X 语言建模规范 3.0

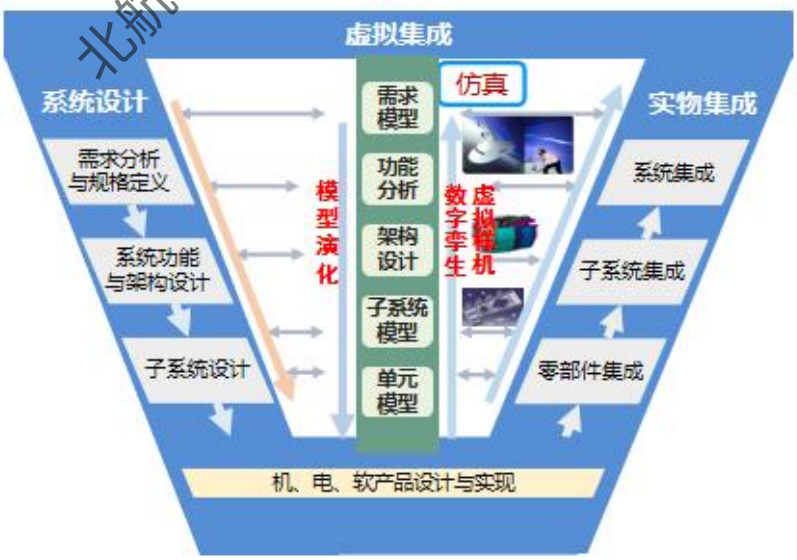
2023 年 9 月

目　　录

1 X 语言概述

1.1 X 语言研发背景

近年来，基于模型的系统工程（Model-based Systems Engineering, MBSE）已成为支持体系建模与开发的重要手段。MBSE 其核心思想是通过一个统一的、形式化、规范化的模型，来支持系统从概念设计、分析、验证到开发的全生命周期的各个阶段，使工程师之间的信息交换从传统的基于文档和物理模型驱动的的研发模式转变为基于模型驱动的研发模式。目前，支持 MBSE 的主流建模语言是由 INCOSE 联合 OMG 在统一建模语言（Unified Modeling Language, UML）基础上，开发的适用于描述工程系统的系统建模语言 SysML（System Modeling Language, SysML）。然而，由于 SysML 缺乏对产品的物理模型描述能力。系统集成阶段还是采用物理系统集成的方式，会导致开发效率低下，成本高昂，局限性大。要彻底改变传统的研发模式，实现 MBSE 的最大价值，还需要借助仿真技术，将基于物理样机的集成验证过程转变为基于数字样机的过程，即基于建模仿真的系统工程（MBSE），这样才能真正做到缩短研发周期、降低成本、提高效率，使得整个研发过程易追溯、便于维护；实现复杂产品研制所要求的一次制造成功。



目前，有两种主流的实现方法，方法一是针对机、电、液、控一体化类型的复杂产品，首先基于系统建模语言（如 SysML、IDEF 等）进行需求建模和架构设计，然后基

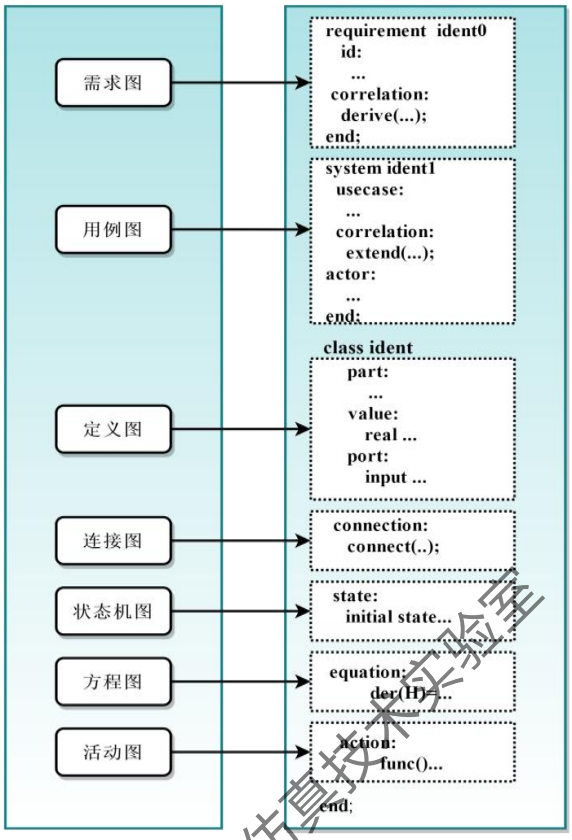
于物理建模语言（如 Modelica、Matlab/Simulink 等）并配合集成标准规范（FMI、HLA 等），实现对复杂产品系统设计和仿真的集成；方法二是通过建立系统建模语言（如 SysML、IDEF 等）和物理建模语言（如 Modelica、Matlab/Simulink 等）的映射关系实现系统模型和物理模型的自动转换，达到对复杂产品系统设计和仿真的集成。

然而，由于系统建模语言（如 SysML、IDEF 等）与物理建模语言（如 Modelica、Matlab/Simulink 等）的开发背景与面向对象的不同，导致两种异构语言的语法语义无法保持一致，从而无法完全的进行相互的映射转换。另外，实现完整的 MBSE 研发过程需要学习多种语言、多个建模平台的使用，特定语言之间转换规则的建立等等，极大的增加了建模人员学习成本。

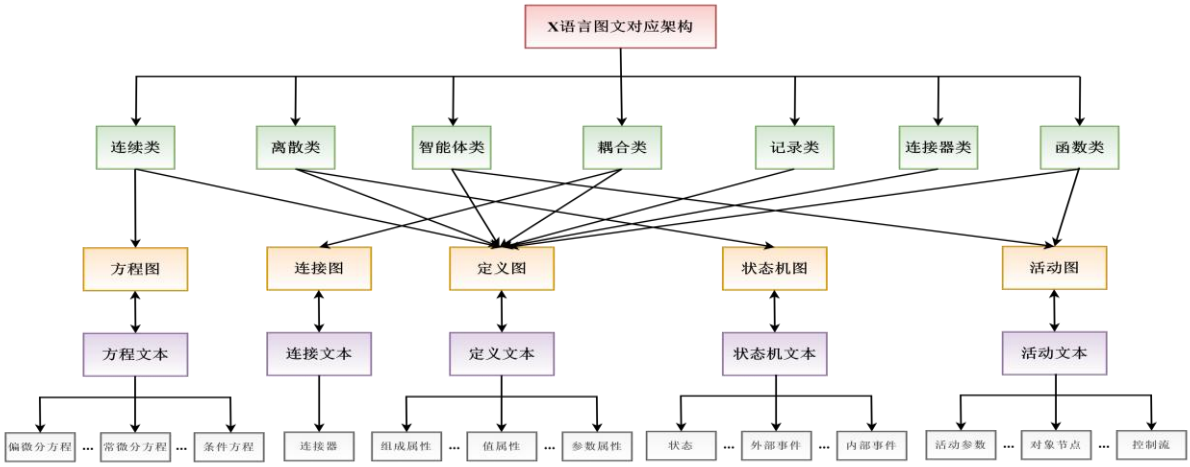
基于此，我们提出了一种面向复杂系统支持 MBSE 的新一代一体化建模仿真语言—X 语言。X 语言全面支持基于模型的系统工程（MBSE），在产品概念设计阶段提供规范的图形化建模描述，还可将规范的图形化模型自动 编译转换成文本化的底层仿真模型，在仿真引擎的驱动下，支持全系统、全流程、多视角的无缝集成仿真，实现从概念模型设计、 系统架构设计、多物理域模型到仿真模型的统一的一体化描述和仿真。

