship is one of belonging. Component attributes are typically represented by a structured separator box with the meta type part below the class.



The figure represents a couple class model named tank_system consisting of 1 model named tank1 and 1 model named source1. Where both tank1 and source1 are instantiations of a particular class tank and source created somewhere in the system.

5.3.3.3 Value attributes

The value attribute represents the state variable of the class, which can be of type integer, real, boolean, etc. The value attribute is typically represented by a structured separator box with the meta type value below the class.

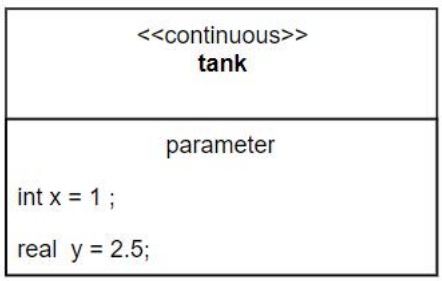


The figure represents a CONTINUOUS class model named tank having 1 integer variable named x and 1 real variable named y. The model has the following characteristics.

5.3.3.4 Parameter properties

The parameter attribute represents the instantiated parameters of the class, whose types can also be integer, real, boolean, etc.

The parameter attribute is typically represented by a structured separator box with the meta type parameter below the class.



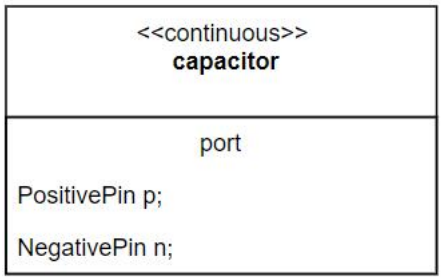
The figure represents a continuous class model named tank with 1 instantiated parameter named x with value 1 and 1 instantiated parameter named y with value 2.5.

5.3.3.5 ports

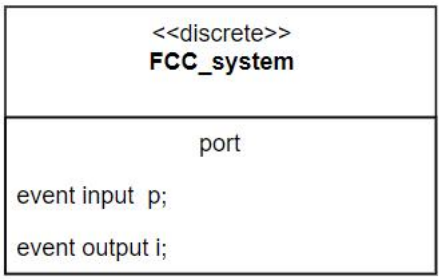
A port is a property that represents the different interaction points at the edges of a structure, through which external entities can interact with that structure (typically exchanging events, energy, data, etc.).

There are two types of ports in the X language, ports that follow generalized Kirchhoff's laws based on connector definitions, and event ports based on event exchanges.

Ports are generally represented by a structure-delimited box with the metatype port below the class.



The figure represents a continuous class model named capacitor having two connector type ports p and n instantiated with PositivePin and NegativePin respectively.

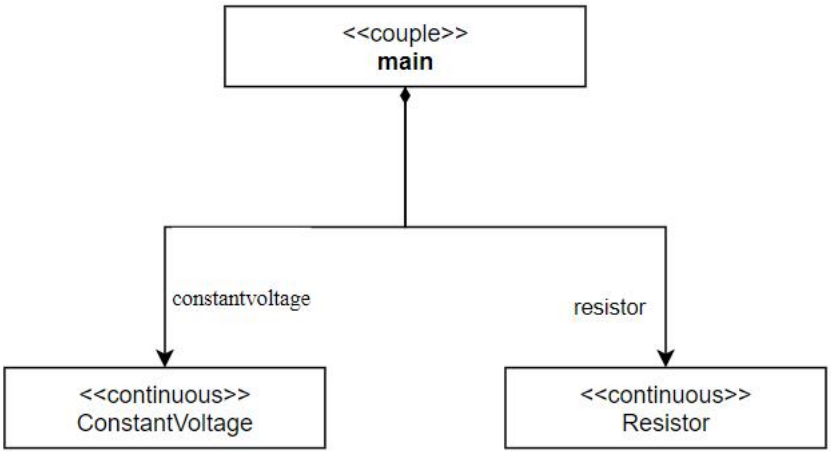


The figure represents a discrete class model named FCC_system having two event input and event output ports named p and i. The model has two event input and event output ports named p and i, respectively.

5.3.3.6 associative composition

Combinatorial associations are another representation of component properties. A combinatorial association between two classes represents a structural decomposition. An

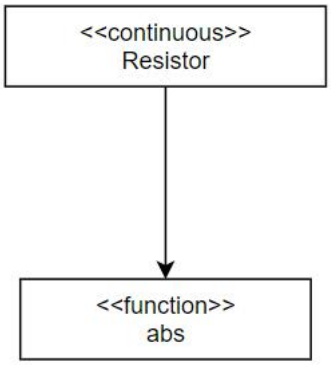
instance of a class at the combinatorial end is formed by combining some instances of classes at the component end. The method of identifying a combinatorial association in a definitional diagram is a solid line between two modules with a solid diamond at the combinatorial end.



The figure indicates that a coupled class model named main consists of an instantiated model constantvoltage of a continuous class model named ConstantVoltage and an instantiated model resistor of a continuous class named Resistor (note: the combinatorial association of the coupled classes also contains references to subclasses).

5.3.3.7 citation link

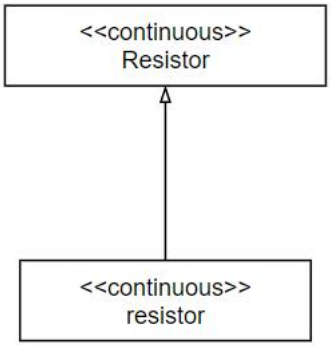
A reference association is a connection that exists between classes and represents a calling relationship between two classes. A reference association in the definition diagram is identified by a solid arrow between two modules.



The figure represents a continuous class model named Resistor calling a function class model named abs

5.3.3.8 generalization

Generalization is another relationship that is often shown in definition diagrams. This relationship represents the inheritance between two elements: a more generalized element, called a supertype, and a more specific element, called a subtype. Generalization is identified by a solid line with a hollow triangular arrow at one end of the supertype.



The figure indicates that the continuous class model named resistor is a generalization of the continuous class model named Resistor (inheriting all its properties)

5.4 Connection diagram

5.4.1 Purpose

A connection graph is created to specify the internal structure of a single class (typically a coupled class). A connection graph expresses the internal components of a class and how they interact to create valid instances.

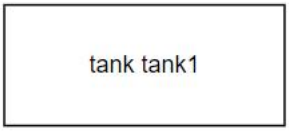
5.4.2 When to create a connection diagram

Connection diagrams are created when we need to establish the internal components of a composite system (coupled class) and its interaction logic. Connection diagrams are usually combined with definition diagrams to build coupled classes to describe a composite system.

5.4.3 Component properties

The component attributes of a connection graph are equivalent to the component

attributes in the component separator box in the corresponding definition graph. Components in a connection diagram are identified by rectangles with realized borders. The format of the strings displayed in the rectangular border is the same as the strings displayed in the component separator boxes in the definition diagram.

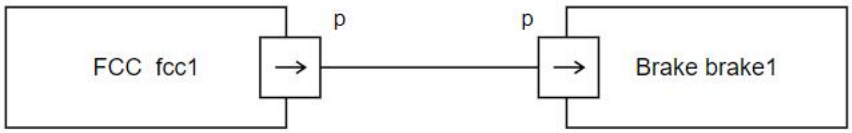


The figure represents a component of the tank_system in 2.3.3.2: the instantiation model of the tank class named tank1.

5.4.4 Connectors

The connectors of a connection diagram are equivalent to the port attributes in the port divider boxes in the corresponding definition diagram. Connectors in connection diagrams are identified in two ways: the first is a small rectangular box attached to the corresponding component; the other is a small rectangular box with a directional arrow attached to the corresponding component.

The figure represents that the instantiated port p of the connector class of source1 has an energy or data interaction with the instantiated port p of the connector class of tank1.



The figure indicates that the event output port p of fcc1 has an event interaction with the event input port p of brake1.

5.5 Equation maps

5.5.1 Purpose

An equation diagram is a type of diagram used to describe the continuous behavior of a continuous system. It is generally based on a system of differential equations to describe the behavioral constraints of a continuous system that changes continuously with time.

5.5.2 When to create a graph of equations

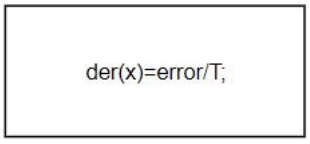
Equation diagrams are created when we build systems that involve continuous behavior over time with well-defined physical properties. Equation diagrams are generally combined with definitional diagrams to construct continuous classes to describe a continuous system.

5.5.3 Types of equations

There are five types of equations in the X language: basic equations, initial equations, conditional equations, event equations, and loop equations. Each of them is described below.

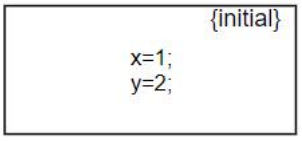
5.5.3.1 Fundamental equations

The underlying equations are generally in the form of differential algebraic equations (sets).



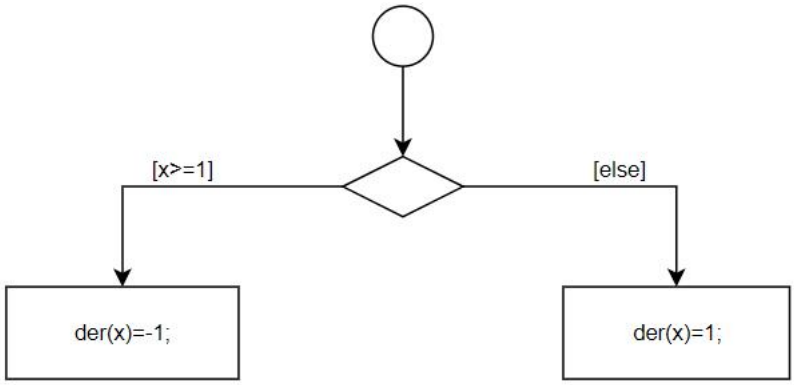
5.5.3.2 Initial equations

Initial equations are generally in the form of algebraic equations. It is modeled graphically with the {initial} symbol in the upper right.



5.5.3.3 Conditional equations

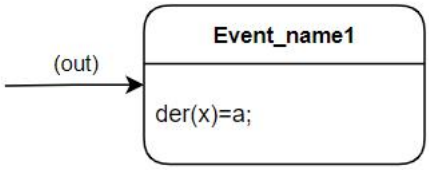
Conditional equations are forms of differential algebraic equations with if-else judgments. The two-branch description is as follows; (same for the multi-branch description)



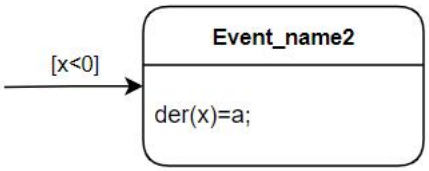
5.5.3.4 Event equations

Event equations come in two forms: external events and two behavioral equations triggered by internal events.

External Events:

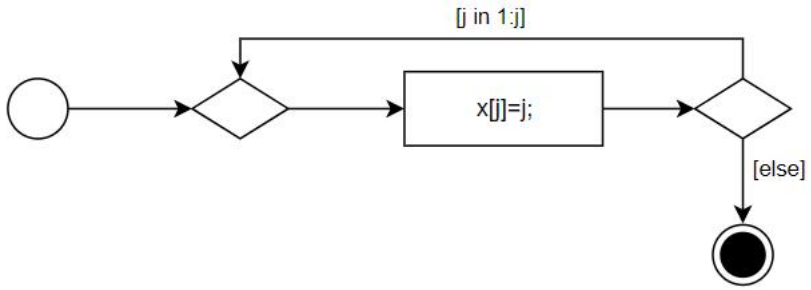


Internal events:



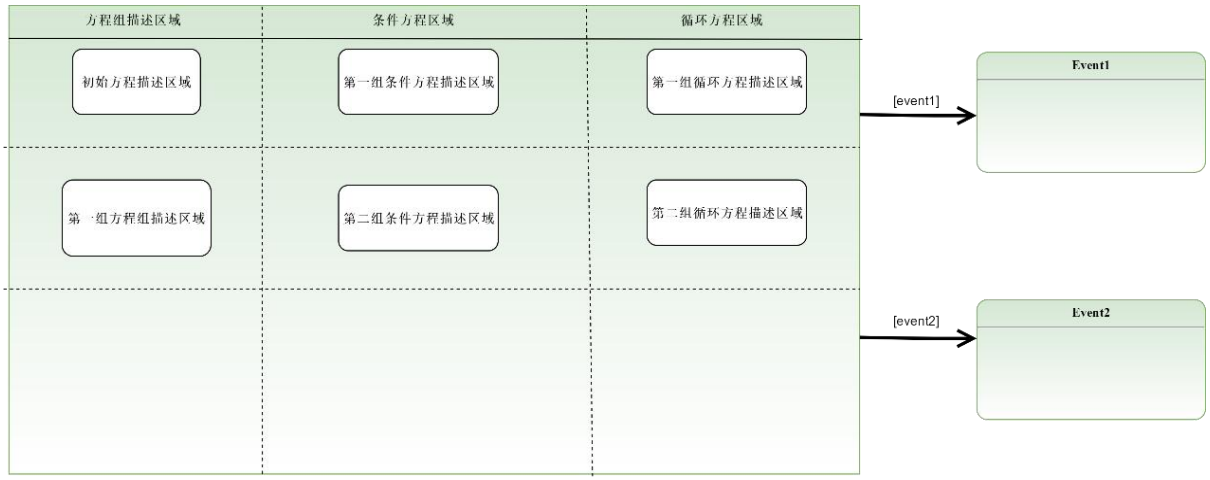
5.5.3.5 Cyclic equations

A cyclic equation is a convenient descriptive form for describing differential algebraic equations that have the same descriptive form.



5.5.3.6 Equation diagram architecture

When the behavior of the continuous system to be described is described by a combination of equations in more than one of the above descriptive forms, the construction of its equation map is carried out in the architecture of the following figure.



5.6 State Machine Diagrams

5.6.1 Purpose

A state machine diagram is a type of diagram used to describe the discrete behavior of a discrete system. It is generally based on the transition mechanism of events advancing the state of the system to describe the behavioral constraints of discrete systems with discrete changes triggered by events.

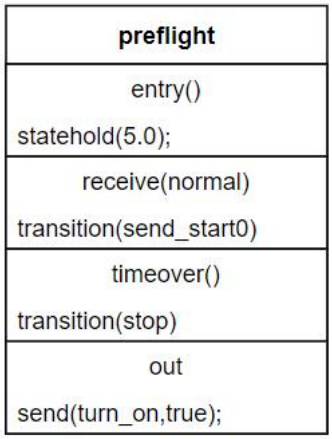
5.6.2 When to create a state machine diagram

State machine diagrams are created when we build systems that involve discrete behaviors triggered by events. State machine diagrams are typically combined with

definitional diagrams to build discrete classes to describe a discrete system.

5.6.3 Status

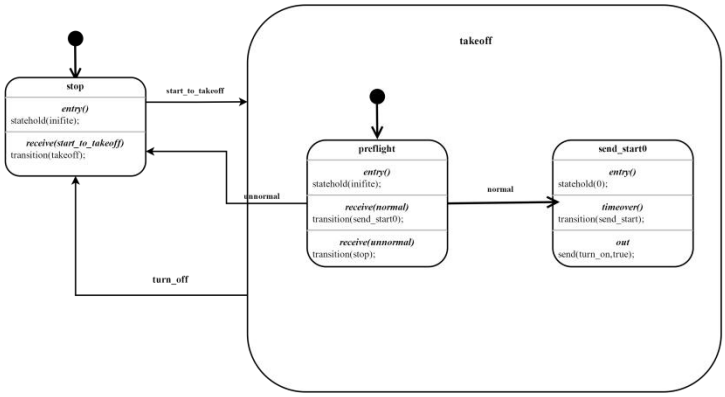
A system will sometimes have a defined set of states that it can be in during the operation of the system. A normal state will include, at a minimum, the duration of being in the state, receiving external events, and behaviors and outputs triggered by internal events.



The figure indicates that the system in the preflight state has a duration of 5s when no event is triggered, when an event named normal is received, it will be transferred to the state named send_start0; when an internal event is triggered i.e., after 5s has elapsed, it will be transferred to the state named stop and the value of the output event port, turn_on, will be assigned to true for output.

5.6.4 Composite state

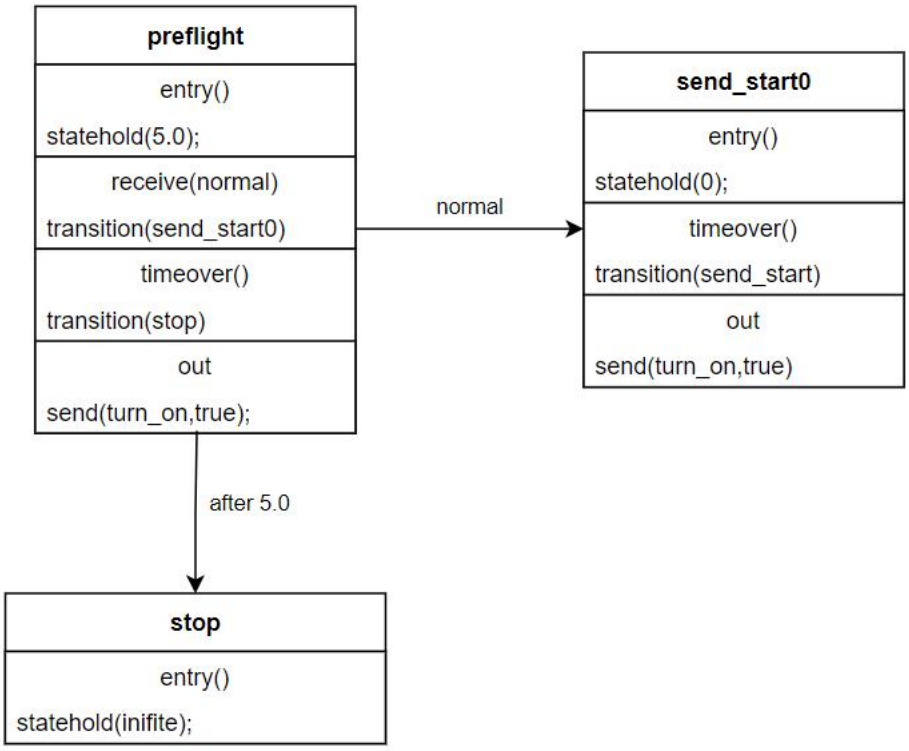
A composite state will generally contain a number of sub-states within it, and when the composite state is inactive, all of its sub-states are inactive. When the composite state is active, then one of its sub-states will be active. In an active state, the composite state responds to events by transferring from one sub-state to another. The form of transitions between sub-states is the same as the form of transitions between states. In addition, composite states may jump from their boundaries or from specific embedded sub-states.



The takeoff in the figure is a composite state, when it receives the event start_to_takeoff, the takeoff is activated, the initial state preflight of its internal sub-state identifier is activated, and when it receives the event normal it will normally undergo a state transition within the composite state to the send_start0 state, and then when it receives the When it receives the event normal, it jumps out of the composite takeoff state and transitions to the stop state. In addition, regardless of any sub-state within the composite takeoff, when it receives the event turn_off, it will directly jump out of the current state to the stop state.

5.6.5 Conversion

A transition represents a change from one state to another. Transitions are identified by solid lines with open arrows drawn from the source vertex to the target vertex. Transitions are labeled with the event that triggered them (internal events are labeled as after time, external events are labeled with the event name).



The transition arrows in the figure indicate that the system accepts the external event NORMAL when it is in the PREFLIGHT state and transfers to the SEND_START0 state; it will transfer to the STOP state after 5s have passed.

5.6.6 Types of events

There are two types of state events in the X language: external events and internal events.

External Events

External events are generally signal events that are input from an external system. A signal event represents the process of accepting a signal instance by a target system that is capable of accepting it. If the state machine has transitions with signal event triggers, then the system executing the state machine must have receptions with the same name.

Internal Events

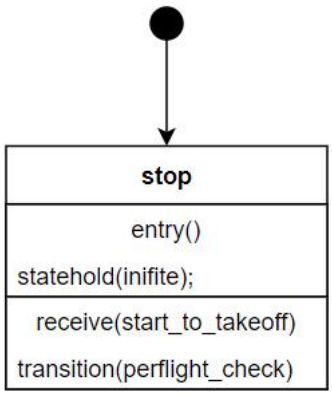
Internal events are generally internal time events. A time event is intuitive in that it represents an instance in time. A time event occurs when that moment arrives during the operation of the system.

5.6.7 Pseudo-states

The difference between pseudo-states and states is that a state machine can be suspended in a state, but not in a pseudo-state. The reason for adding pseudo-states to a state machine is to specify control logic on transitions between states.

5.6.7.1 initial pseudostate

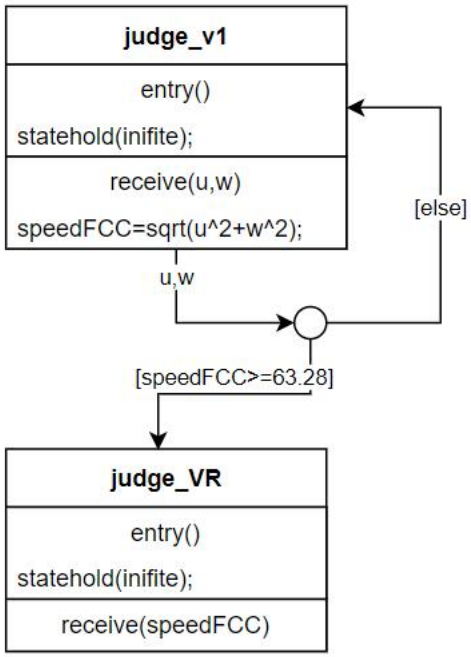
Initial pseudo-states indicate the first state when state execution begins. Initial pseudo-states in X are simply identifiers that signal that the initial state is about to begin. The initial pseudo-state is identified by a small solid circle.



The figure represents the initial state of the system as STOP

5.6.7.2 Connection Pseudo-State

Connecting pseudo-states can combine multiple transitions between states into a single composite transition. A connected pseudo-state is also identified by a small solid circle. Unlike initial pseudo-states, connected pseudo-states must have one or more input transitions, and one or more output transitions.



The figure indicates that when the judge_v1 state receives an external event (u,w) it transfers to the connection pseudo state and then performs a conditional judgment i.e. when speedFCC>=63.28, it transfers to the state judge_VR, otherwise it transfers to its own state.

5.7 Activity diagrams

5.7.1 Purpose

An activity diagram is a behavioral graph used to describe the sequence of behaviors and events that occur. It is generally used to describe the algorithms of PLAN in the classes of functions and intelligences.

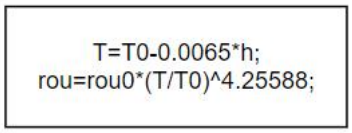
5.7.2 When to create an activity diagram

Activity diagrams are created when we build systems that involve functions that need to be called or plan that builds classes of intelligences. Activity diagrams are usually combined with definition diagrams to build function classes to describe the functions called in a system.

5.7.3 Actions

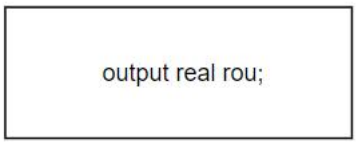
An action is a type of node that can exist within an activity; it is a node that models the

basic functional unit of the activity. An action represents some type of processing or transformation that occurs when an activity is executed during system operation. Actions are identified by rectangles. In the X language, activities are generally described as statements in text form.



5.7.4 Activity parameters

An activity parameter represents an input or output of the activity in a general way. Activity parameters are identified by a rectangular method and are generally described with input/output.

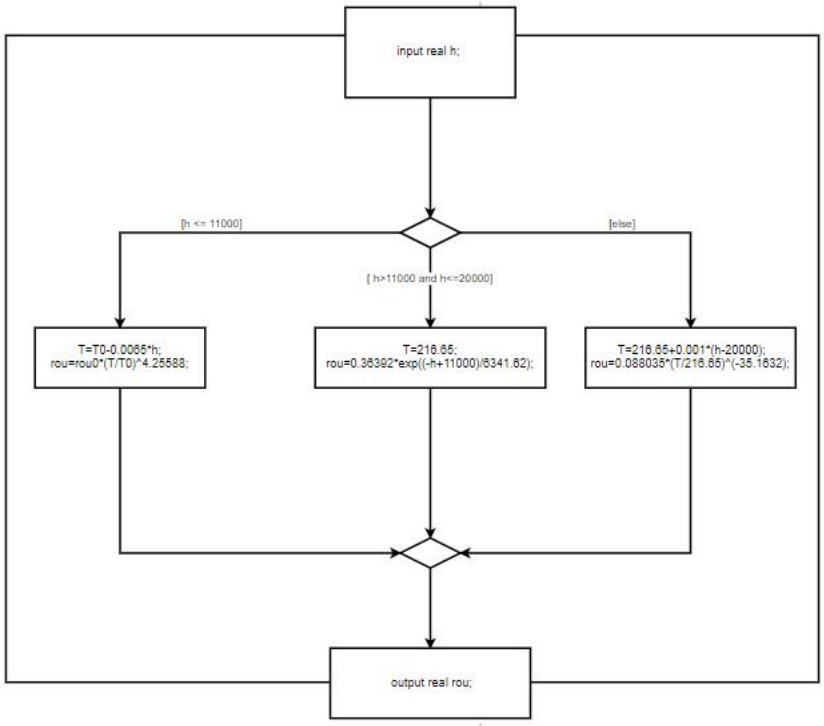


5.7.5 Control nodes

Using control nodes, activities can be directed to execute along a path. There are 2 types of control nodes: decision nodes, and merge nodes.

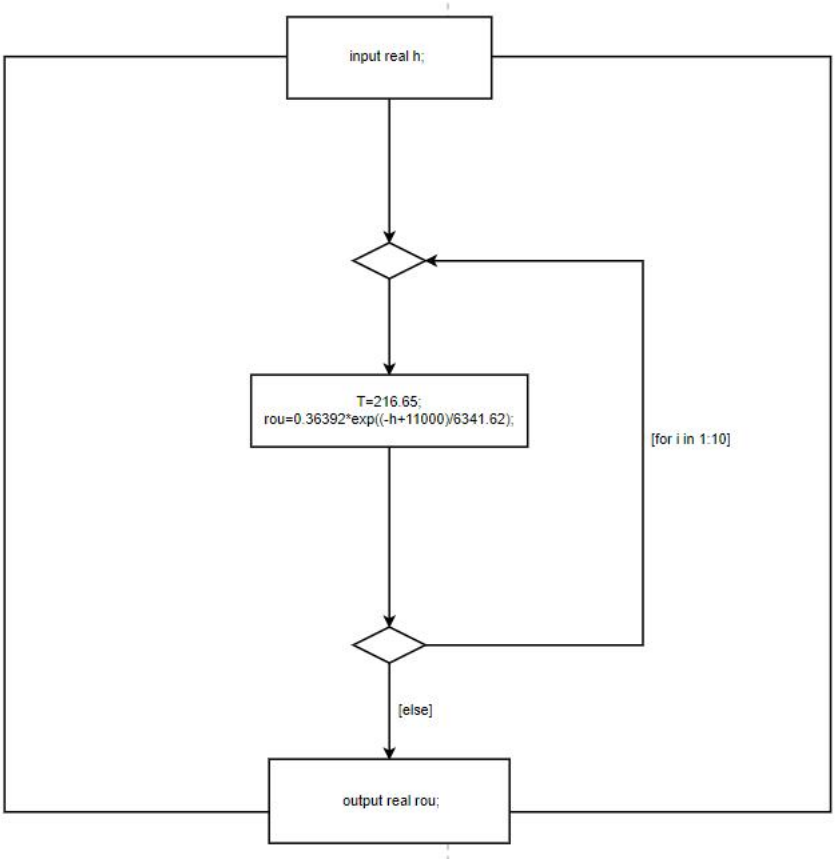
5.7.5.1 Decision nodes

Determines the beginning of the replaceable sequence in the node labeling activity. Its marking method is a hollow diamond. Decision nodes must have a single input edge, and typically have two or more output edges, each of which will carry a Boolean expression shown as a string in the middle of square brackets.



5.7.5.2 Merging nodes

The merge node marks the end of the optional sequence in the activity. Its marking method is the same as that of the decision node: a hollow rhombus. Merge nodes have two or more input edges and only one output edge. Generally merge nodes are used in combination with decision nodes to model loops in the activity.



5.8 Model Structure Diagram

5.8.1 Purpose

A model structure diagram is a graphical representation of the system model file architecture (file structure) in the X language. The system model is organized by the hierarchical relationships of the system. There is no single correct structure for a system model; different approaches will suggest different model structures, and the same model structure will produce different results depending on the goals of the project. Once a valid model structure for the system has been determined, creating a model structure diagram will provide an easy-to-understand view of the project.

5.8.2 When to Create Model Structure Diagrams

At the beginning of the project, based on system and stakeholder concerns, a model structure diagram is defined to describe the initial architecture of the system. Then, as the

design changes, new architectures will be defined, and the model structure diagram will be updated to decompose higher-level architectural elements into lower-level architectural elements.

5.8.3 Elements and Relationships of Model Structure Diagrams

A model is the basic element of a model structure diagram. Models can be defined as concrete X-language types for describing a specific kind of process or item, or as abstract elements for describing objects with a certain characteristic or property. Models are used in model structure diagrams using a folder-like graphic with the model name, which is labeled in the upper-left corner.

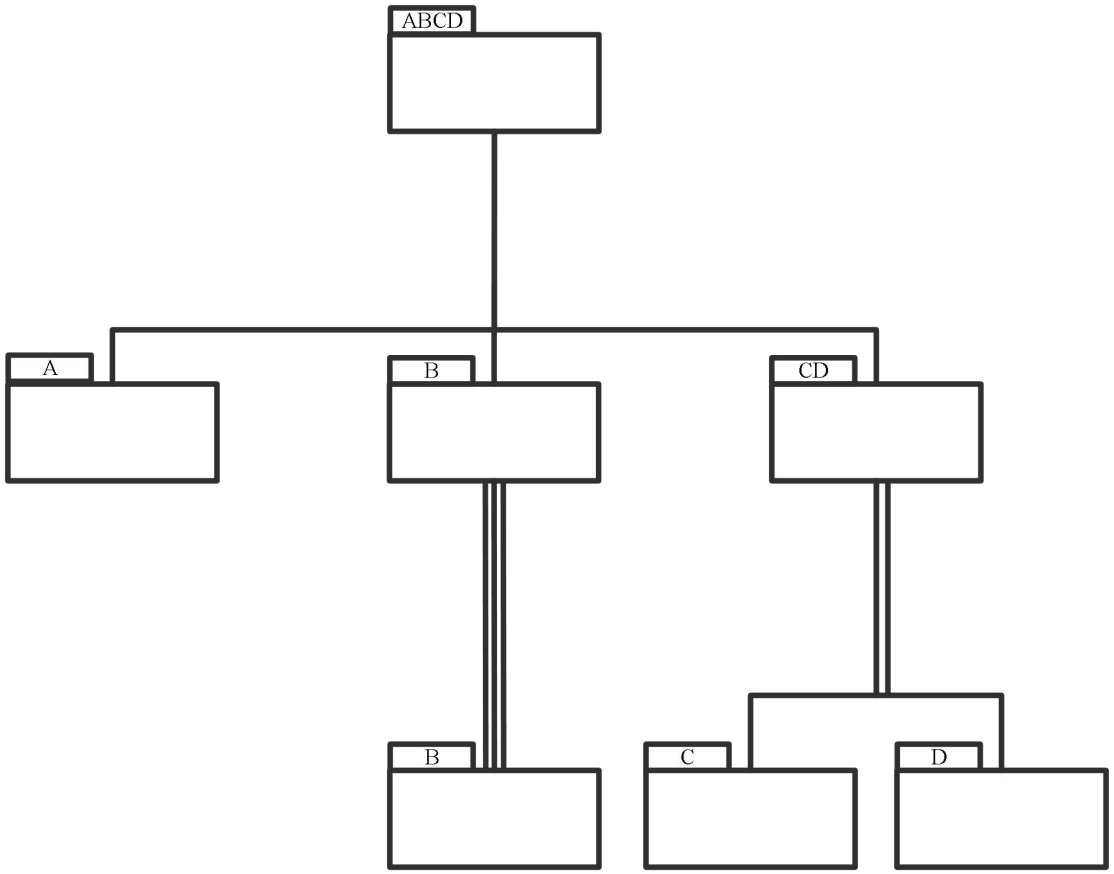
The figure represents an entity named ABCD

5.8.3.1 Entity Architecture Portfolio Approach

1) Combination of Connections

Model elements are combined and connected in the model structure diagram using three ways. These are: aspect, multifaceted and specialization.

An aspect connection expresses that a subsequent model describes an aspect of a model, i.e., it describes that the previous model node is composed of subsequent model nodes, denoted by |. Similarly, a multifaceted connection expresses that its preceding node consists of multiple models of the same type, denoted by |||, and the number of models can be determined when system instantiation is required. A specialization connection consists of a bi-vertical line connecting to a model, denoted by ‖, which describes a specialization of a model, i.e., a different implementation of a model, a model may have multiple specializations, and a unique specialization is also determined at the time of system instantiation. The three combinations are shown below.

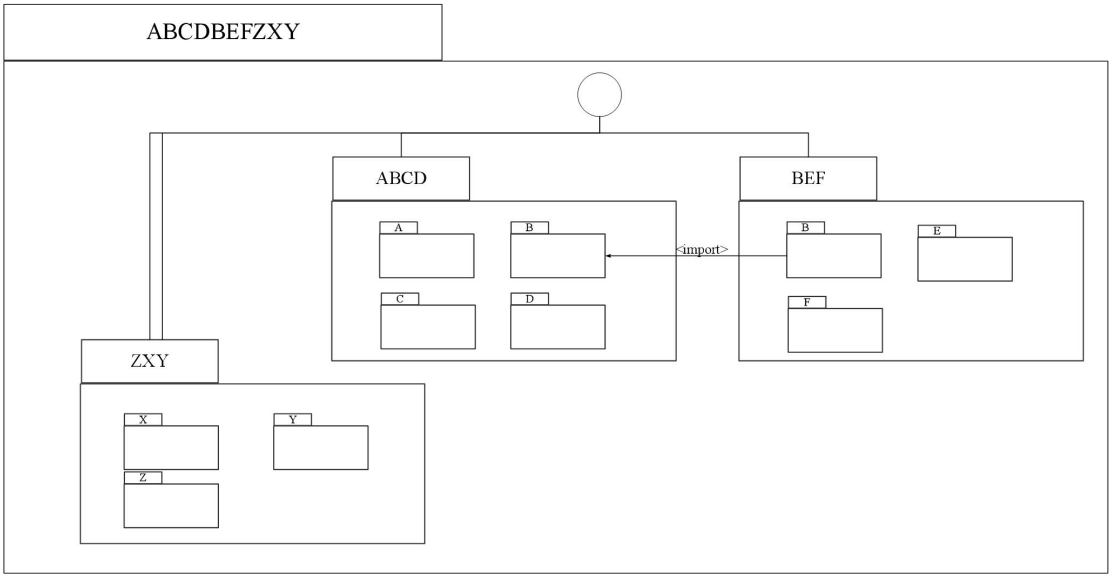


The diagram depicts three types of connections

The ABCD entity in the diagram consists of three aspects, A, B, and CD, where B can include many of them, and two special descriptions, C and D, exist for the CD aspect.