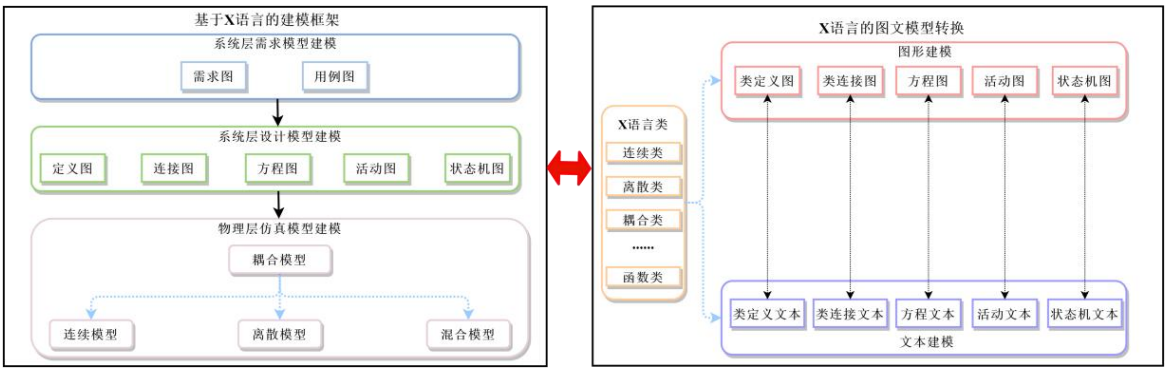
1.2 X 语言核心架构

X 语言是一种面向复杂系统支持 MBSE 的新一代一体化建模仿真语言，其设计目标是提供一种实现对复杂系统全流程（需求、设计、 验证等）、多领域（机、电、液、控等）、多粒度（零部件、组件、设备、子系统、系统乃至体系）、多特征（连续、离散、混合等）一体化建模仿真语言。

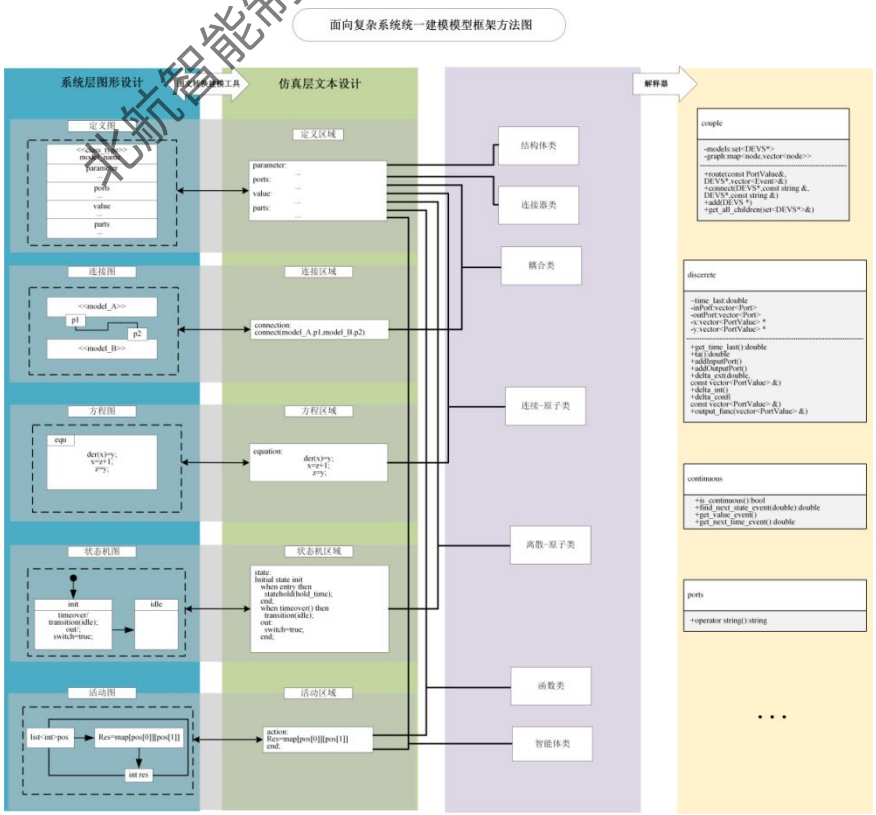


X 语言深度融合现有建模语言 SysML、Modelica 的思想及描述规范，基于 DEVS（离散事件系统规范），实现系统架构和物理特性的一体化建模，并基于统一的仿真引擎实现全系统模型的仿真验证。X 语言是一种面向对象的语言，在 X 语言中定义了七种特定类（连续、离散、耦合等），可实现对多特征，多粒度模型的描述。每种类都具有图形和文本两种建模形式，二者之间一一对应，可相互转换。





X 语言图形建模包括七种类型的图，分别是：需求图、用例图、定义图、连接图、方程图、活动图、状态机图，其中，需求图和用例图实现系统需求模型建立、明确系统功能；定义图、连接图实现模型的结构描述；方程图、活动图、状态机图实现模型的连续、离散以及连续离散混合行为的描述。X 语言通过定义图、连接图完成对系统模型的架构建模，通过方程图、活动图、状态机图补全系统功能、行为描述后，可以转换成对应的仿真文本，再通过 X 语言解释器将模型解释成可仿真文件，并由 X 语言仿真引擎实现对系统性能的验证，从而实现对复杂系统全流程、多领域、多粒度、多特征的一体化建模仿真。



X 语言面向复杂系统的建模仿真过程如上图所示，建模人员进行需求分析之后，基于定义图、连接图建立系统的顶层架构模型，基于方程图、状态机图、活动图建立不同子系统的功能、行为模型；通过自主开发的 XLab 工具将建立的复杂系统的图形化模型转换成仿真文本，生成的仿真文本可以直接发送给解释器，解释器收到仿真文本后，针对不同类文件解释生成 C++代码，最后在基于 DEVS 的仿真器上编译执行。因此，基于 X 语言，可以真正实现对复杂系统的全流程、多领域、多粒度、多特征的一体化建模仿真，使得整个研发过程易追溯、便于维护，真正做到缩短复杂产品研发周期、降低成本、提高效率。