2)Graphic inline combination method

In addition to the direct connection approach, the X language model structure diagram also provides an inline approach to models, which facilitates model file management by modelers. In this description, if there are other models defined within a model, then these models are aspects or specializations of this upper model, and the specific relationships are represented by connecting lines, except that the connecting lines are changed from direct connections to connecting the ontology of this model, which is represented by the use of circular icons. ABCD and BEF as shown below are two aspects of ABCDBEFZXY, and then ZXY is a specialization of ABCDBEFZXY. In particular, if the submodels within a model do not have any connecting lines, it means that all these submodels are only aspects of that model. For example, A, B, C, and D are all aspects of ABCD.

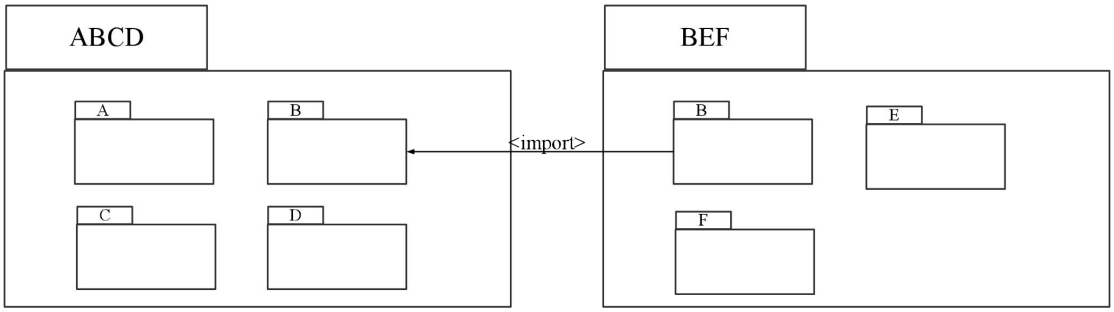


5.8.4 Namespaces

In general. an entity is the owner (includer) of all of its subentities displayed in the content area one-by-one unless another namespace inclusion relationship is explicitly shown on the diagram. the X language supports the use of the qualified name identifier method to characterize the namespace in which the model resides. For example, the string ABCD.D,. D, the named element is located at the end of the string. The "." in the string indicates that the D package is contained in the entity ABCD.

5.8.5 Model references

Because the creation of model hierarchical relationships may introduce packages (and their contents) from other models and packages into other systems, the X language architecture diagram provides a mechanism to express that one package introduces the contents of another package. This introduction relationship is identified by a solid line with an open arrow and the keyword <import> near the line, as shown in the following figure, where ABCD introduces entity B from BEF.


ABCD
A
B
C
D
<import>
BEF
B
E
F

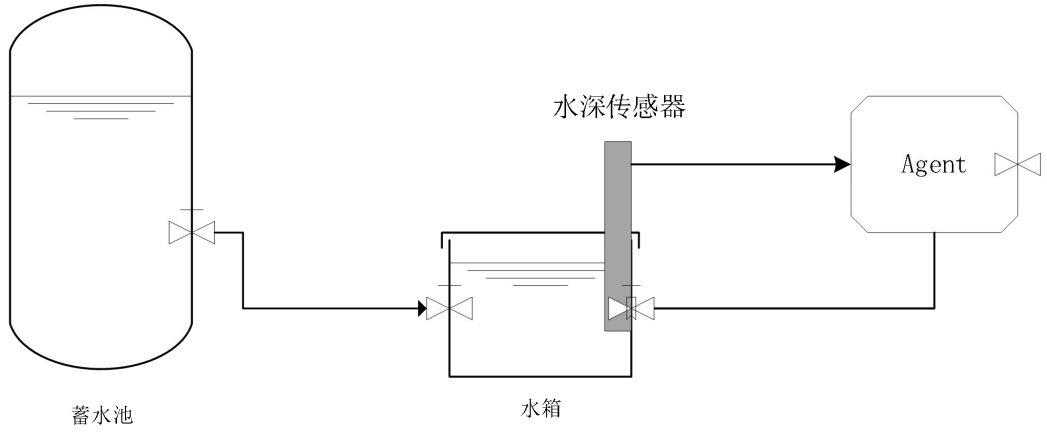
6 X Language Modeling Case Study

6.1 Water Tank Model

6.1.1 Background Description

The model consists of three components: a water source, a water tank, and an intelligent agent (PI controller). The water tank acquires water from the water source and is then regulated by the PI controller to maintain the water level within a certain range.

In the reservoir model, the output water flow is relatively small when the time is less than 150 seconds. After 150 seconds, the output increases threefold. Therefore, it can be observed that after the initial stabilization, the water level in the tank increases again around 150 seconds and is once again stabilized by the PI controller.

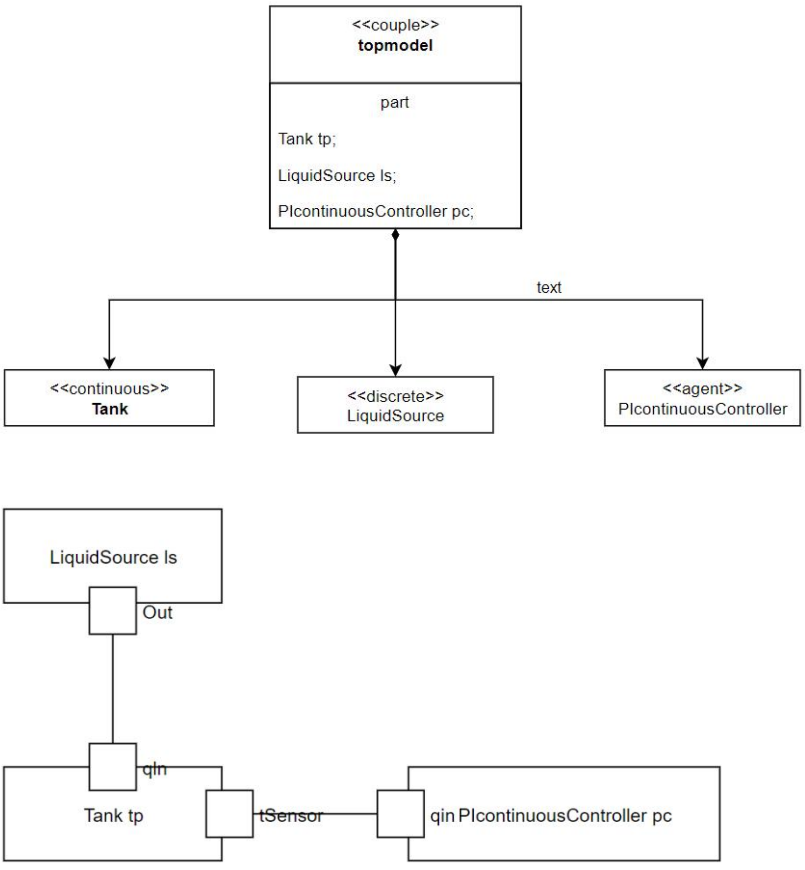


6.1.2 Modeling Analysis

Firstly, establish the top-level system model, *topmodel*, which describes all the subsystem models included in the entire model system. These subsystem models include the *tank* (water tank), *LiquidSource* (water source), and the intelligent agent controller *PIcontinuousController* model. The top-level model should also describe the connection relationships between the various subsystem models.

For the subsystem models, the tank model primarily describes specific parameters such as tank volume and equations governing the inflow and outflow of water. The LiquidSource model mainly depicts the state transitions of the reservoir among various output water flow rates. The intelligent agent model describes the relevant equations for water level control.

6.1.3 System Model

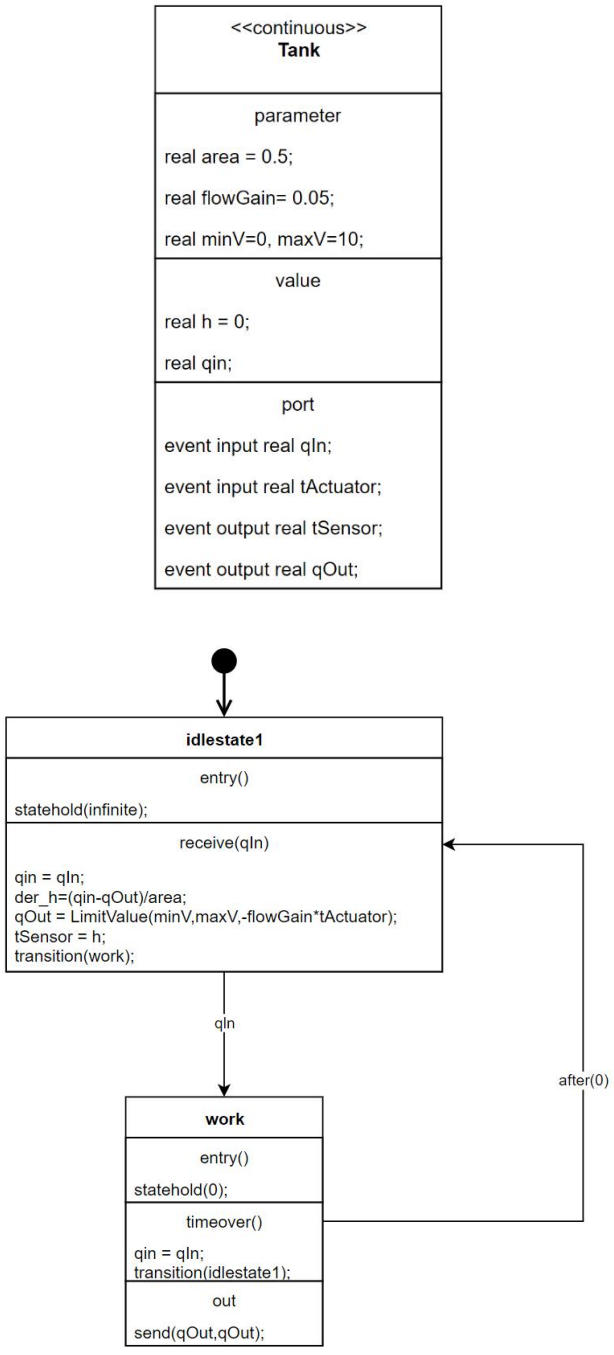



```
couple topmodel
    import Tank;
    import LiquidSource;
    import PIcontinuousController;
    part:
        Tank tp;
        LiquidSource ls;
        PIcontinuousController pc;
    connection:
        connect(ls.Out,tp.qIn);
        connect(tp.tSensor,pc.qin);
```

end;

6.1.4 Subsystem Model

Tank Model - *Tank*:



continuous Tank

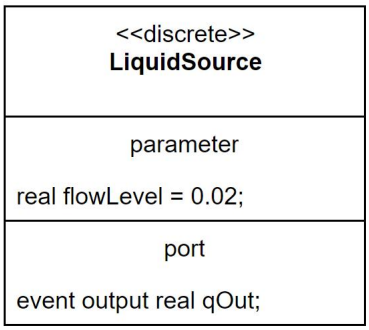
import LimitValue;

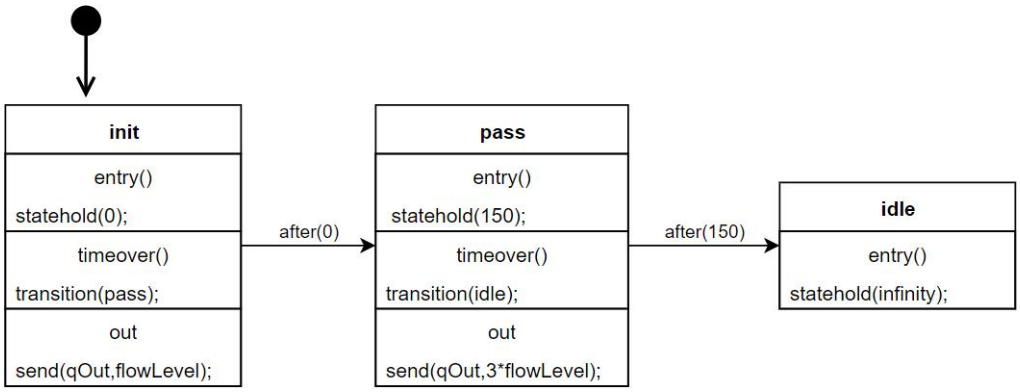
parameter:

real area = 0.5;

```
        real flowGain= 0.05;
        real minV=0, maxV=10;
    value:
        real h = 0;
        real qin;
    port:
        event input real qIn;
        event input real tActuator;
        event output real tSensor;
        event output real qOut;
    state:
        initial state idlestate1
            when entry() then
                statehold(infinite);
            end;
            when receive(qIn) then
                qin = qIn;
                        der_h=(qin-qOut)/area;
                        qOut = LimitValue(minV,maxV,-flowGain*tActuator);
                        tSensor = h;
                        transition(work);
            end;
        end;
        state work
            when entry() then
                statehold(0);
            end;
            when timeover() then
                qin = qIn;
                        transition(idlestate1);
            out
                send(qOut,qOut);
            end;
        end;
```

Liquid Source Model - *LiquidSource*：

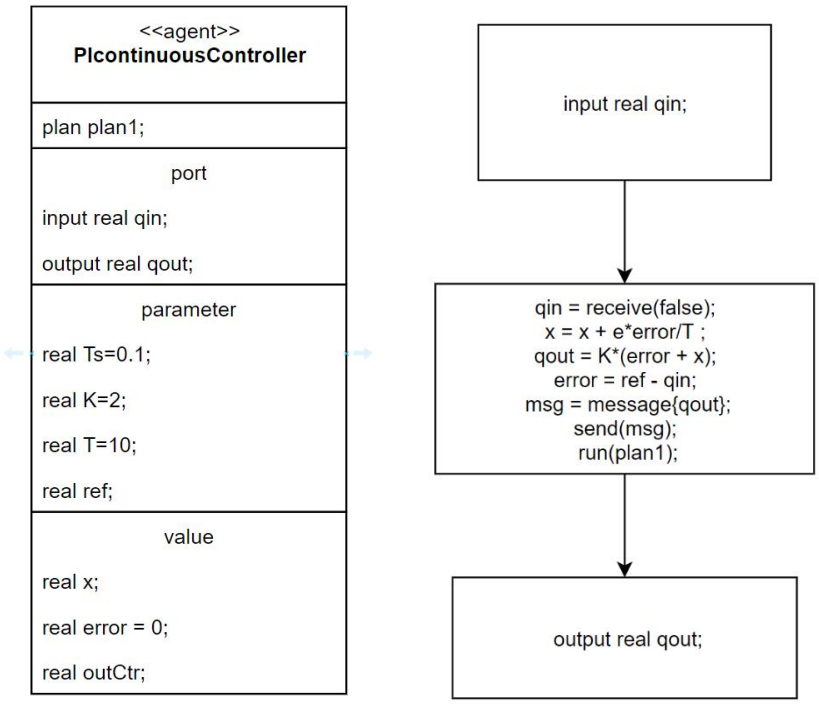





```
discrete LiquidSource
    parameter:
        real flowLevel = 0.02;
    port:
        event output real qOut;
    state:
        initial state init
            when entry() then
                statehold(0);
            end;
            when timeover() then
                transition(pass);
            out
                send(qOut,flowLevel);
            end;
        end;
        state pass
            when entry() then
                statehold(150);
            end;
            when timeover() then
                transition(idle);
            out
```

```
            send(qOut,3*flowLevel);
        end;
    end;
    state idle
        when entry() then
            statehold(infinity);
        end;
    end;
```

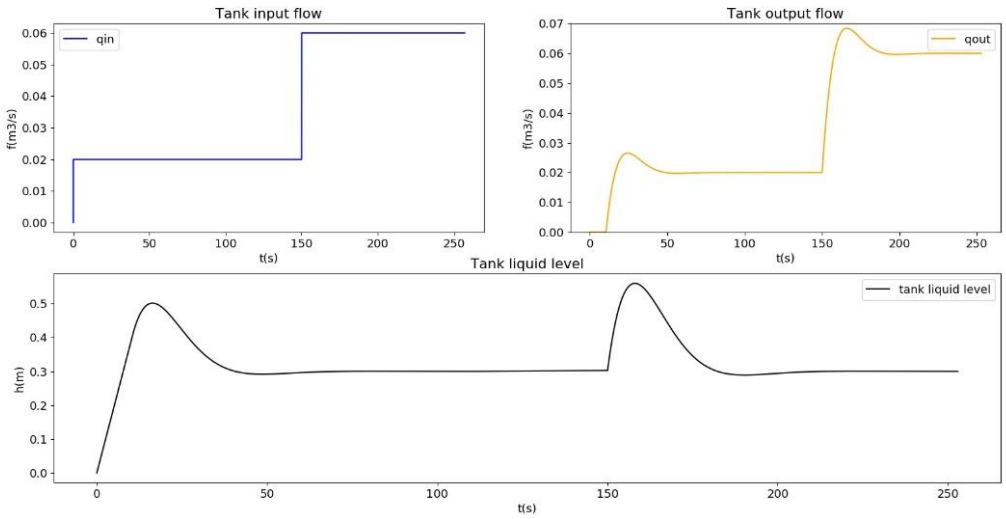
Intelligent Agent Model - *PIcontinuousController*:




```
agent PIcontinuousController
    plan plan1;
port:
    input real qin;
    output real qout;
parameter:
    real Ts=0.1;
    real K=2;
    real T=10;
    real ref;
value:
    real x;
    real error = 0;
    real outCtr;
```

```
action:
        qin = receive(false);
        x = x + e*error/T ;
        qout = K*(error + x);
        error = ref - qin;
          msg = message{qout};
        send(msg);
          run(plan1);
end;
```

6.1.5 Simulation Results



6.2 Watchdog Model

6.2.1 Background Description

The watchdog system is used to enhance the anti-interference capability of the program. If the microcontroller experiences program "runaway" under specific interference, the watchdog program can automatically reset the microcontroller, preventing the program from entering an infinite loop. The watchdog system is often connected to the input/output pins of the microcontroller, and signals controlled by the program pass through these pins to control the state changes of the watchdog system. If the system gets stuck in an infinite loop, and the watchdog circuit detects abnormal signals, it triggers an alarm mechanism to automatically

reset the system program. The state transition of this process is shown in Figure 2.1.

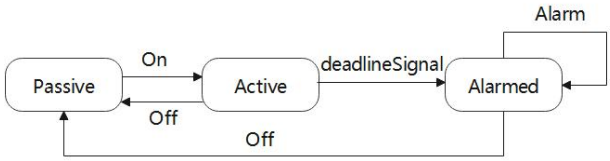


Figure 2.1 Watchdog System State Transition

The watchdog system includes the watchdog circuit and three connected interface terminals that control the input of three signals, regulating the system's transition between different states. It has two basic states, passive and active, but can also enter a third state, triggering an alarm upon receiving a *deadline* event. When entering the third state, the watchdog immediately issues an alarm.

The watchdog system comprises the watchdog model itself and three event generators, as illustrated in Figure 2.2.

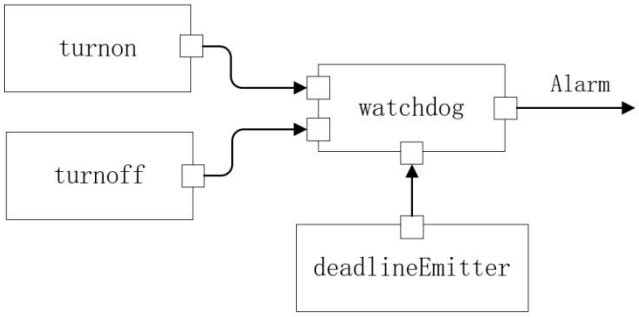
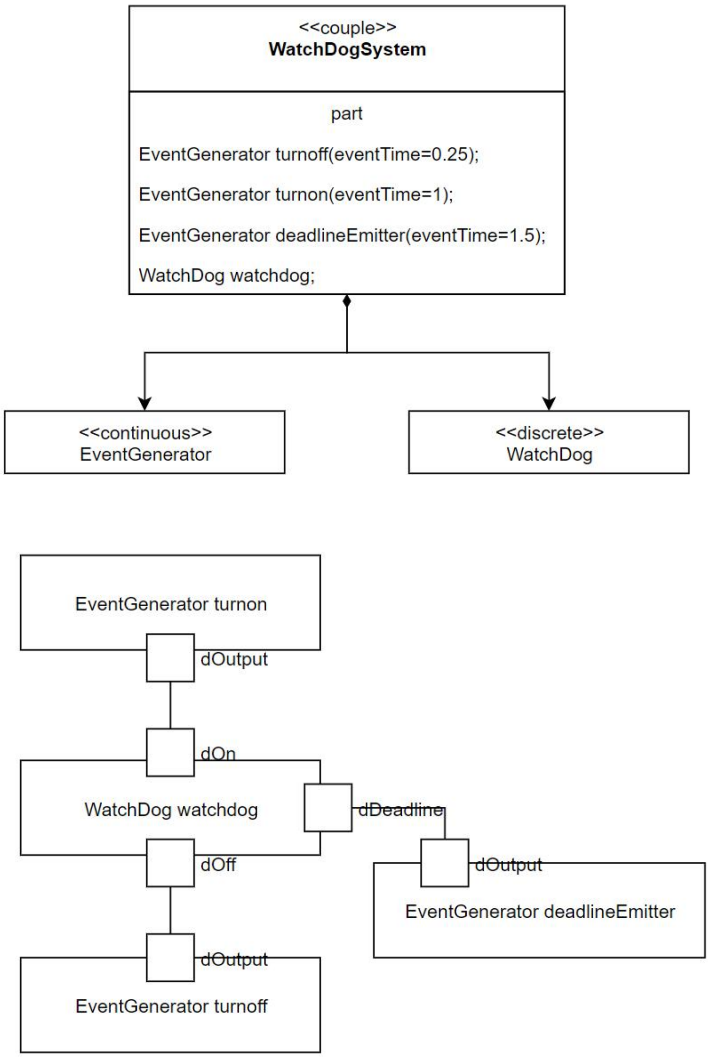


Figure 2.2 Watchdog System Model

6.2.2 Modeling Analysis

Firstly, establish the top-level system model *WatchDogSystem*, integrating discrete and continuous modules using coupling classes, and express the relationships between various subsystem models in this top-level model.

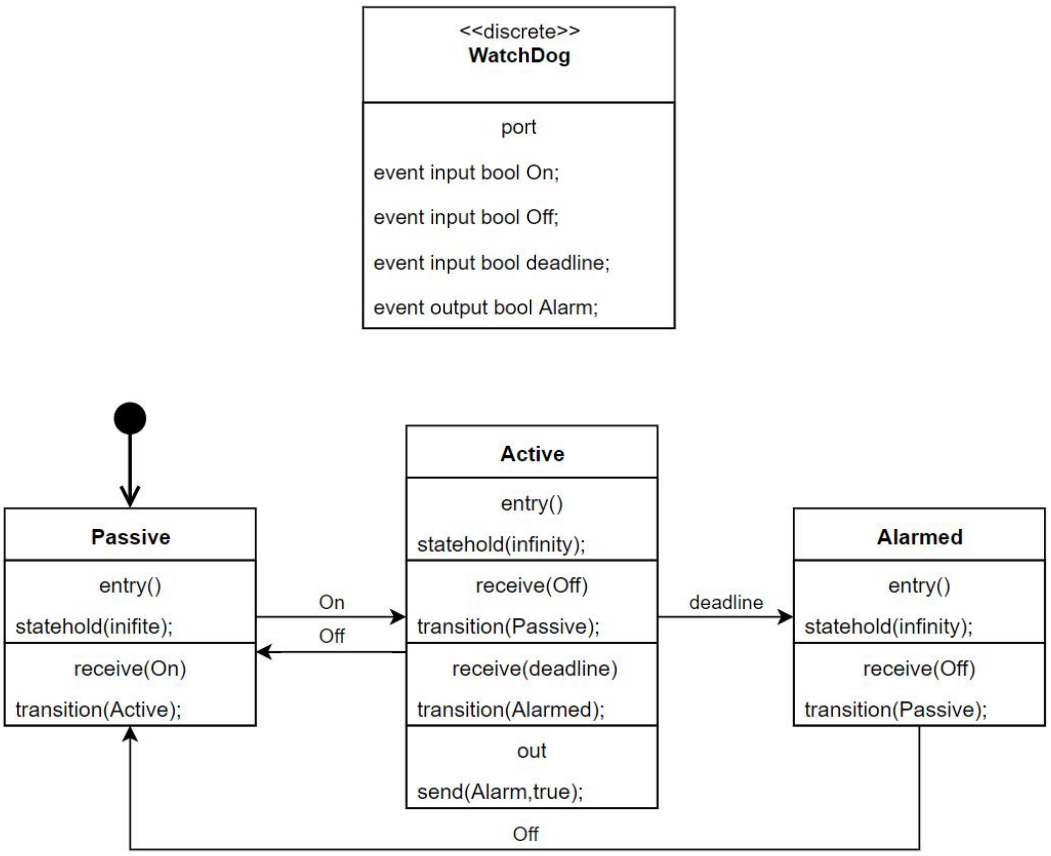
6.2.3 System Model




```
couple WatchDogSystem
    import EventGenerator;
    import WatchDog;
part:
    EventGenerator turnoff(eventTime=0.25);
    EventGenerator turnon(eventTime=1);
    EventGenerator deadlineEmitter(eventTime=1.5);
    WatchDog watchdog;
connection:
    connect(turnon.dOutput, watchdog.dOn);
    connect(turnoff.dOutput, watchdog.dOff);
    connect(deadlineEmitter.dOutput, watchdog.dDeadline);
end;
```

6.2.4 Subsystem Model

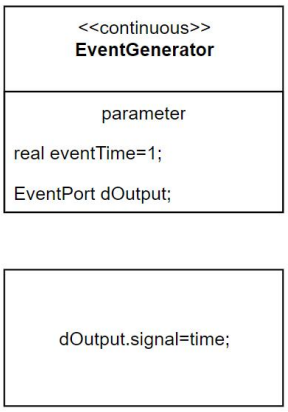
***WatchDog*:**



discrete WatchDog
port:
    event input bool On;
    event input bool Off;
    event input bool deadline;
    event output bool Alarm;
state:
initial state Passive
    when entry() then
        statehold(inifite);
    end;
    when receive(On) then
        transition(Active);
    end;
end;
state Active
    when entry then

```
            statehold(infinity);
        end;
        when receive(Off) then
            transition(Passive);
        end;
        when receive(deadline) then
            transition(Alarmed);
            out:
            send(Alarm,true);
        end ;
    end;
    state Alarmed
        when entry then
            statehold(infinity);
        end;
        when receive(Off) then
            transition(Passive);
        end;
    end;
    end;
```

***EventGenerator*：**



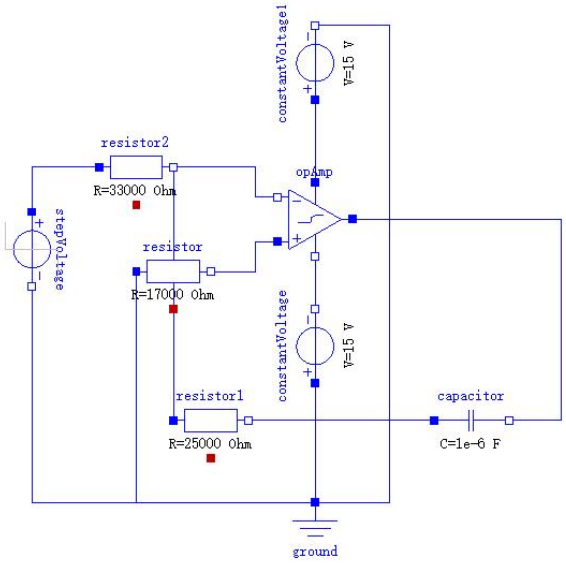

```
continuous EventGenerator
parameter:
  real eventTime=1;
  EventPort dOutput;
equation:
  dOutput.signal=time;
end;
```

6.2.5 Simulation Results

6.3 Circuit Model

6.3.1 Background Description

Modeling and simulating the circuit model provided below. This case is a global continuous class case, primarily modeling various modules using *continuous*.



The modeling approach is to use *continuous* to model various circuit components. In the top-level circuit coupling module, instantiate various circuit component classes and connect them to form the final circuit model.

6.3.2 System Model

```
couple main
import OpAmp;
import Resistor;
```

```
import Ground;
import ConstantVoltage;
import StepVoltage;
import Capacitor;
part:
OpAmp opAmp;
Resistor resistor(R = 17000);
Ground ground;
ConstantVoltage constantVoltage(V = 15);
ConstantVoltage constantVoltage1(V = 15);
Capacitor capacitor(C = 1e-6);
Resistor resistor1(R = 25000);
Resistor resistor2(R = 33000);
StepVoltage stepVoltage(V = 1);
connection:
   connect(constantVoltage.p, ground.p);
   connect(constantVoltage.n, opAmp.VMin);
   connect(constantVoltage1.p, opAmp.VMax);
   connect(constantVoltage1.n, ground.p) ;
   connect(resistor.n, opAmp.in_p);
   connect(opAmp.out, capacitor.n) ;
   connect(capacitor.p, resistor1.n) ;
   connect(resistor2.n, opAmp.in_n) ;
   connect(resistor2.n, resistor1.p) ;
   connect(stepVoltage.p, resistor2.p) ;
   connect(stepVoltage.n, ground.p) ;
   connect(resistor.p, ground.p)
end;
```

6.3.3 Subsystem Model

Capacitor Module

```
continuous Capacitor
import Interfaces.OnePort;
extends OnePort;
parameter:
real C=1;
initial equation:
v=0;
equation:
i=C*der(v);
```

```
end;
```

Resistor *Module*

```
continuous Resistor
import Interfaces.OnePort;
extends OnePort;
parameter:
real R=1;
equation:
v = R*i;
end;
```

Ground Module

```
continuous Ground
import Interfaces.Pin;
port:
Pin p;
equation:
    p.v=0;
end;
```

Step Voltage Source Module

```
continuous StepVoltage
import Interfaces.VoltageSource
extends VoltageSource
parameter:
real V=1;
equation:
y=offset + (if time < startTime then 0 else V);
end;
```

Continuous Voltage Source Module

```
continuous ConstantVoltage
import Interfaces.OnePort;
extends OnePort;
parameter:
real V=1;
equation:
v = V;
```

```
end;
```

Operational Amplifier Module

```
continuous OpAmp
import Interfaces.PositivePin;
import Interfaces.NegativePin;
parameter:
real slope=1000;
port:
PositivePin in_p;
NegativePin in_n;
PositivePin out;
PositivePin VMax;
NegativePin VMin;
value:
real vin;
real f;
real absSlope;
equation:
in_p.i = 0;
in_n.i = 0;
VMax.i = 0;
VMin.i = 0;
vin = in_p.v - in_n.v;
f = 2/(VMax.v - VMin.v);
if Slope<0 then
absSlope=-Slope;
else
absSlope=Slope;
end;
out.v = (VMax.v + VMin.v)/2 + absSlope*vin/(1 + absSlope*smooth(0, (if (f*vin < 0) then -f*vin else f*vin)));
end;
```

Interface or Reference Definition Module, placed in a file named "*Interfaces*", containing the following six classes:

Positive Terminal Module

```
connector PositivePin
port:
```

```
real v;
flow real i;
end;
```

Negative Terminal Module

```
connector NegativePin
port:
real v;
flow real i;
end;
connector Pin
port:
    real v;
    flow real i;
end;
```

One-Port Module

```
continuous Oneport
import PositivePin;
import NegativePin;
port:
PositivePin p;
NegativePin n;
value:
real v;
flow real i;
equation:
    v = p.v - n.v;
    0 = p.i + n.i;
    i = p.i;
end;
```

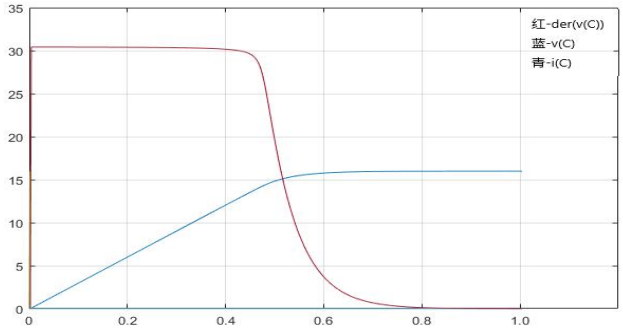
Signal Source Module

```
continuous SignalSource
parameter:
real offset=0;
real startTime=0;
port:
output real y;
end;
```

Voltage Source Module

```
continuous VoltageSource
import OnePort;
import SignalSource;
extends OnePort;
extends SignalSource;
equation:
y=v
end;
```

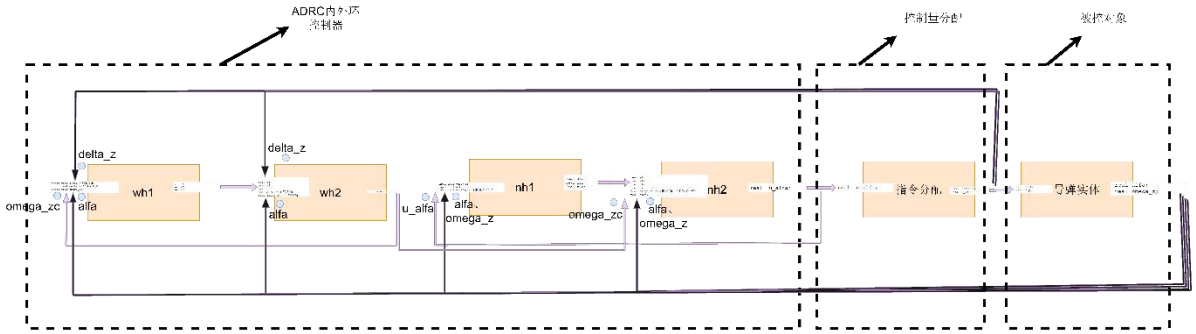
6.3.4 Simulation Results



6.4 Missile Attitude Control Model

6.4.1 Background Description

Modeling and simulating the missile attitude control model provided below. This case is a global iteration class case, meaning the data at different module times have an iterative transmission relationship. Mainly, *discrete* is used to model various modules.