2 X 语言关键词、操作符、数据类型及内置函数

2.1 关键字

agentover	智能体仿真结束标识	time	当前时刻仿真时间标识
infinite	离散事件仿真中的无穷大标识	flow	流变量标识
event	事件标识	input	输入端口标识
output	输出端口标识	elapsetime	状态当前已持续的时间标识
int	整型类型	real	实数类型
bool	布尔类型	string	字符类型
while	逻辑循环条件判断标识	then	条件执行标志
end	语块的结束限定	send	发送事件/消息函数
transition	状态事件转移标识	couple	耦合类标识
discrete	离散类标识	continuous	连续类标识
agent	实数型标识	when	事件触发判断标识
der	时间微分标识	for	for 循环标识
connect	模型端口连接标识	function	函数定义标识
record	特殊数据结构定义标识	receive	接受到事件发生标识
true	真	false	假
extend	继承标识	import	外部引用标识
state	状态定义标识	timeover	状态内部事件触发标识
entry	进入状态事件标识	statehold	状态持续时间设定
physical	物理通讯标识	intelligent	智能算法通讯标识

2.2 操作符

+	加号	^	乘方	( )	括号
-	减号	=	赋值	[ ]	访问下标
*	乘号	,	逗号	.	取域
/	除号	;	分号	>、<、<=、>=	比较符
++	自增 1	--	自减 1	==	恒等式
.*	矩阵乘法	./	矩阵除法		

2.3 数据类型

2.3.1 int

说明：整型变量，如：-1、0、1

声明：int a = 1;

方法：+、-、*、/。

2.3.2 real

说明：实数型（浮点型变量），如：-0.01、0.1、7.996

声明：real a = 1.1;

方法：+、-、*、/。

2.3.3 string

说明：字符串类型，如“a bsfsfsfs”、“字符串”

声明：string a = “temp”;

方法：

1) 判断长度 a.size()，返回值为无符号整型；
2) 字符串加法 a = a + “eve”，返回值为字符串；

3) 字符串索引 a[3]，返回值为对应位置字符串。

2.3.4 bool

说明：真假类型的变量，如 true、false(首字母小写)

声明：bool a = true;

2.3.5 数组

说明：固定长度的数组，与 c 语言数组一致

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

方法：访问 a[3];

2.3.6 list

说明：存储制定类型变量的不定长数据结构，支持较复杂的操作

声明：list<int> a = {1,2,3,4,5};

方法：

1) 在末尾添加元素 a.append();
2) 列表长度(空可以用 0 代替) a.size(), 返回值为无符号整型；
3) 列表访问 a[3], 返回值为对应位置元素；
4) 列表插入 a.insert(int pos, int insertVarriable)。

2.3.7 map

说明：存储键值对的字典格式，如{“a”:1, “b”:2}

声明：map<string, int> a = {“a” :1}

方法：

1) 字典访问 a[“a”] = 1;
2) 字典长度 a.size(), 返回值为无符号整型;
3) 字典添加元素 a.add();
4) 字典删除元素 a.remove();

2.4 内置函数

2.4.1 run(plan planName, …)

用于运行智能体的计划，传入的计划将会顺序执行.

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

1) 在智能体的 plan 中实现消息的发送；
2) 在 DEVS 中实现信息的输出，将输出数值发送给输出端口。

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

1) 在智能体中实现消息的接受；
2) 在离散类中实现对定义端口的事件的接受，可包括多个带那个端口参数。

2.4.4 statehold(real time1)

离散类中定义状态持续的时间。

2.4.5 entry()

定义进入状态之前需要执行的行为。

2.4.6 timeover()

定义进入状态持续时间结束之后需要执行的行为。

2.4.7 trasition(state s1)

定义进入状态结束之后进行状态转换的目标状态。

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

以所选择的 randtype 的随即方式返回 a，b 之间的随机数值。

2.4.9 void seed(real a)

设置随机数种子，必须在使用 random 函数之前使用。

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

1) 连接两个连接器时连接关系不分先后；
2) 在连接两个离散端口时输出端口为第一个参数，目标输入端口为第二个参数。

2.4.11 mathlib（函数库）

real der()，求导函数；

real sin()，正弦函数；

real cos()，余弦函数；

real tan()，正切函数；

real arccos()，反正弦函数；

real arcsin()，反余弦函数；

real arctan()，反切函数；

real sqrt()，求平方根函数；

int abs()，求绝对值函数；

int mod()，取余数函数；

real log(real a, real b)，对数函数，返回以 a 为底 b 的对数；

real ln(real a)，对数函数，返回以 e 为底 a 的对数。

3 X 语言类以及外部文件导入的描述规范

3.1 连续类

3.1.1 简介

连续原子类 continuous 类可对连续模型进行建模。连续原子类本身不可再拆分，在仿真中是最小的结构单元。

3.1.2 基本结构

continuous 类常包含头部份、定义部分、等式部分。

头部份包括导入外部模型（结构关键字 import）以及继承外部模型（结构关键字 extends）两部分内容。通过 import 部分导入外部类，通过 extends 继承父类。

定义部分描述了 continuous 类的属性，包括 parameter、value、port 三个结构关键字引领的部分。其中，parameter 部分定义了 continuous 模型的恒定属性，等号后为属性的值，属性在仿真中不会更改，是个常数，比如恒温电阻的阻值；value 部分定义了模型在仿真过程中的变量，如果变量后有等号，则等号后为变量的初始值，如果变量恒定，则可在其前加上限定词 constant；port 部分定义了模型的端口部分即输入输出部分，这部分主要有两种表达形式，一种是用 input/output 作为限定词修饰的量，另一种是对 connector 类的实例化（connector 类将在后文详细介绍）。

等式部分描述了 continuous 类的行为。该部分以 equation 作为关键字，所使用的变量以及常量都是在定义部分定义过的量。der()作为内置微分函数表示对变量的微分运算，此外，此部分还可以应用 if，for 等结构体来描述模型的行为。注意，等式部分的等于号表示的是方程中的一种相等的约束关系，并不表赋值。

下面给出 continuous 类的一个定义模板：

```
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; //实例化导入的 out_record1 命名为 argname3
    …
value:
    datatype varname1;
    datatype varname2;
    out_record1 varname3; //实例化导入的 out_record1 命名为 varname3
    …
port:
    input datatype argname1;
    output datatype argname2;
    out_connector1 argname3;//实例化导入的 out_connector1 命名为 argname3
    out_connector1 argname4;
    …
equation:
    out_function1();//使用导入的 out_function1 函数
    //此部分对模型各变量之间的关系进行描述
```


3.1.3 用法

在 continuous 类中，import 后的模型类常常是 connector 类，function 类，record 类以及 continuous 类，其中导入的 continuous 类只能用于继承，connector 类用于后面 port 部分的实例化，function 类用于 equation 部分的等式建立，record 类用于参数定义。extends 可用于对于模型的继承，在 continuous 类中，继承的对象只能是 continuous 类。

continuous 类的调用有两种。一种是被 continuous 类调用并继承，另一种是作为 couple 类的组件被调用并实例化。

3.1.4 示例代码

下例对一阶系统进行建模。

```
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 离散类

3.2.1 简介

离散原子类 discrete 类可对离散模型进行建模。离散原子类本身不可再分，是离散模型最小的仿真单元。与连续原子类类似，最小的仿真单元并不意味着描述能力上离散原子类仅仅能建立“小”模型。例如，对于某复杂事件的建模，既可以将复杂事件拆解成简单事件再加以连接，也可以直接对复杂事件整体进行建模，只要逻辑关系满足即可。

3.2.2 基本结构

discrete 类是用来描述复杂产品中的原子模型（离散行为）。一般地，discrete 类是由头部，定义部分与状态部分三个部分组成。头部对外部类进行导入或者是进行继承操作，定义部分用来初始化参数和变量的值，以及相关组件和输入输出端口；state 部分用来定义原子模型的状态以及状态之间的转移逻辑。

以下面的示例代码来说明 discrete 类的基本结构及组成。

头部主要进行对外部类的导入（import）和继承（extend）操作。Import 指的是下文需要对外部类进行实例化，extends 可用于对于模型的继承，在 discrete 类中，继承的对象只能是 discrete 类。

定义部分对 discrete 类的属性参数进行描述，包括参数属性（parameter）、值属性（value）和端口属性（port）。其中 parameter 与 value 与 continuous 类中的功能相似，port 中不再含有 connector 连接类的实例化，并且需要在 input/output 前加上限定词 event 来表示事件的输入输出。其中输入端口的数值由外部模型进行控制，且在两次接受输入期间，输入端口所存储的数值保持不变，可视作常量，输出变量则类似变量。而 value 模块中声明的变量可以定义初始值，在仿真初始化阶段赋了初始值的变量将被初始化为初始值。

状态部分是离散原子类独有的部分，以关键词 state 引领该部分内容，以表示离散事件中不同状态之间的转移情况和转移条件。state 部分由多个状态机组成，每个状态机都由 state 与此状态机名作为开端，并以 end 结束。初始状态用关键字 inital state 加初始状态机名字组成。相对来说，状态分为两种状态：复合状态和简单状态。复合状态即状态内能包括其他简单状态和复合状态。如对于下面的例子来说，离散模型 daodanshiti 中有两种状态，分别为 work 和 send，其中 work 是初始状态，即用关键字 inital state 引领。

而对于 state 中的一些状态转移行为，可用一些固定的函数来进行表示，主要函数如下：

行为名称	用途
Entry 语句	定义进入状态之前需要进行的行为（一般包括对状态持续时间的声明）
Receive 语句	定义状态在接受特定输入时的行为及状态转换
State-event 语句	定义状态变量满足特定条件时的行为以及状态转换
Time-event 语句	定义状态持续时间结束时的行为以及状态转换
Catch-equation 语句	定义状态在其持续期间的连续行为，使用方程定义

下面给出 discrete 类的一个定义模板：

```
discrete DiscreteName
    import out_discrete1;
        extends out_discrete1;
```

```
parameter:
        datatype argname1;
    out_ discrete1 argname2; //实例化导入的 out_record1 命名为 argname2
value:
     datatype varname1;
    out_ discrete1 varname2; //实例化导入的 out_record1 命名为 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 复合状态

复合状态内部可以定义其他复合状态或者简单状态。复合状态的行为定义相比简单状态有以下需要注意的地方：

复合状态中同时只能有一个状态处于激活的状态，不能同时多个状态被激活。

1. 复合状态定义的行为对定义在它内部的状态而言同样有效，如，复合状态中使

用了 receive 语句，那么复合状态内定义的所有状态都能触发该语句。如下面的复合状态 StateName1 所示，其内部包括了初始状态 StateName2，虽然 StateName2 内没有定义 receive 语句，但是当处于 StateName2 状态时，如果接收到 ReceiveEvent1，也会触发在 StateName1 内定义的 receive 行为。同时，复合状态中定义的行为如果在其内部的状态中被重新定义，那么以内部状态定义的为准。如 StateName3 状态中针对 receive（ReceiveEvent1）这一行为进行了重定义，那当该事件触发时，以 StateName3 中的定义为准。

```
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 用法

3.2.4.1 entry 语句

此函数在 state 中表示进入该状态前需要执行的行为。典型的行为包括状态持续时间，使用 statehold 语句定义，需要注意的是，未在 entry 语句中声明持续时间或未定义 entry 语句的状态默认持续时间为 infinite（无限）。在 entry 语句中，能够迭代如 if 等陈述语句。如上文案例中离散模型的文本程序：

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

即表示此状态的持续时间为无限长，即没有时间控制此状态的转移和继续运行情况，此状态转移和后续功能运行只由输入的相关事件触发。（此部分一般搭配 timeover 语句实现）

3.2.4.2 receive 语句

Receive 语句为外部事件，其核心语句 receive 定义了哪些端口接收到消息时该事件行为会被触发。Receive 语句中可以包括一个或多个参数，每一个参数都必须对应到离散类中声明的输入接口。

外部事件将会直接导致状态的转移，此时使用 transition 语句表示状态的转移，transition 语句的参数必须为 State 模块中定义的多个状态中的一个，且状态可以实现自循环，既可以实现从一个状态转移到他自己。此语句以 end 结束。

如上文案例中离散模型的文本程序：

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

此程序即表示在接收到 ReceiveEvent1 事件后，在运算后通过 transition 语句完成由当前状态向 StateName2 状态的转移过程。

3.2.4.3 state event 和 time event 语句

此两种语言定义了状态转移的另外两种情况，即由方程或状态变量的改变而引起的

连续行为和时间限制对于状态的影响。

State-event 语句一般用于和 catch-equation 语句一起使用，因为在普通的状态中，状态变量在进入状态后都将保持不变，所以定义状态事件缺乏实际意义。而 catch-equation 可以描述在状态持续时间内以方程为基础的连续行为，即在状态持续时间内状态变量在方程的求解推进过程中将会不断改变，一旦满足 State-event 语句中所声明的条件，即可触发状态事件。

Time-event 语句定义了 DEVS 中普遍的内部事件，即当状态在 entry 语句中定义的状态持续时间结束时，将会触发时间事件，时间事件使用 timeover 表示。

在此两种语句中，可以额外描述输出。当内部行为描述完毕，在两个规则各自的 out 部分，可以定义输出，输出到对应端口使用 send 语句表示。Send 函数表示输出到对应端口，send 函数包括两个参数，第一个参数为模型的输出端口，第二个参数为输出到端口的数值。

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

此段程序表示此状态在经过 entry 中定义的时间（在本例中为 0）后，进行 transition 中写明的状态转移，即由 send 状态转移至 StateName1 状态。同时将 value 输出至对应端口 port1。

同样，还可以用连续事件的变化来标志状态转移，如下图案例：

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

此案例表示当 h<10 时进行状态转移，并将 value 值输出至端口 port1。

3.2.4.4 for-equation

for 方程用于定义数组方程，即数组的每个元素能够满足的方程。由 for 循环变量的范围定义 for 方程循环的次数，在计算方程数目时，for 方程的数目将会被计算为其循环的次数。

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

此案例即为变量 i 由 1 变化到 10 进行循环。

3.2.4.5 if-equation

if 方程由一个 if 子句、若干个 else if 子句以及 0 个或者 1 个 else 子句构成，其中 expression 必须能够转化为 bool 表达式。在解算过程中，依次评估 if 和 else if 的判断条件，如果有一个条件为真则选择该条件下的 equation 进行解算，否则，如果存在 else 语句块则选择 else 语句块下的 equation 进行解算，如果不存在则判断结束，不执行任何 equation。

除了上述条件之外，为了保证在各个条件下方程的数量是固定不变的，也就是为了保证方程组一定是可解的，在 if-equation 的各个分支中包含的方程的数量必须一致。