ADRC Missile Attitude Control Model Schematic

6.4.2 System Model

```
couple controlsystem
    import daodanshiti;
    import inputstart;
    import neihuan1;
    import neihuan2;
    import waihuan1;
    import waihuan2;
    import zhilingfenpei;
part:
    daodanshiti daodan1;
    inputstart start1;
    neihuan1 nei1;
    neihuan2 nei2;
    waihuan1 wai1;
    waihuan2 wai2;
    zhilingfenpei zhiling1;
connection:
    connect(start1.alfa__c_out, wai1.alfa__c);
    connect(daodan1.alfa, wai1.alfa);
    connect(zhiling1.delta_z, wai1.delta_z);
    connect(wai2.omega_zc, wai1.omega_zc);
    connect(wai1.e3, wai2.e3);
    connect(wai1.z21, wai2.z21);
    connect(wai1.z22, wai2.z22);
    connect(daodan1.alfa, wai2.alfa);
    connect(zhiling1.delta_z, wai2.delta_z);
    connect(wai2.omega_zc, nei2.omega_zc);
    connect(daodan1.alfa, nei1.alfa);
    connect(daodan1.omega_z, nei1.omega_z);
    connect(nei2.u_alfa, nei1.u_alfa);
    connect(nei1.e1, nei2.e1);
    connect(nei1.z11, nei2.z11);
    connect(nei1.z12, nei2.z12);
    connect(daodan1.alfa, nei2.alfa);
    connect(daodan1.omega_z, nei2.omega_z);
    connect(daodan1.alfa, zhiling1.alfa);
    connect(nei2.u_alfa, zhiling1.u_alfa);
    connect(zhiling1.delta_z, daodan1.delta_z);
    connect(zhiling1.T_y, daodan1.T_y);
```

```
end;
```

6.4.3 Subsystem Model

Inner Loop 1

```
discrete neihuan1
    //import fal;
parameter:
    real time_step = 0.01;
    real beta21 = 100;
    real a2 = 0.95;
    real delta2 = 0.1;
    real bata11 = 100;
    real beta12 = 120;
    real a1 = 0.95;
    real delta1 = 0.1;
    real cs2 = -0.32;
value:
    real u__omega_z;
    real f__omega_z;
    real cs1;
    real der_z11;
    real der_z12;
port:
    //input
    event input real alfa;
    event input real u_alfa;
    event input real omega_z;
    //output
    event output real e1;
    event output real z11;
    event output real z12;
state:
    initial state work
        when entry() then
            statehold(infinite);
        end;
        when receive(u_alfa) then
            der_z11 = z12 - bata11 * e1 + f__omega_z + u_alfa;
            z11 = z11 + der_z11;
            e1 = z11 - omega_z;
```

```
                der_z12 = -beta12 * fal(e1, a1, delta1);
                z12 = z12 + der_z12;
                transition(send_message);
            end;
            when receive(alfa,omega_z) then
                transition(work);
            end;
        end;
        state send_message
            when entry() then
                statehold(0);
            end;
            when timeover() then
                transition(work);
            out
                send(e1,e1);
                send(z11,z11);
                send(z12,z12);
            end;
        end;
    end;
```

Inner Loop 2

```
    discrete neihuan2
        //import fal;
    parameter :
        real time_step = 0.01;
        real beta21 = 100;
        real a2 = 0.95;
        real delta2 = 0.1;
        real bata11 = 100;
        real beta12 = 120;
        real a1 = 0.95;
        real delta1 = 0.1;
        real cs2 = -0.32;
    value:
        real e2;
        real u__omega_z;
        real f__omega_z;
        real cs1;
        real der_omega_z;
```

```
port:
    //input
    event input real e1;
    event input real z11;
    event input real z12;
    event input real omega_zc;
    event input real omega_z;
    event input real alfa;
    //output
    event output real u_alfa;
state:
    initial state work
        when entry() then
            statehold(infinite);
        end;
        when receive(e1,z11,z12) then //neihuan1
            transition(work);
        end;
        when receive(omega_zc) then //forward
            e2 = omega_zc - z11;
            u__omega_z = beta21 * fal(e2, a2, delta2);
            der_omega_z = u__omega_z;
            omega_z    = omega_z+time_step*der_omega_z;
            cs1=3.2086*alfa + 11.025;
            f__omega_z = cs1 * alfa + cs2 * omega_z;//cs1 cs2
            u_alfa = u__omega_z - z12 - f__omega_z;//u_alfa
            transition(send_message);
        end;
        when receive(omega_z,alfa) then //Feedback
            transition(work);
        end;
    end;
    state send_message
        when entry() then
            statehold(0);
        end;
        when timeover() then
            transition(work);
        out
            send(u_alfa,u_alfa);
        end;
    end;
```

```
end;
```

Outer Loop 1

```
discrete waihuan1
    //import fal;
parameter:
    real time_step = 0.01;
    real bata31 = 80;
    real beta32 = 150;
    real beta41 = 60;
    real a3 = 0.95;
    real a4 = 0.95;
    real delta3 = 0.1;
    real delta4 = 0.1;
    real cs1 = 0.3;//c_x : to be determined
    real cs3 = -0.082;
value:
    real e4;
    real u__alfa;
    real f__alfa;
    real cs2;
    real der_alfa;
    real der_z21;
    real der_z22;
port:
    //input
    event input real alfa__c;
    event input real alfa;
    event input real delta_z;
    event input real omega_zc;
    //output
    event output real e3;
    event output real z21;
    event output real z22;
state:
    initial state idle
        when entry() then
            statehold(infinite);
        end;
        when receive(alfa__c) then//Calculate once, then proceed to complete what is
needed.
```

```
                e4 = alfa__c - alfa;//##e4 = alfa__c - z21; Calculate e4
                u__alfa = beta41 * fal(e4, a4, delta4);//Calculate u_alfa

                der_alfa = u__alfa;
                alfa = alfa+time_step*der_alfa;//update alfa
                cs2 = -3.2553 * alfa - 0.3828;
                f__alfa = cs1 * sin(alfa) + cs2 * alfa * cos(alfa) + cs3 * delta_z *
cos(alfa);//cs1 //cs2    cs3
                //State Observer
                der_z21 = z22 - bata31 * e3 + f__alfa + omega_zc;
                z21 = z21+time_step*der_z21;//update z21
                e3 = z21 - alfa;
                der_z22 = -beta32 * fal(e3, a3, delta3);
                z22 = z22+time_step*der_z22;//update z22
                transition(send_message);
            end;
        end;
        state work
            when entry() then
                statehold(infinite);
            end;
            when receive(alfa__c) then//Calculate once, then proceed to complete what is
needed.
                e4 = alfa__c - alfa;//##e4 = alfa__c - z21; Calculate e4
                u__alfa = beta41 * fal(e4, a4, delta4);//Calculate u_alfa
                der_alfa = u__alfa;
                alfa = alfa+time_step*der_alfa;//updatealfa
                cs2 = -3.2553 * alfa - 0.3828;
                f__alfa = cs1 * sin(alfa) + cs2 * alfa * cos(alfa) + cs3 * delta_z *
cos(alfa);//cs1    //cs2    cs3
                //State Observer
                der_z21 = z22 - bata31 * e3 + f__alfa + omega_zc;
                z21 = z21+time_step*der_z21;//update z21
                e3 = z21 - alfa;
                der_z22 = -beta32 * fal(e3, a3, delta3);
                z22 = z22+time_step*der_z22;//update z22
                transition(send_message);
            end;
            when receive(omega_zc) then
                transition(work);
            end;
            when receive(delta_z) then
```

```
                transition(work);
            end;
            when receive(alfa) then
                transition(work);
            end;
        end;
        state send_message
            when entry() then
                statehold(0);
            end;
            when timeover() then
                transition(work);
            out
                send(e3,e3);
                send(z21,z21);
                send(z22,z22);
            end;
        end;
    end;
```

Outer Loop 2

```
    discrete waihuan2
        //import fal;
    parameter:
        real time_step = 0.01;
        real alfa__c=0.4;
        real bata31 = 80;
        real beta32 = 150;
        real beta41 = 60;
        real a3 = 0.95;
        real a4 = 0.95;
        real delta3 = 0.1;
        real delta4 = 0.1;
        real cs1 = 0.3;//c_x: to be determined
        real cs3 = -0.082;
    value:
        real e4;
        real u__alfa;
        real f__alfa;
        real cs2;
        real der_alfa;//derivative
```

```
    port:
        //input
        event input real e3;
        event input real z21;
        event input real z22;
        event input real alfa;
        event input real delta_z;
        //output
        event output real omega_zc;
    state:
        initial state work
            when entry() then
                statehold(infinite);
            end;
            when receive(e3,z21,z22) then    //feedforward input
            //Error Feedback Controller
                e4 = alfa__c - alfa;//##e4 = alfa__c - z21;    alfa does not need to be updated
                u__alfa = beta41 * fal(e4, a4, delta4);
                der_alfa = u__alfa;
                alfa = alfa+time_step*der_alfa;//updatealfa
                //connection point
                //u__alfa    = omega_zc+ f__alfa+ z22;
                cs2 = -3.2553 * alfa - 0.3828;
                f__alfa = cs1 * sin(alfa) + cs2 * alfa * cos(alfa) + cs3 * delta_z *
cos(alfa);//cs1    cs2    cs3
                omega_zc = u__alfa-f__alfa-z22;
                transition(send_message);
            end;

            when receive(alfa) then    //feedback input
                transition(work);
            end;
            when receive(delta_z) then    //feedback input"
                transition(work);
            end;
        end;
        state send_message
            when entry() then
                statehold(0);
            end;
            when timeover() then
                transition(work);
```

```
            out
                send(omega_zc,omega_zc);
            end;
        end;
    end;
```

missile entity

```
    discrete daodanshiti
        import fal;
    parameter:
        real time_step = 0.01;
        real cs11=0.3;//c_x unknown, to be determined
        real cs13=-0.082;
        real cs14=-1/(255*1000);
        real cs23=-0.32;
        real cs24=1.26/306.3;////This parameter involves 'd,' cannot be determined.
    value:
        real delta_z;
        real T_y;
        real cs21;
        real cs22;
        real cs12;
        //derivative
        real der_alfa;
        real der_omega_z;
    port：
        //input
        event input real delta_z;
        event input real T_y;
        //output
        event output real alfa;
        event output real omega_z;
    state：
        initial state work
            when entry() then
                statehold(infinite);
            end;
            when receive(e3,z21,z22,alfa,delta_z) then
                cs21 = 3.2086 * alfa + 11.025;
                cs22 = 16.503 * abs(alfa) - 93.985;
                cs12 = -3.2553 * alfa - 0.3828;
```

```
                    //Error feedback controller
                    der_omega_z = cs21 * alfa + cs22 * delta_z + cs23 * omega_z + cs24 *
T_y;
                    omega_z = omega_z+time_step*der_omega_z;//update omega_z
                    der_alfa = omega_z + cs11 * sin(alfa) + cs12 * alfa * cos(alfa) +cs13 *
delta_z * cos(alfa) +cs14 * T_y * cos(alfa);
                    alfa = alfa+time_step*der_alfa;//update alfa
                    transition(send);
                end;
            end;
            state send
                when entry() then
                    statehold(0);
                end;
                when timeover() then
                    transition(work);
                out:
                    send(omega_zc,omega_zc);
                    send(alfa,alfa);
                end;
            end;
        end;
```

Command allocation

```
        discrete zhilingfenpei
            //import fal;
        parameter:
            real time_step = 0.01;
            real b2 = 1.26 / 306.3 ;//rad
        value:
            real b1;
        port:
            //input
            event input real alfa;
            event input real u_alfa;
            //output
            event output real delta_z;
            event output real T_y;
        state:
            initial state work
                when entry() then
                    statehold(infinite);
                end;
```

```
            when receive(u_alfa) then    //Feedforward
                //Error feedback controller
                b1 = 16.503 * abs(alfa) - 93.985;//2.24
                //b2* T_y = 0.5 * u_alfa;
                T_y = 0.5 * u_alfa/b2;
                //b1* delta_z = 0.5 * u_alfa;
                delta_z = 0.5 * u_alfa/b1;

                transition(send_message);
            end;
        end;
        state send_message
            when entry() then
                statehold(0);
            end;
            when timeover() then
                transition(work);
            out
                send(delta_z,delta_z);
                send(T_y,T_y);
            end;
        end;
    end;
```

Command input

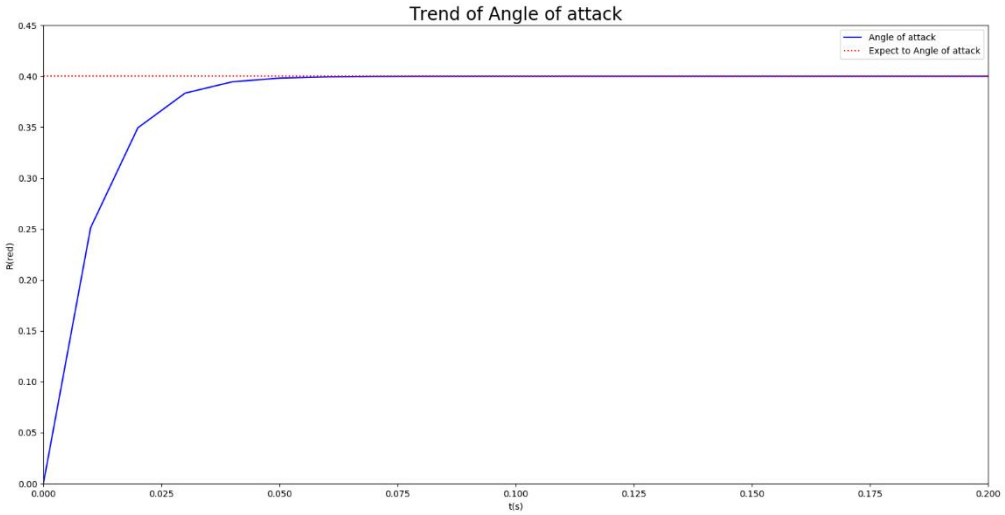
```
discrete inputstart
parameter:
    real alfa__c = 0.4;
port:
    event output real alfa__c_out;
state:
    initial state start
        when entry() then
            statehold(0);
        end;
        when timeover() then
            transition(work);
        out
            send(alfa__c_out,alfa__c);
        end;
    end;
```

```
    state work
        when entry() then
            statehold(0.01);
        end;
        when timeover() then
            transition(work);
        out
            send(alfa__c_out,alfa__c);
        end;
    end;
end;
```

function

```
function fal
port:
    input real x;
    input real a1;
    input real delta1;
    output real y;
action:
    if abs(x) > delta1 then
        y = (abs(x)) ^ a1 * sgn(x);
    else
        y = x /( delta1 ^ (1 - a1));
    end;
end;
```

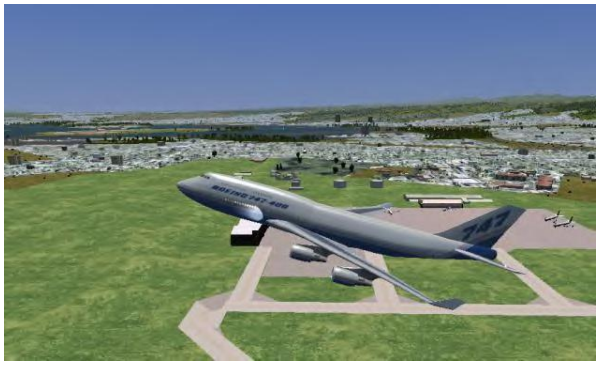
6.4.4 Simulation Results



6.5 Aircraft Takeoff Model

6.5.1 Background Description

Taking the widely used Boeing 747-400 as a typical commercial aircraft globally, this study focuses on the specific flight scenario of the aircraft, namely the takeoff scenario. Using the X language, the research involves requirements analysis, functional analysis, system design, and physical performance simulation verification for the takeoff scenario. This approach aims to address issues in traditional design methods, such as difficulty in tracing requirements and challenges in early verification of system functional architecture.



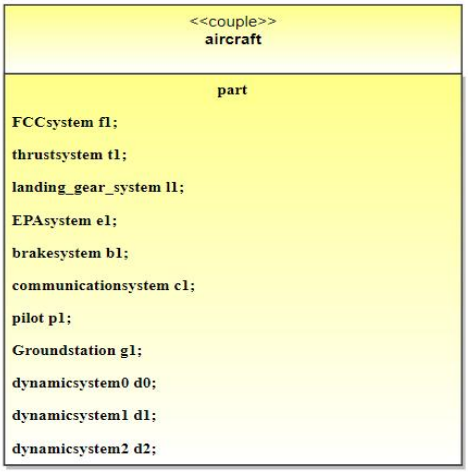
6.5.2 Modeling Analysis

Requirements and functions for the aircraft takeoff scenario are analyzed based on X-language requirement diagrams and use case diagrams. The system design is carried out using X-language definition diagrams and connection diagrams to build subsystem components that meet the requirements and functions of the takeoff scenario, along with their interaction logic. X-language definition diagrams and state machine diagrams are employed for the structure and behavior construction of subsystem components. Finally, simulation text is generated to simulate the functionality and performance, achieving the goal of validating the rationality of the design.