```
if expression then
    ...
else if expression 2 then
    ...
end;
```


3.2.5 建模案例

下例为水箱模型的 X 语言建模形式：

```
discrete LiquidSource
    parameter:
        real flowLevel = 0.02;
    port:
        event output real qOut;
    state:
        initial state init
            when entry() then
```

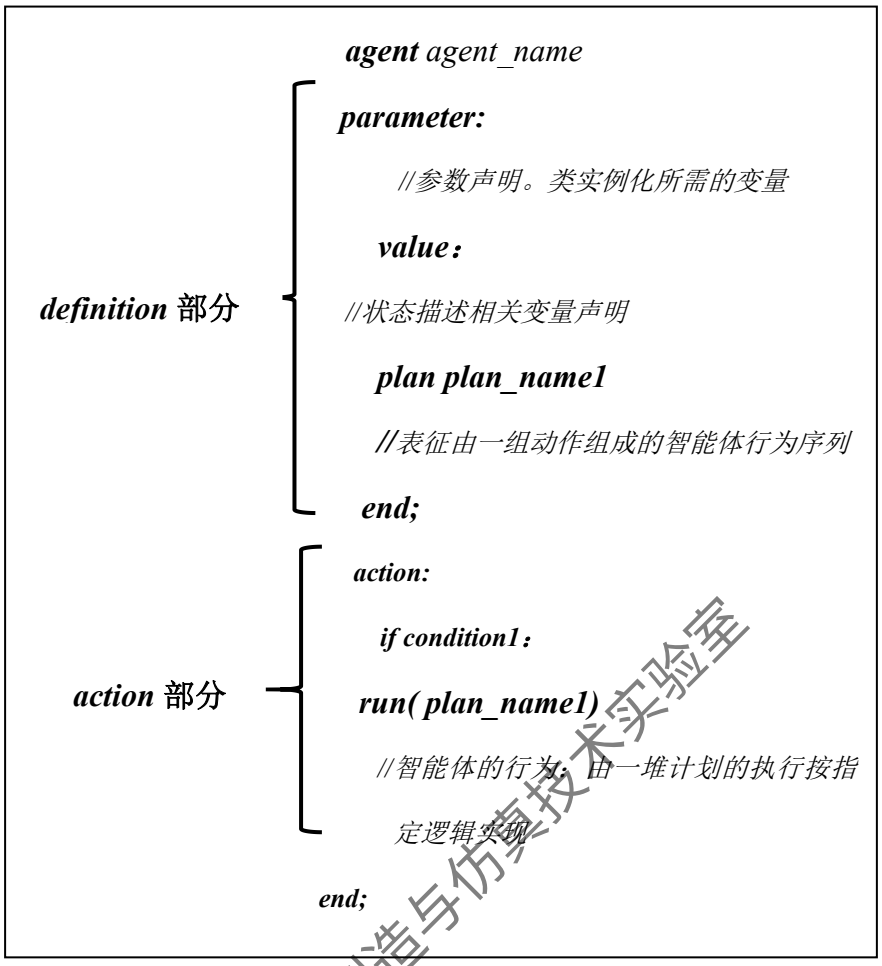
```
                statehold(0);
            end;
            when timeover() then
                transition(pass);
            out：
                send(qOut,flowLevel);
            end;
        end;
        state pass
            when entry() then
                statehold(150);
            end;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

3.3 智能体类

3.3.1 简介

agent 类是用来描述复杂产品中的智能体模型。一般地，agent 类是由 definition 部分和 action 部分两个部分组成。definition 部分用来初始化参数和变量的值，以及函数和计划的声明也在该部分完成；action 部分用来控制计划的执行，以及设置智能体仿真的开始和终止条件。

agent 类描述架构如下图：



3.3.2 基本结构

智能体模型的创建由 agent 类组成，其中 agent 中的上半部分为智能体的定义部分，即变量、函数、与计划的定义部分，而 action 为对计划的调用以及控制智能体生命周期的部分。整个 agent 类相当于以继承的方式被使用。

例如：

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

3.3.3 用法

整体的智能体实例化与运行方式与 devs 的原子模型加入耦合模型仿真器一致，变量类型为 agent 基类。

例如：

```
import BaseAgent;
agent base1 = BaseAgent.baseagent("base1", [50, 0, 10], "imissile1");  // 实例化方法
```

代表定义一个具有完整生命周期过程的智能体，后面接的智能体名字在全局是唯一的。

用于定义变量，函数，计划

智能体的实例化参数使用 parameter 传入。

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

智能体的动作连接或者调用的部分，

run 表示连接 plan 的执行顺序，理解为计划执行链。

例如：

```
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：

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

```
end;
```

发送消息的动作，是一个函数，链接了智能体类中的消息存储变量，一旦声明就代表一个消息被发送到指定的智能体对象上。

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

当智能体没有收到 receive 时，整个就会处于挂起状态，不执行其他语句直到消息的到来。

接受来自智能体 agt.mailbox 的消息内容，按照消息的具体协议内容执行相关的操作。重要的作用就是对于传给智能体的消息进行解读！同时需要发送反馈的信息。receive()是解析消息的内容，通过协议理解语义，将信息按照 message 格式进行解读。

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

receive 动作只有在收到信息后才会进行完成，若 agent 一直没有收到消息，则 receive 动作会一直处于挂起状态，直到有消息进入 agt.mailbox。

plan 在上层文本类似一个函数，输入与输出全靠全局变量来控制，send 和 receive 方法只能由 plan 模块进行调用。

例如：

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

```
    msg3_content = receive();
end;
```

3.4 耦合类

3.4.1 简介

耦合类是 X 语言的一个重要的类，也是 X 语言实现仿真级建模与系统建模联通的关键。耦合模型在 X 语言中承担了连接多领域模型仿真的任务，在耦合模型中，不仅能够连接相同领域（如离散模型），也能够连接不同领域的模型，只需要二者之间的端口能够相互匹配。是 X 语言中实现复杂系统建模与仿真的主要类工具。

3.4.2 基本结构

耦合类以关键字 couple 统领。耦合类包括头部份、属性部分、连接部分等三个部分。

头部份包括导入外部模型（结构关键字 import）以及继承外部模型（结构关键字 extends）两部分内容。Extends 部分的类只能是 couple 类，import 部分根据需要导入外部类，以便继承或者实例化。属性部分对于模型基本属性进行描述，连接部分对于不同子模型之间的连接关系进行描述，表示子模型两个端口之间的有向连接。

3.4.3 用法

属性部分包括 parameter、port、part 三个部分。Parameter 表示 couple 类的固有属性，为常数。Port 表示 couple 类的接口。Part 部分表示 couple 包含的子模块，内容包含 continuous、discrete、couple 类的实例化。

连接部分以 connection 关键字统领，内部以 connect 的形式连接各个端口。connect 连接的两端连接的量的属性必须一致，可以同为单变量，但是需要值的数值类型一致，也可以同为 connector 类。此连接关系有方向，用 connect 连接的两个端口表示从第一个端口向第二个端口的有向连接。

例如下例中，导入了 connector、continuous、discrete、couple 四个模型，并继承了 couple_1。part 部分，对于 continuous_1 与 discrete 类进行实例化。port 部分，利用导入的 connector_1 定义自己的端口。connection 部分，将 name_2 的 p 端口与 name_3 的

n 端口相连（从 name_2 的 p 端口到 name_3 的 n 端口），将 name_2 的 n 端口与本 couple 模型的 name_1 端口相连，并且，由于 name_1 为 connector 类，可以推测 name_2 的 p 端口也是 connector 类。

示例代码如下：

```
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 建模案例（水箱模型）

```
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 连接器类

3.5.1 简介

连接器类是实现组件化建模的关键的一种类，connector 是一种让模型与模型交换信息的方法。connector 主要针对物理系统的非因果建模。非因果物理建模方法区分了两类不同的变量：流变量与势变量。一般用 connector 来表示两个组件端口之间具有的绑定语义或者满足基尔霍夫定理，即，势变量相等，流变量之和等于 0。

3.5.2 基本结构

连接器类以关键字 connector 统领。只包含 value 一个部分。value 部分可以包含多个不同数据类型的量。一般在建模过程中，连续类会调用连接器类实现非因果建模。所使用的数据类型为 real。下面给出 connector 类的定义模板：

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

3.5.3 用法

connector 类只能被 continuous 类与 discrete 类调用。实例化是在调用主体的 port 部分里进行定义。

3.5.4 示例代码

下面的例子定义了电路中的正极子。

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

3.6 记录类

3.6.1 简介

记录类是 X 语言中聚合数据类型的一类。记录类可以有自己的变量，用于实现较复杂的数据结构。

3.6.2 基本结构

记录类以关键字 record 统领。只包含 value 一个部分。value 部分可以包含多个不同数据类型的量，用于表达复杂的数据结构。具体描述形式模板如下：

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

3.6.3 用法（record 的构造函数）

现在，我们已经知道了如何定义一个 record 类型。那应该如何创建 record 类型呢？如果我们想声明一个变量，恰巧这个变量属于 record 类型，变量声明的本身就会创建一个 record 类型实例，而且我们还可以通过修改语句指定 record 类型内的变量值，例如：

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

但在某些情况下，我们可能希望创建一个 record 类型而不是一个变量（例如，在表达式中使用，作为参数传递给函数或者在修改功能中使用）。对每个 record 类型定义，都会自动生成与 record 类型名称完全相同的函数名。这个函数称为“记录构造函数”。记录构造函数输入与 record 类型内部定义相匹配的变量，并返回一个 record 类型实例。所以，在上述 Vector 定义实例中，我们也可以通过记录构造函数初始化 parameter 变量，如下所示：

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

在这种情况下，变量 v 的值通过表达式 Vector(x=1.0, y=2.0, z=0.0)调用记录构造函数进行赋值。

3.6.4 示例代码

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

3.7 函数类

3.7.1 简介

函数类可以方便得描述物体的数学特性，有时候有必要创建用于特殊目的的新函数，用函数类进行创建。

3.7.2 基本结构

函数类以关键字 function 统领。包含头部份、定义部分、行为部分。头部份描述了导入的外部函数。定义部分包括 port 与 value 两个部分，port 中定义了函数的输入输出：函数的输入参数、返回值定义在 port 中，输入参数以限定词 input 进行声明，返回值以 output 进行声明。行为部分以关键字 action 引领，action 部分可以用于描述输入输出之间的关系。

考虑上述所有情况，下面给出函数类定义的模板：

```
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 用法

函数类可以被 continuous、discrete、function 类调用。调用时，需要在头部进行 import 导入。

3.7.4 示例代码

下面对一个分段函数进行建模。

```
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 模块类

3.8.1 简介

模块类是 X 语言中用于描述因果模块图这种具有特定因果关系的仿真模型的描述架构。模块类不同于函数类，模块类能够描述与仿真时间相关的操作，如积分和微分。

3.8.2 基本结构

模块类以关键字 block 统领。包含头部份、定义部分、行为部分。头部份描述了导入的外部函数。定义部分包括 port 与 value 两个部分，port 中定义了函数的输入输出：模块的输入参数、输出都在 port 中，输入参数以限定词 input 进行声明，输出以 output

进行声明。行为部分以关键字 action 引领，action 部分可以用于描述输入输出之间的关系，以及定义与仿真时间相关的操作。

考虑上述所有情况，下面给出函数类定义的模板：

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 用法

模块类可以作为模块参与到 couple 类的构建中，并通过因果关系确定的连接来实现对控制模型等多领域模型的建模和仿真。

3.8.4 示例代码

下面对一个分段函数进行建模。

block differential:

port:

```
input real x;
output real y;
action:
    y = der(x);
end;
```

3.9 X 语言对外部文件的导入

3.9.1 简介

在导入外部文件时，需要使用外部导入函数，外部函数目前定义了三种类型的外部模型导入的方法：DLL 连接库模型、FMU 模型和 Julia 模型。

3.9.2 基本语法

外部导入函数的关键字为 external ()，该语句只能在函数类中进行使用，具体用法如案例所示。函数类中的头部分和普通的函数一致。在定义部分中的 port 需要同外部导入的函数的输入输出参数对应，如果外部导入函数的参数对应需要中间变量的辅助，在 value 部分中定义中间变量。在 action 部分可调用 external 函数对外部文件进行导入，external 函数的第一个参数标识导入的文件的类型，如 Julia。后续参数依据不同类型进行不同的定义，如 Julica 文件的导入只需要补充 Julia 文件的存储位置即可。需要注意的是，一个函数中只能导入唯一的一个外部文件。且外部导入只表示该文件被导入，并不等同于该函数被调用，只有当导入了该文件的 X 语言函数被调用该函数才会被调用执行行为。

考虑上述所有情况，下面给出外部函数导入格式的模板：

```
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 X 语言详细规范总结

4.1 X 语言巴克斯范式

//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 语言图形描述规范

5.1 需求图

5.1.1 目的

基于文字的需求在传统上是系统工程的重要产品。这并不意味着所有方法都需要基于文字的需求。越来越被广泛使用的技术是创建用例来替代基于文字的功能性需求，创建约束表达式来替换基于文字的非功能性需求。然而，当我们需要显示这些需求，以及它们与其他模型元素之间的关系时，就可以创建需求图。

当看图者需要看到从需求到系统模型中依赖于它的元素的可跟踪性时，这张图尤其有价值。

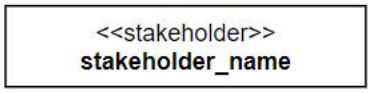
5.1.2 何时创建需求图

向模型增加新的元素时，你会创建一种关系，从那些元素指向驱动创建它们的需求。以这种方式确立需求的可跟踪性是贯穿设计和开发的活动。你可能需要创建需求图，在这项工作的任何时间点显示那些关系。

5.1.3 利益相关者

利益相关者代表整个项目的需求方或者对需求的负责方，是整个项目开发的需求提出者或践行者。利益相关者的标识符是一个矩形，在名称之前有元类型<<stakeholder>>。

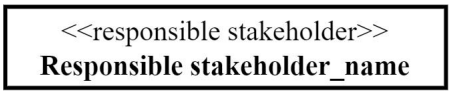
具体形式如下：



5.1.4 责任利益相关者

责任利益相关者代表整个项目的对需求的负责方，是整个项目开发的践行者。责任利益相关者的标识符是一个矩形，在名称之前有元类型<<responsible_stakeholder>>。

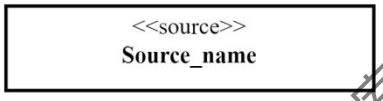
具体形式如下：



5.1.5 需求源

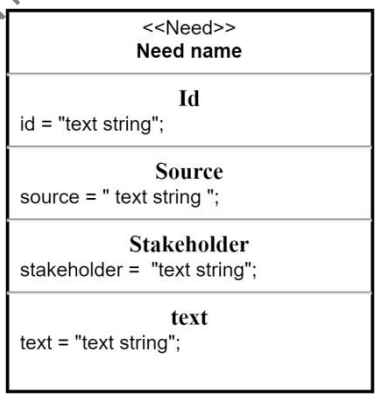
需求源代表整个项目需求的起源，可能是某个官方单位或者需求规范文档等。需求源的标识符是一个矩形，在名称之前有元类型<<source>>。

具体形式如下：



5.1.6 利益相关者需求（stakeholder needs）

利益相关者需求的标识符是一个矩形，在名称之前有元类型<<need>>。利益相关者需求有 4 个属性：id、source、stakeholder、text，这四种属性的类型都是 string。其中，id 表示该需求的编号、source 表示需求的来源、stakeholder 表示提出需求的利益相关者、text 一般通过自然语言描述需求的具体内容（利益相关者的需求往往没有具体的约束）。



5.1.6.1 文本示例

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 需求（requirement，可以验证的）

需求的标识法是一个矩形，在名称之前有元类型<<requirement>>。需求有 7 个属性：id、responsible stakeholder、priority、type、level、role、text，这四种属性的类型都是 string。其中，id 表示该需求的编号、responsible stakeholder 表示需求的践行负责方、type 代表需求的类型（一般包括功能性需求和非功能性需求，非功能性需求可细分为性能需求、设计需求、环境需求、适用性需求）、level 表示需求的层级（一般包括系统级需求以及组件级需求）、role 代表面向 MDAO 的需求角色（一般包括设计变量、设计变量边界、输入参数、约束、目标对象）text 一般通过自然语言描述需求的具体内容（针对不同类型的需求，需求的描述有其约束格式）。

具体形式如下：

<<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 文本示例

requirement Requirement_name:

Id : "text string"; 编号

Responsible stakeholder: "text string"; 负责实现或执行该需求的利益相关者

Priority: High|Medium|Low; 需求的优先级

Type:Functional|Nonfunctional|Performance|Design(constraint)|Environmental|Suitability; 需求的具体类型：功能需求和非功能需求（包括：性能需求、设计需求、环境需求、适用性需求）

Level: System requirement|Component requirement; 需求的层级：系统级需求和组件级需求

Role: Design variable|Design variable bound|Input parameter|Contraint|Objevtive; 面向 MDAO 的需求角色：设计变量、设计变量边界、输入参数、约束、对象

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]"

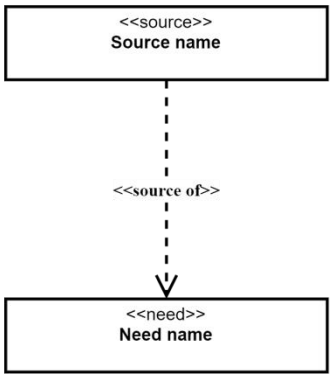
end;

5.1.8 需求关系

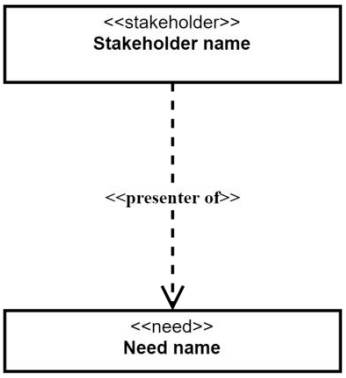
在系统模型中记录需求很有用。然而，需求和其他模型元素直接的关系有更大的价值。在建模过程中，一般可能会使用十种需求关系：来源、提出、负责、包含、跟踪、继承、改善、满足、验证、映射。