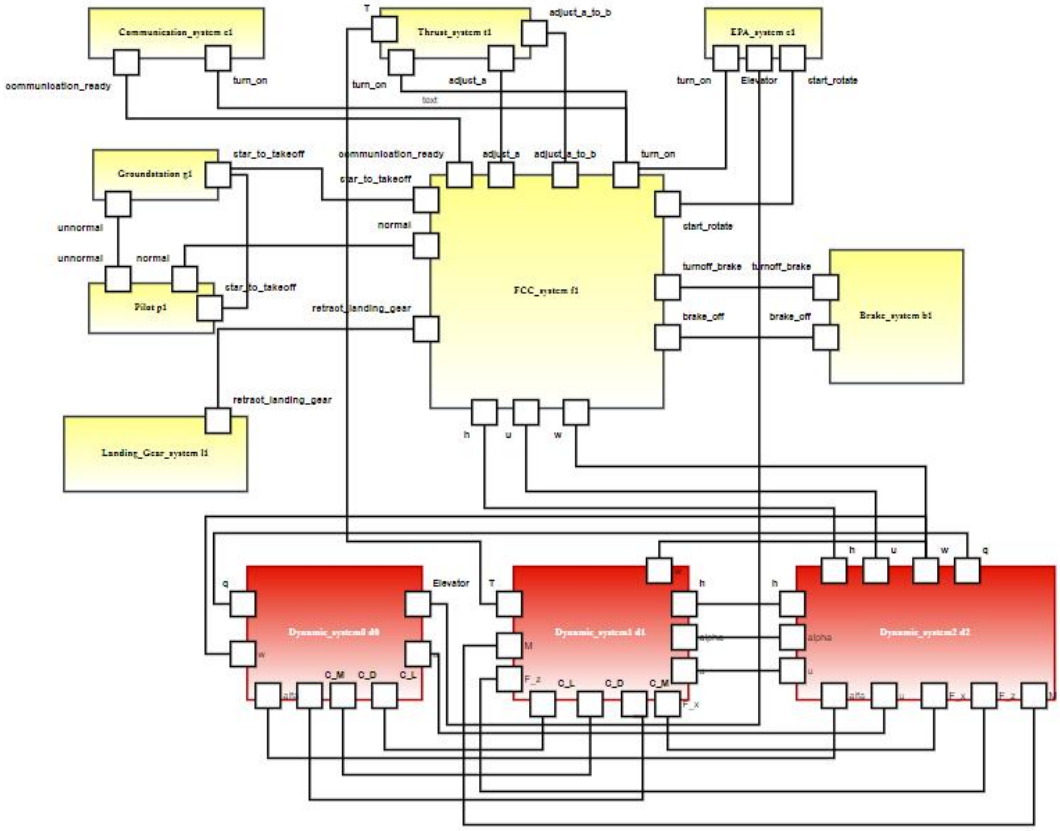
6.5.3 System Model

Based on the functional architecture analysis, the entire Boeing 747-400 takeoff process involves multiple aircraft subsystems. In X language, following a top-down modeling approach, the internal composition of each subsystem and their interaction relationships within the entire flight system are established using coupling classes. X-language coupling classes are described at the graphical modeling level using definition and connection diagrams (as shown below). The definition diagram describes the subsystems involved in the takeoff process, while the connection diagram illustrates the interaction relationships between subsystems. Additionally, after establishing the graphical model of the flight system, the X-language textual model can be automatically generated (as shown in the diagram).

Definition Diagram:



Connection Diagram:



couple aircraft

import FCCsystem;
import thrustsystem;
import landing_gear_system;
import EPAsystem;
import brakesystem;
import communicationsystem;
import pilot;
import Groundstation;

```
import dynamicsystem0;
import dynamicsystem1;
import dynamicsystem2;
part:
    FCCsystem f1;
    thrustsystem t1;
    landing_gear_system l1;
    EPAsystem e1;
    brakesystem b1;
    communicationsystem c1;
    pilot p1;
    Groundstation g1;
    dynamicsystem0 d0;
    dynamicsystem1 d1;
    dynamicsystem2 d2;
connection:
    connect(l1.retract_landing_gear,f1.retract_landing_gear);
    connect(p1.normal,f1.normal);
    connect(p1.unnormal,g1.unnormal);
    connect(p1.star_to_takeoff,g1.star_to_takeoff);
    connect(g1.star_to_takeoff,f1.star_to_takeoff);
    connect(f1.turn_on,c1.turn_on);
    connect(f1.brake_off,b1.brake_off);
    connect(c1.communication_ready,f1.communication_ready);
    connect(t1.turn_on,f1.turn_on);
    connect(t1.adjust_a_to_b,f1.adjust_a_to_b);
    connect(t1.adjust_a,f1.adjust_a);
    connect(e1.turn_on,f1.turn_on);
    connect(e1.start_rotate,f1.start_rotate);
    connect(b1.turnoff_brake,f1.turnoff_brake);
    connect(d0.Elevator,e1.Elevator);
    connect(d0.C_L,d1.C_L);
    connect(d0.C_D,d1.C_D);
    connect(d0.C_M,d1.C_M);
    connect(d1.T,t1.T);
    connect(d1.F_x,d2.F_x);
    connect(d1.F_z,d2.F_z);
    connect(d1.M,d2.M);
    connect(d2.h,d1.h);
    connect(d2.h,f1.h);
    connect(d2.u,f1.u);
    connect(d2.w,f1.w);
```

```
        connect(d2.w,d1.w);
        connect(d2.w,d0.w);
        connect(d2.alfa,d0.alfa);
        connect(d2.u,d0.u);
        connect(d2.alpha,d1.alpha);
        connect(d2.u,d1.u);
        connect(d2.q,d0.q);
end;
```

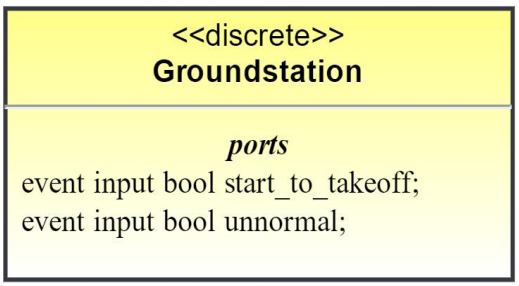
6.5.4 Subsystem Model

From the established flight system model, we have defined the signal or data interaction logic for each subsystem during the flight process. Building upon the analysis above, we use X language to establish the structure and behavioral logic of each subsystem.

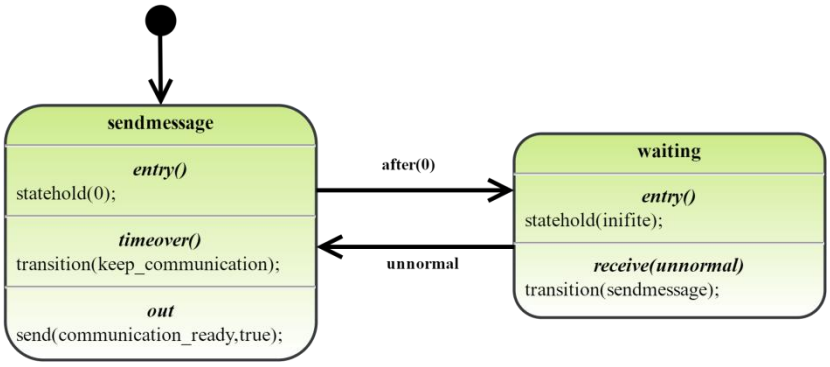
6.5.4.1 Base Station

Throughout the entire flight process, the base station communicates with the pilot and sends takeoff commands. Here, we use X-language discrete classes to implement the structure and behavioral description of the base station. X-language discrete classes are described at the graphical modeling level using definition and state machine diagrams (as shown below). The definition diagram describes the input and output signals of the base station, while the state machine diagram illustrates the states and behavioral logic of the base station. Additionally, after establishing the graphical model of the base station, the X-language textual model can be automatically generated (as shown in the diagram).

Definition Diagram:



State Machine Diagram:



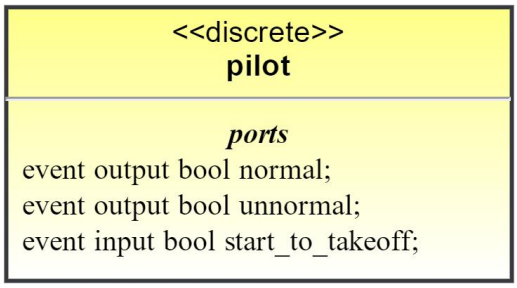

```
discrete Groundstation
    port:
        event input bool unnormal;
        event output bool start_to_takeoff;
    state:
        initial state sendmessage
            when entry() then
                statehold(0);
            end;
            when timeover() then
                transition(waiting);
            out
                send(start_to_takeoff,true);
            end;
        end;
        state waiting
            when entry() then
                statehold(inifite);
            end;
            when receive(unnormal) then
                transition(sendmessage);
            end;
        end;
    end;
end;
```

6.5.4.2 Pilot

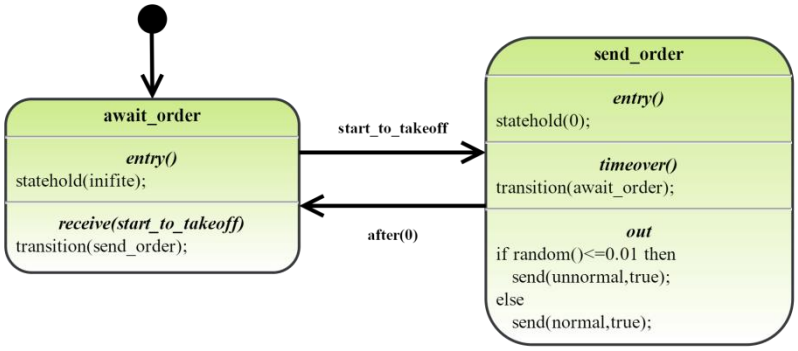
The pilot plays a crucial role throughout the entire flight process, responsible for controlling the aircraft at various stages. Using X-language discrete classes, we implement the structure and behavioral description of the pilot. X-language discrete classes are described at

the graphical modeling level using definition and state machine diagrams (as shown below). The definition diagram describes the input and output signals of the pilot, while the state machine diagram illustrates the states and behavioral logic of the pilot. Additionally, after establishing the graphical model of the pilot, the X-language textual model can be automatically generated (as shown in the diagram).

Definition Diagram:



State Machine Diagram:




```
discrete pilot
    port:
        event input bool start_to_takeoff;
        event output bool normal;
        event output bool unnormal;
    state:
        initial state await_order
            when entry() then
                statehold(inifite);
            end;
            when receive(start_to_takeoff) then
                transition(send_order);
```

```
            end;
        end;
        state send_order
            when entry() then
                statehold(0);
            end;
            when timeover() then
                transition(await_order);
            out
                if random()<=0.01 then
                    send(unnormal,true);
                else
                    send(normal,true);
                end;
            end;
        end;
    end;
```

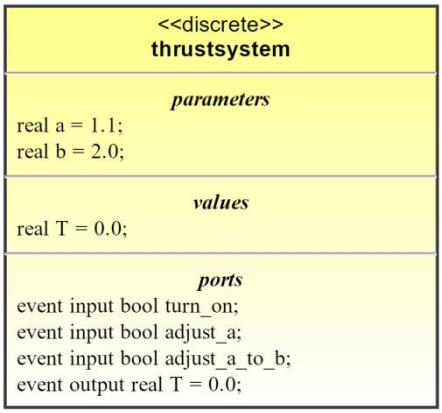
6.5.4.3 Flight Control Computer System

Throughout the entire process, serving as the main control system, the Flight Control Computer System coordinates and controls the logical behavior of other subsystems during the takeoff process. After receiving the takeoff command, the flight control system enters a pre-detection state. Upon confirmation of correctness, it begins the takeoff preparation. After receiving a normal command from the communication system, it starts sending commands to release the brakes to the braking system. Upon receiving the brake release command, it sends thrust command 1 to the thrust system and, 3 seconds later, sends thrust command 2 to the thrust system. When the speed exceeds the takeoff decision speed (VR), it sends elevator adjustment commands to the power system. When the flight altitude exceeds 12 meters, it sends a command to retract the landing gear to the landing gear system, completing the takeoff process.

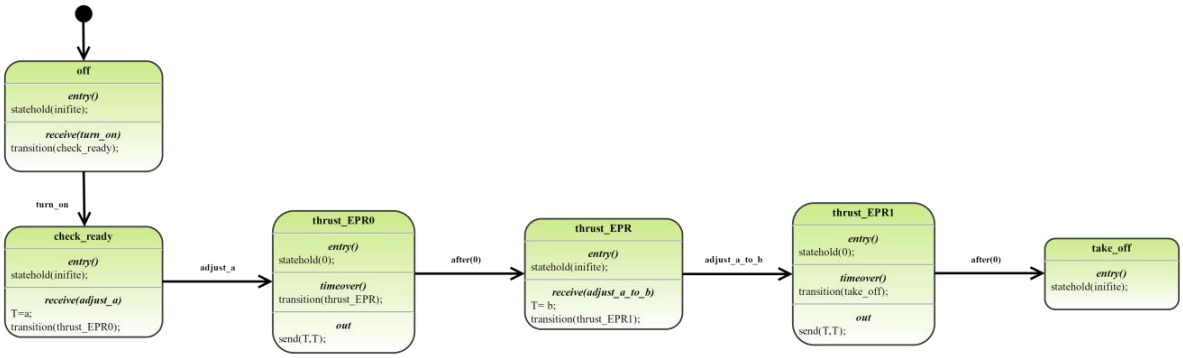
Here, we use X-language discrete classes to implement the structure and behavioral description of the flight control system. X-language discrete classes are described at the graphical modeling level using definition and state machine diagrams (as shown below). The definition diagram describes the input and output signals of the flight control system, while

the state machine diagram illustrates the states and behavioral logic of the flight control system. Additionally, after establishing the graphical model of the flight control system, the X-language textual model can be automatically generated (as shown in the diagram)

Definition Diagram:



State Machine Diagram:



discrete FCCsystem

value:

real speedFCC;

real altitudeFCC;

port:

event input bool start_to_takeoff;

event input bool normal;

event input bool unnormal;

event input real u;

event input real w;

event input real h;

event input bool communication_ready;

event input bool brake_off;

event output bool adjust_a;

```
        event output bool adjust_a_to_b;
        event output bool turnoff_brake;
        event output bool start_rotate;
        event output bool retract_landing_gear;
        event output bool turn_on;
    state:
        initial state stop
            when entry() then
                statehold(inifite);
            end;
            when receive(start_to_takeoff) then
                transition(perflight_check);
            end;
        end;
        state preflight
            when entry() then
                statehold(inifite);
            end;
            when receive(normal) then
                transition(send_start0);
            end;
            when receive(unnormal) then
                transition(stop);
            end;
        end;
        state send_start0
            when entry() then
                statehold(0);
            end;
            when timeover() then
                transition(send_start);
            out
                send(turn_on,true);
            end;
        end;
        state send_start
            when entry() then
                statehold(inifite);
            end;
            when receive(communication_ready) then
                transition(release_brake0);
            end;
```

```
end;
state release_brake0
    when entry() then
        statehold(0);
    end;
    when timeover() then
        transition(release_brake);
    out
        send(turnoff_brake,true);
    end;
end;
state release_brake
    when entry() then
        statehold(inifite);
    end;
    when receive(brake_off) then
        transition(thrust_EPR0);
    end;
end;
state thrust_EPR0
    when entry() then
        statehold(0);
    end;
    when timeover() then
        transition(thrust_EPR);
    out
        send(adjust_a,true);
    end;
end;
state thrust_EPR
    when entry() then
        statehold(3.0);
    end;
    when timeover() then
        transition(judge_v1);
    out
        send(adjust_a_to_b,true);
    end;
end;
state judge_v1
    when entry() then
        statehold(inifite);
```

```
                end;
                when receive(u,w) then
                    speedFCC=sqrt(u^2+w^2);
                if speedFCC >= 63.28 then
                    transition(judge_VR);
                else
                    transition(judge_v1);
                end;
            end;
        end;
        state judge_VR
            when entry() then
                statehold(inifite);
            end;
            when receive(u,w) then
                speedFCC= sqrt(u^2+w^2);
            if speedFCC >= 70.48 then
                transition(judge_altitude0);
            else
                transition(judge_VR);
            end;
        end;
    end;
    state judge_altitude0
        when entry() then
            statehold(0);
        end;
        when timeover() then
            transition(judge_altitude);
        out
            send(start_rotate,true);
        end;
    end;
    state judge_altitude
        when entry() then
            statehold(inifite);
        end;
        when receive(h) then
            altitudeFCC = h;
        if altitudeFCC >=23 then
            transition(relanding_gear);
        else
```

```
                transition(judge_altitude);
        end;
end;
end;
state relanding_gear
when entry() then
        statehold(0);
end;
when timeover() then
        transition(keep_climb);
out
        send(retract_landing_gear,true);
end;
end;
state keep_climb
when entry() then
        statehold(inifite);
end;
end;
end;
end;
```

6.5.4.4 Thrust System

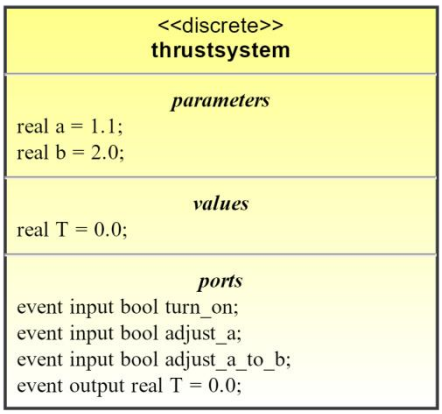
Throughout the entire process, the thrust system, upon receiving the start command, enters the check preparation state. When receiving thrust system command 1, it adjusts the thrust control signal to 1.1. Upon receiving thrust system command 2, it adjusts the thrust control signal to 2.

Here, we use X-language discrete classes to implement the structure and behavioral description of the thrust system. X-language discrete classes are described at the graphical modeling level using definition and state machine diagrams (as shown below). The definition diagram describes the input and output signals of the thrust system, while the state machine diagram illustrates the states and behavioral logic of the thrust system. Additionally, after establishing the graphical model of the thrust system, the X-language textual model can be

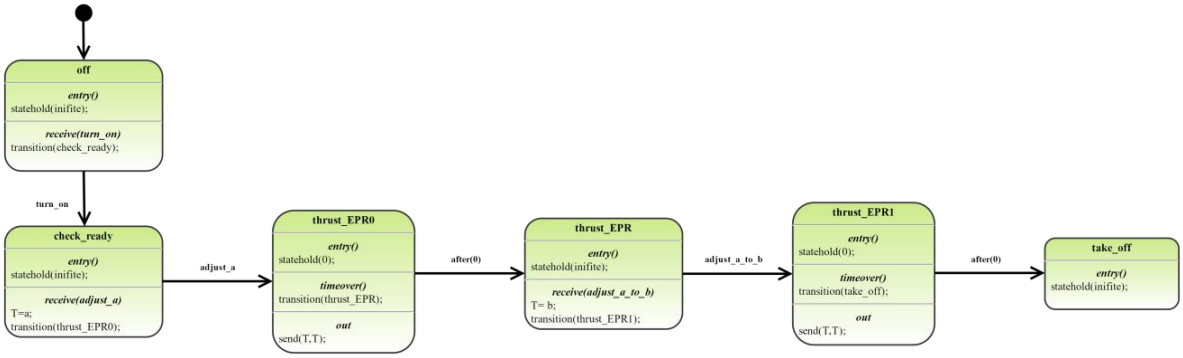
automatically generated (as shown in the diagram).

Graphical Modeling:

Definition Diagram:



State Machine Diagram:



Textual Modeling

discrete thrustsystem

port:

event input bool turn_on;

event input bool adjust_a;

event input bool adjust_a_to_b;

event output real T = 0.0;

parameter:

real a = 1.1;

real b = 2.0;

real c = 909000;

value:

real t0;

real t;

state:

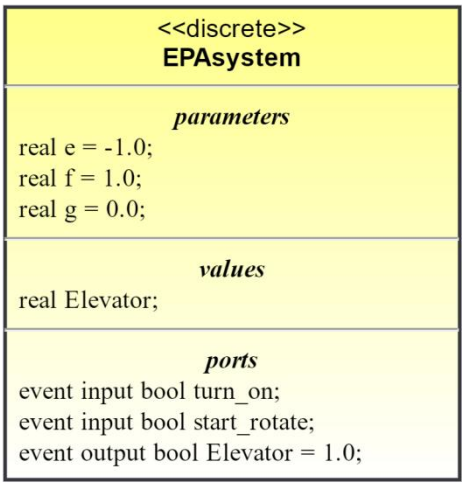
```
initial state off
        when entry() then
                statehold(inifite);
        end;
        when receive(turn_on) then
                transition(check_ready);
        end;
end;
state check_ready
        when entry() then
                statehold(inifite);
        end;
        when receive(adjust_a) then
                t0=a;
                t = t0*c;
                transition(thrust_EPR);
        end;
end;
state thrust_EPR
        when entry() then
                statehold(0.01);
        end;
        when timeover() then
                transition(thrust_EPR);
        out
                send(T,t);
        end;
        when receive(adjust_a_to_b) then
                t0=a;
                t = t0*c;
                transition(thrust_EPR1);
        end;
end;
state thrust_EPR1
        when entry() then
                statehold(0);
        end;
        when timeover() then
                transition(take_off);
        out
                send(T,t);
        end;
```

```
            end;
        end;
    end;
```

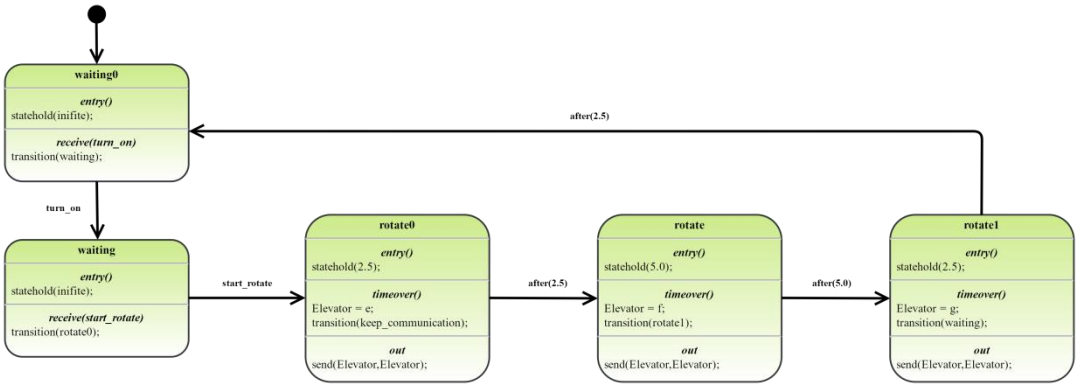
6.5.4.5 Power System

Graphical Modeling:

Definition Diagram:



State Machine Diagram:



Textual Modeling:

```
discrete EPAsystem
    parameter:
        real e = -1.0;
        real f = 1.0;
        real g =0.0;
        int cont = 0;
    value:
```

```
        real E;
    port:
        event input bool turn_on;
        event input bool start_rotate;
        event output bool Elevator;
    state:
        initial state waiting0
            when entry() then
                statehold(inifite);
            end;
            when receive(turn_on) then
                transition(waiting);
            end;
        end;
        state waiting
            when entry() then
                statehold(inifite);
            end;
            when receive(start_rotate) then
                transition(rotate0);
            end;
        end;
        state rotate0
            when entry() then
                statehold(0.01);
            end;
            when timeover() then
                E = f;
                cont = cont +1;
            if cont<= 250 then
                transition(rotate0);
            else
                transition(rotate);
            end;
            out
                send(Elevator,E);
            end;
        end;
        state rotate
            when entry() then
                statehold(5.0);
            end;
```

```
                when timeover() then
                    E= f ;
                    cont=cont+1;
                if cont <= 500 then
                    transition(rotate);
                else
                    transition(rotate1);
                end;
                out
                    send(Elevator,E);
                end;
            end;
            state rotate1
                when entry() then
                    statehold(2.5);
                end;
                when timeover() then
                    E = g ;
                       transition(waiting);
                if cont <= 250 then
                    transition(rotate1);
                else
                    transition(rotate2);
                end;
                out
                    send(Elevator,E);
                end;
            end;
            state rotate2
                when entry() then
                    statehold(0);
                end;
                when timeover() then
                    E= g ;
                    transition(waiting);
                out
                    send(Elevator,E);
                end;
            end;
        end;
    end;
```

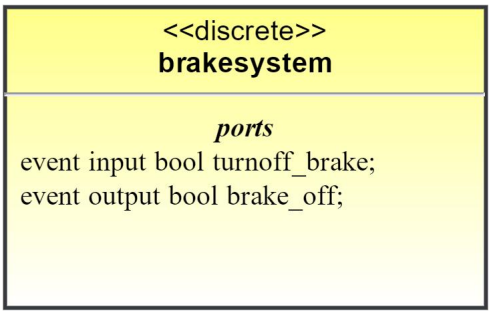
6. 5. 4. 6 Landing Gear System

Throughout the entire process, the landing gear system, upon receiving the command to retract the landing gear, adjusts the landing gear control signal to initiate the retraction state.

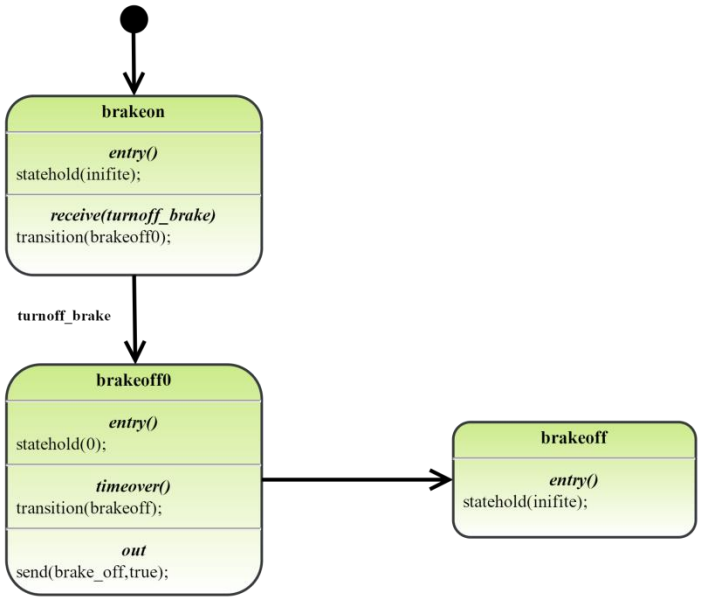
Here, we use X-language discrete classes to implement the structure and behavioral description of the landing gear system. X-language discrete classes are described at the graphical modeling level using definition and state machine diagrams (as shown below). The definition diagram describes the input and output signals of the landing gear system, while the state machine diagram illustrates the states and behavioral logic of the landing gear system. Additionally, after establishing the graphical model of the landing gear system, the X-language textual model can be automatically generated (as shown in the diagram).

Graphical Modeling:

Definition Diagram:



State Machine Diagram:



Textual Modeling:

```
discrete landing_gear_system
    port:
        event input bool retract_landing_gear;
        event output int Lg;
    value:
        int lg;
    state:
        initial state landing_gear_down
            when entry() then
                statehold(inifite);
            end;
            when receive(retract_landing_gear) then
                transition(landing_gear_up0);
            end;
        end;
        state landing_gear_up0
            when entry() then
                statehold(0);
            end;
            when timeover() then
                lg = 0;
                transition(landing_gear_up);
            out
                send(Lg,lg);
            end;
        end;
```

```
            state landing_gear_up
                when entry() then
                    statehold(inifite);
                end;
            end;
        end;
    end;
```

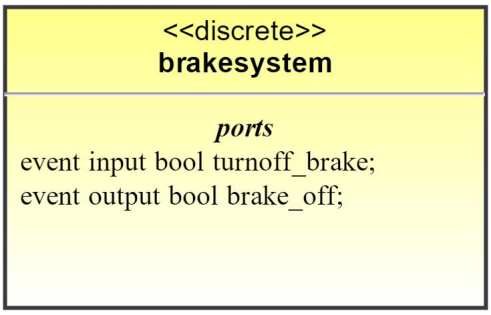
6.5.4.7 Braking System

Throughout the entire process, the braking system, upon receiving the brake release command, releases the brakes and sends a brake released command to adjust to the brake release state.

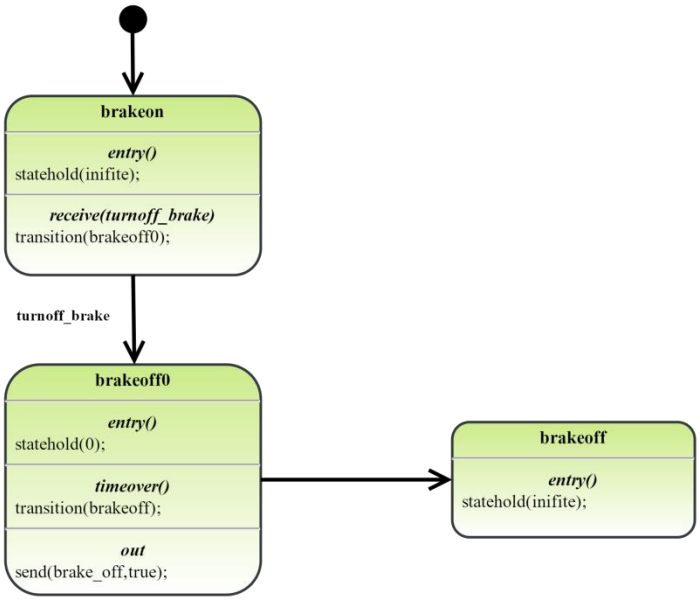
Here, we use X-language discrete classes to implement the structure and behavioral description of the braking system. X-language discrete classes are described at the graphical modeling level using definition and state machine diagrams (as shown below). The definition diagram describes the input and output signals of the braking system, while the state machine diagram illustrates the states and behavioral logic of the braking system. Additionally, after establishing the graphical model of the braking system, the X-language textual model can be automatically generated (as shown in the diagram).

Graphical Modeling:

Definition Diagram:



State Machine Diagram:



Textual Modeling:

```
discrete brakesystem
    port:
        event input bool turniff_brake;
        event output bool brake_off;
    state:
        initial state brakeon
            when entry() then
                statehold(inifite);
            end;
            when receive(turnoff_brake) then
                transition(brakeoff0);
            end;
        end;
        state brakeoff0
            when entry() then
                statehold(0);
            end;
            when timeover() then
                transition(brakeoff);
            out
                send(brake_off,true);
            end;
        end;
        state brakeoff
            when entry() then
```

```
                    statehold(inifite);
                end;
            end;
        end;
    end;
```

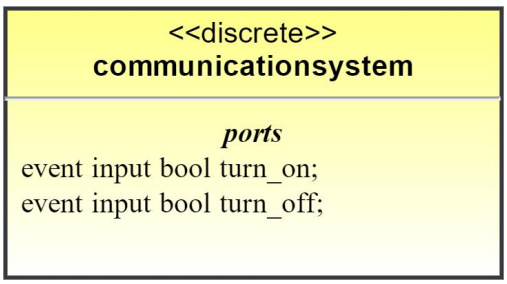
6. 5. 4. 8 Communication System

Throughout the entire process, the communication system, upon receiving the start command, sends a communication normal command 3 seconds later and adjusts to the communication holding state.

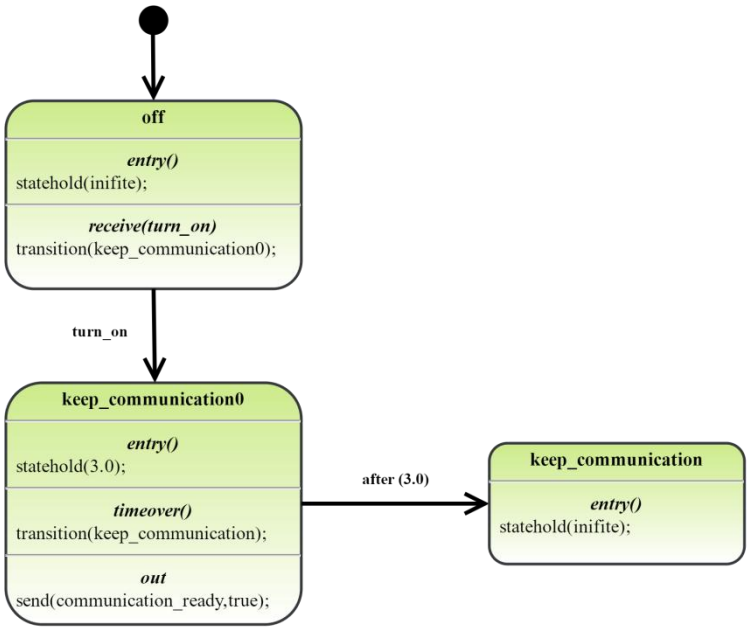
Here, we use X-language discrete classes to implement the structure and behavioral description of the communication system. X-language discrete classes are described at the graphical modeling level using definition and state machine diagrams (as shown below). The definition diagram describes the input and output signals of the communication system, while the state machine diagram illustrates the states and behavioral logic of the communication system. Additionally, after establishing the graphical model of the communication system, the X-language textual model can be automatically generated (as shown in the diagram).

Graphical Modeling:

Definition Diagram:



State Machine Diagram:



extual Modeling:

```
discrete communicationsystem
    port:
        event input bool turn_on;
        event output bool communication_ready;
    state:
        initial state off
            when entry() then
                statehold(inifite);
            end;
            when receive(turn_on) then
                transition(keep_communication0);
            end;
        end;
        state keep_communication0
            when entry() then
                statehold(3);
            end;
            when timeover() then
                transition(keep_communication);
            out
                send(communication_ready,true);
            end;
        end;
        state keep_communication
```

```
                when entry() then
                    statehold(inifite);
                end;
            end;
        end;
    end;
```

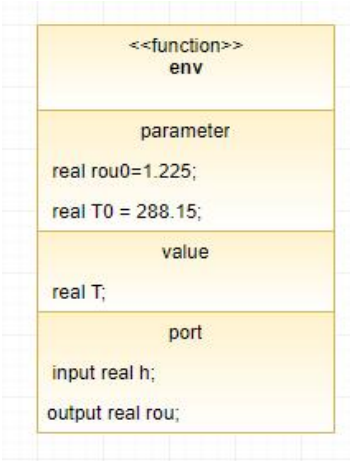
6.5.4.9 Flight Environment System

The establishment of the Flight Environment System is primarily to make the simulation scenario more realistic. Due to the variation in flight altitude, atmospheric density constantly changes. However, atmospheric density affects dynamic pressure (changes in dynamic pressure lead to changes in aircraft aerodynamics and aerodynamic moments). In summary, the input to the Flight Environment System is the real-time flight altitude in the flight simulation, and the output is atmospheric density (ρ).

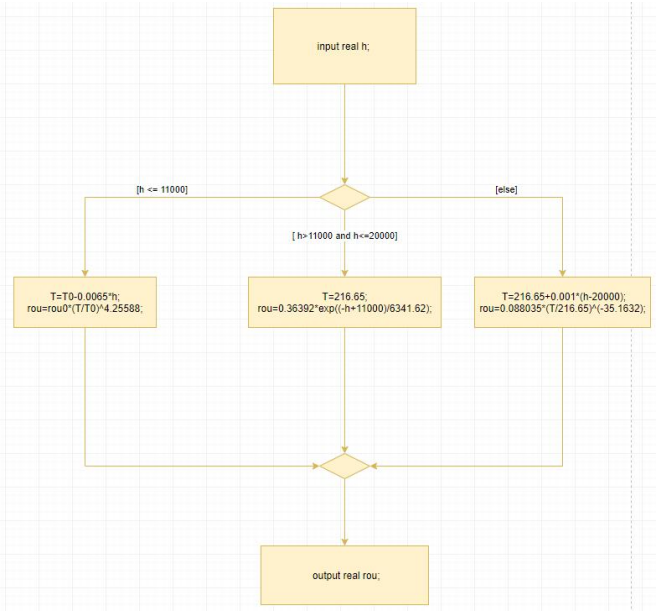
Here, we use X-language function classes to implement the structure and behavioral description of the Flight Environment System. X-language function classes are described at the graphical modeling level using definition and activity diagrams (as shown below). The definition diagram describes the input and output of the Flight Environment System, while the activity diagram illustrates the behavioral logic of the Flight Environment System. Additionally, after establishing the graphical model of the Flight Environment System, the X-language textual model can be automatically generated (as shown in the diagram).

Graphical Modeling:

Definition Diagram:



Activity Diagram:



Textual Modeling:

```
function env
  parameter:
     real rou0=1.225;
     real T0 = 288.15;
  value:
     real T;
  port:
     input real h;
     output real rou;
```

```
    action:
        if h <= 11000 then
            T=T0-0.0065*h;
            rou=rou0*(T/T0)^4.25588;
        elseif h>11000 and h<=20000 then
            T=216.65;
            rou=0.36392*exp((-h+11000)/6341.62);
        else
            T=216.65+0.001*(h-20000);
            rou=0.088035*(T/216.65)^(-35.1632);
        end if;
    end;
```

6.5.4.10 Dynamics System

The aircraft takeoff process includes three stages: taxiing, lifting the nose wheel, and climbing. The entire process is mainly influenced by the aerodynamic forces acting on the aircraft, including lift (T) and drag (D), the support force from the ground (F_N), friction force (F_f), gravity (G), and thrust (T).

During the taxiing phase, the aircraft is in contact with the ground, and there is a support force (F_N) and friction force (F_f). When the aircraft's nose wheel leaves the ground, the support force (F_N) and friction force (F_f) disappear instantly. Through force analysis of the takeoff process, in the body coordinate system, the combined forces of the support force (F_N) in the X-axis direction and Z-axis direction are given by the following equation.

$$\begin{cases} F_N = -[G + D\sin(\theta - \alpha) - T\sin(\varphi_T + \theta) - L\cos(\theta - \alpha)] \\ F_x = T\cos\varphi_T - F_N\sin\theta + F_f\cos\theta - G\sin\theta - L\sin\alpha + D\cos\alpha \\ F_Z = -T\sin\varphi_T + F_N\cos\theta + F_f\sin\theta + G\cos\theta + L\cos\alpha + D\sin\alpha \end{cases}$$

Where φ_T is the engine installation angle, θ is the aircraft pitch angle, and α is the aircraft angle of attack.

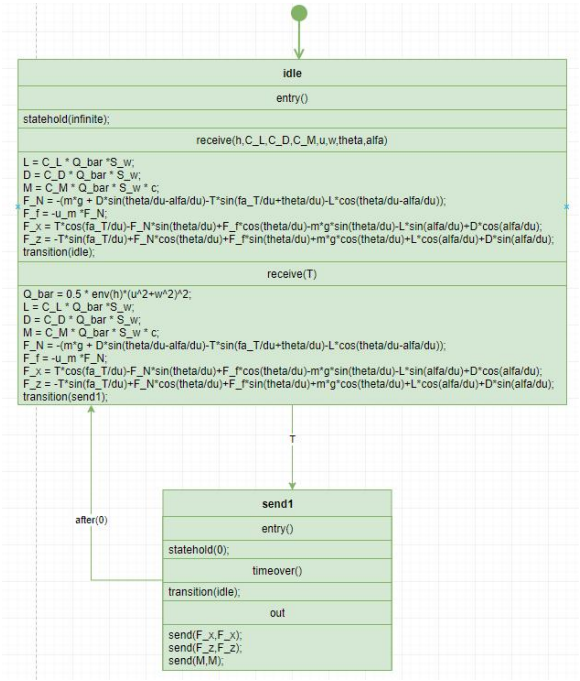
Here, we use X-language discrete classes to implement the structure and behavioral description of the flight dynamics system. X-language discrete classes are described at the

graphical modeling level using definition and state machine diagrams (as shown below). The definition diagram describes the input and output signals of the flight dynamics system, while the state machine diagram illustrates the states and behavioral logic of the flight dynamics system. Additionally, after establishing the graphical model of the flight dynamics system, the X-language textual model can be automatically generated (as shown in the diagram).

Graphical Modeling:

Definition Diagram:

State Machine Diagram:



Textual Modeling:

```
discrete dynamicsystem1
    parameter:
        real m = 288772;
        real g = 9.8;
        real fa_T = 2.5;
        real u_m = 0.04;
        real c = 8.32;
        real S_w = 510.97;
        real du = 0.01745;
    value:
        real F_N;
        real F_f;
        real Q_bar;
        real L;
        real D;
    port:
        event input real h;
        event input real theta;
        event input real alfa;
        event input real u;
        event input real w;
        event output real C_L;
        event output real C_D;
        event output real C_M;
        event output real T;
        event output real F_x;
        event output real F_z;
```

```
        event output real M;
    state:
        initial state idle
            when entry() then
                statehold(infinite);
            end;
            when receive(h,C_L,C_D,C_M,u,w,theta,alfa) then
                L = C_L * Q_bar *S_w;
                D = C_D * Q_bar * S_w;
                M = C_M * Q_bar * S_w * c;
                F_N = -(m*g + D*sin(theta/du-alfa/du)-T*sin(fa_T/du+theta/du)-L*cos(theta/du-alfa/du));
                F_f = -u_m *F_N;
                F_x = T*cos(fa_T/du)-F_N*sin(theta/du)+F_f*cos(theta/du)-m*g*sin(theta/du)-L*sin(alfa/du)+D*cos(alfa/du);
                 F_z = -T*sin(fa_T/du)+F_N*cos(theta/du)+F_f*sin(theta/du)+m*g*cos(theta/du)+L*cos(alfa/du)+D*sin(alfa/du);
                transition(idle);
            end;
            when receive(T) then
                Q_bar = 0.5 * env(h)*(u^2+w^2);
                L = C_L * Q_bar *S_w;
                D = C_D * Q_bar * S_w;
                M = C_M * Q_bar * S_w * c;
                F_N = -(m*g + D*sin(theta/du-alfa/du)-T*sin(fa_T/du+theta/du)-L*cos(theta/du-alfa/du));
                F_f = u_m *F_N;
                F_x = T*cos(fa_T/du)-F_N*sin(theta/du)+F_f*cos(theta/du)-m*g*sin(theta/du)
```

```
-L*sin(alfa/du)+D*cos(alfa/du);
                F_z = -T*sin(fa_T/du)+F_N*cos(theta/du)+F_f*sin(theta/du)+m*g*cos(theta/
du)+L*cos(alfa/du)+D*sin(alfa/du);
                transition(send1);
            end;
        end;
        state send1
            when entry() then
                statehold(0);
            end;
            when timeover() then
                transition(idle);
            out
                send(F_x,F_x);
                send(F_z,F_z);
                send(M,M);
            end;
        end;
    end;
```

6.5.4.11 Kinematic System

As the longitudinal motion variation is crucial in the aircraft takeoff scenario, this document focuses on the analysis of the aircraft's longitudinal motion. The kinematic equation for the aircraft takeoff process is given by the following expression:

$$\begin{cases} \frac{F_x}{m} = \dot{u} + wq \\ \frac{F_z}{m} = \dot{w} + uq \\ M = \dot{q}I_y \\ \dot{\theta} = q \\ \dot{X} = u\cos\theta + w\sin\theta \\ \dot{Y} = -u\sin\theta + w\cos\theta \end{cases}$$

Where u and w are the velocities in the X-axis and Z-axis directions, $\dot{u}$ and $\dot{w}$ are the accelerations in the X-axis and Z-axis directions, m is the weight of the aircraft, θ and q are the pitch angle and pitch angular rate, and X and Y are the distances traveled in the

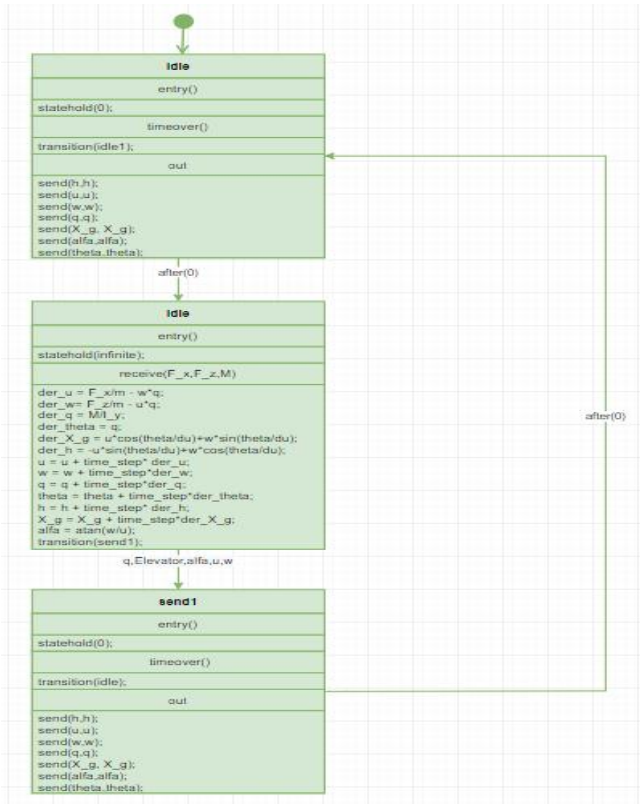
X-axis and Z-axis directions, respectively.

Here, based on X-language discrete classes, we implement the structure and behavioral description of the aircraft's kinematic system. X-language discrete classes are depicted at the graphical modeling level using definition and state machine diagrams (as shown below). The definition diagram describes the input and output signals of the aircraft kinematic system, while the state machine diagram illustrates the states and behavioral logic of the aircraft kinematic system. Additionally, after establishing the graphical model of the aircraft kinematic system, the X-language textual model can be automatically generated (as shown in the diagram).

Graphical Modeling:

Definition Diagram:

State Machine Diagram:



Textual Modeling:

```
discrete dynamicsystem2
parameter:
    real I_y = 44.86*10^6;
    real time_step = 0.01;
    real m = 288772;
    real du = 0.01745;
value:
    real der_h;
    real der_u;
    real der_w;
    real der_q;
    real der_X_g;
    real der_theta;
port:
    event output real h;
    event output real theta;
    event output real alfa;
    event output real u;
    event output real w;
    event output real q;
    event output real X_g;
    event input real F_x;
```

```
    event input real F_z;
    event input real M;
  state:
    initial state idle
      when entry() then
        statehold(0);
      end;
      when timeover() then
        transition(idle1);
      out
        send(h,h);
        send(u,u);
        send(w,w);
        send(q,q);
        send(X_g, X_g);
        send(alfa,alfa);
        send(theta,theta);
      end;
    end;
    state idle
      when entry() then
        statehold(infinite);
      end;
      when receive(F_x,F_z,M) then
        der_u = F_x/m - w*q;
        der_w= F_z/m - u*q;
        der_q = M/I_y;
        der_theta = q;
        der_X_g = u*cos(theta/du)+w*sin(theta/du);
        der_h = -u*sin(theta/du)+w*cos(theta/du);
        u = u + time_step* der_u;
        w = w + time_step*der_w;
        q = q + time_step*der_q;
        theta = theta + time_step*der_theta;
        h = h + time_step* der_h;
        X_g = X_g + time_step*der_X_g;
        alfa = atan(w/u);
        transition(send1);
      end;
    end;
    state send1
      when entry() then
```

```
            statehold(0);
        end;
        when timeover() then
            transition(idle);
        out
            send(h,h);
            send(u,u);
            send(w,w);
            send(q,q);
            send(X_g, X_g);
            send(alfa,alfa);
            send(theta,theta);
        end;
    end;
end;
```

6.5.5 Simulation Results

Figures 4.1 to 4.4 depict the variations in various performance parameters (flight altitude, horizontal flight distance, pitch angle, and speed, etc.) of the aircraft during takeoff. Combining Figures 4.1 to 4.3, we observe that the aircraft initiates the rotation for liftoff when the speed reaches approximately 80 m/s. As the aircraft ascends to a height of 25 meters above the ground, the flight speed approaches 85 m/s. Furthermore, Figure 4.4 indicates that during the takeoff process, the pitch angle and angle of attack of the aircraft remain consistent in the early stages, reaching about 6° before gradually diverging. Subsequently, the pitch angle continues to increase, stabilizing at around 15°, while the angle of attack gradually decreases to 0. This suggests that during takeoff, as the pitch angle reaches approximately 6°, the aircraft is lifting off the ground and ultimately climbing with a pitch angle of around 15°.

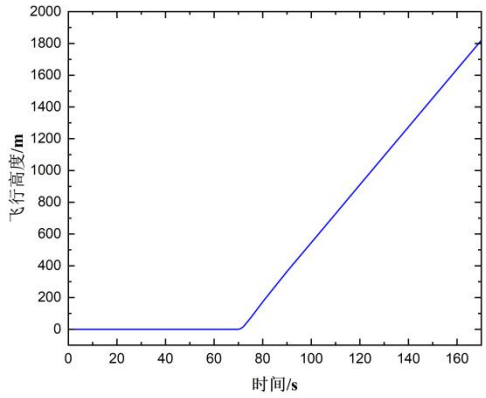


Figure 4.1 Simulation Results: Variation of Flight Altitude with Time

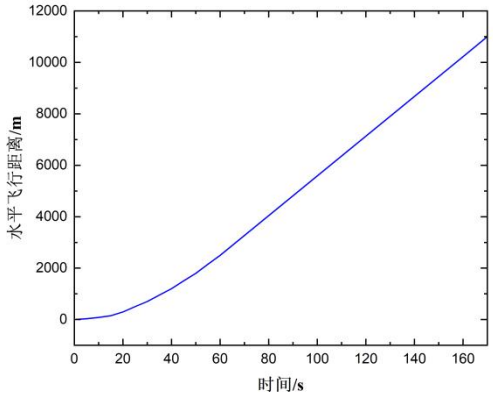


Figure 4.2 Simulation Results: Variation of Horizontal Flight Distance with Time

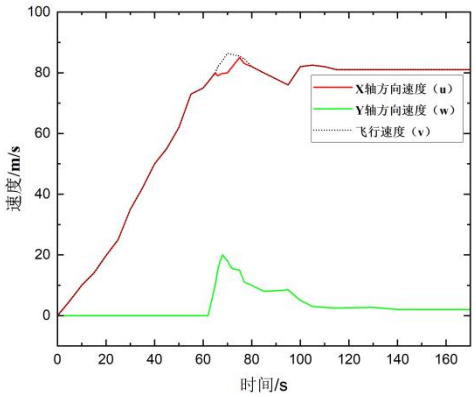


Figure 4.3 Simulation Results: Variation of Speed with Time

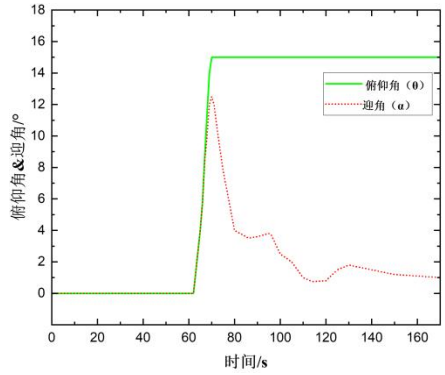


Figure 4.4 Simulation Results: Variation of Pitch Angle and Angle of Attack with Time