这些关系在系统模型中确立了需求的可跟踪性，这在系统工程组织中一般是一个过程需求。然而，从实践出发，在模型中记录这些关系可以让你使用建模工具自动生成需求可跟踪性，并在需求发生变更时，执行自动下游的影响分析。这些功能会节省大量时间，而那会直接转换成对成本的节省。

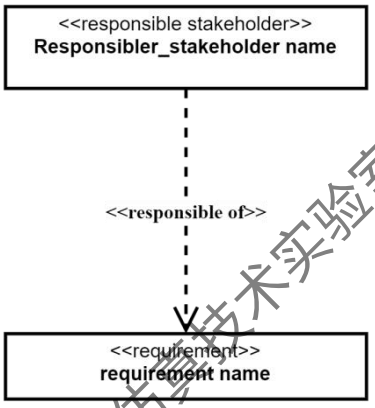
需求来源于某一官方单位或需求规范文档，来源关系描述需求的起源。具体形式如下：



需求一般由所有的利益相关方提出，提出关系描述与原始需求直接相关的利益相关者。具体形式如下：



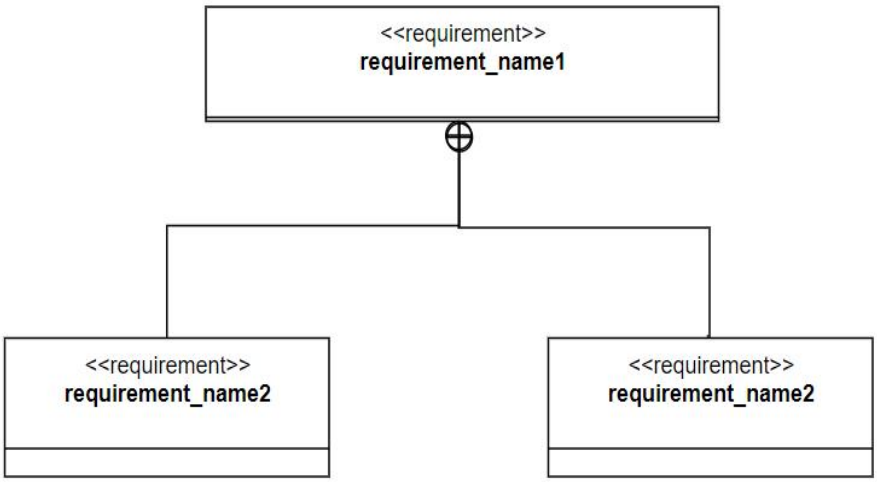
需求在设计过程中对会有相应的践行负责的利益相关者。具体形式如下：



5.1.8.1 包含关系

需求之间存在包含关系，即需求可以包含其他需求。一般通过十字准线标识法表示需求之间的包含关系。

具体形式如下：

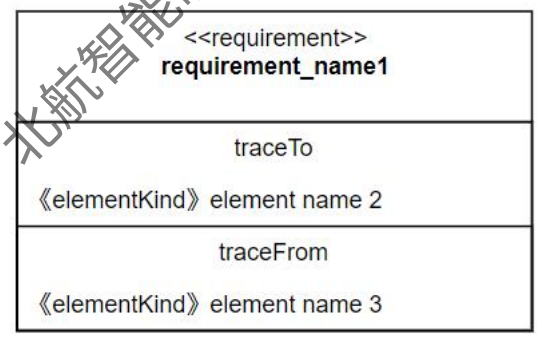


图中表示需求 1 包含需求 2 和需求 3。

5.1.8.2 跟踪关系

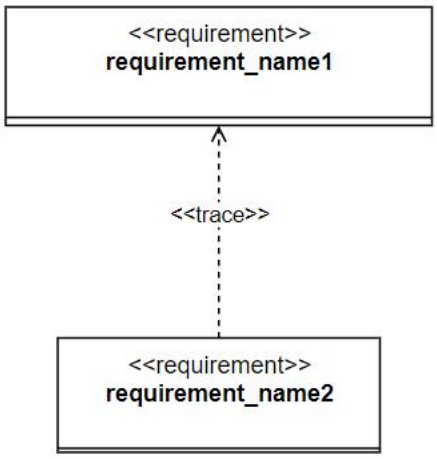
正式情况下，跟踪关系是一种依赖关系。跟踪关系是一种弱关系。它只是表达了一种基本的依赖关系：对提供方元素的修改可能会导致对客户端元素修改的需要。跟踪关系一般可以通过需求属性栏或者通过带有开口箭头的虚线（上方带有<<trace>>元类型）表示。

方式一：需求属性栏表示法



图中 traceTo 表示需求 1 会跟踪回需求 2，traceFrom 表示需求 3 会跟踪回需求 1。

方式二：带有开口箭头的虚线（上方带有<<trace>>元类型）表示法

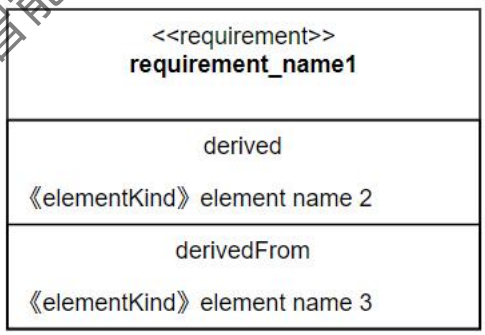


图中表示需求 2 会跟踪回需求 1

5.1.8.3 继承需求关系

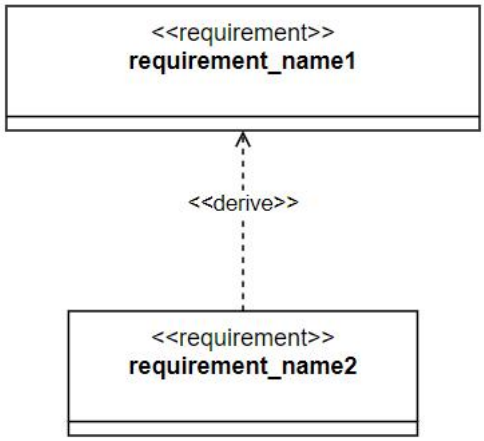
继承需求关系是另一种依赖关系。这种关系必须在客户端和提供方端都有需求。继承需求关系表示客户端的需求继承了提供方的需求。继承需求关系一般可以通过需求属性栏或者通过带有开口箭头的虚线（上方带有<<derive>>元类型）表示。拥有多级继承关系完全是合法的，并且依赖关系是可传递的。因此，如果基本的需求发生变更，那么下游的影响会贯穿整个继承需求关系链。

方式一：需求属性栏表示法



图中 derived 表示需求 2 继承自需求 1，derivedFrom 表示需求 1 继承自需求 3。

方式二：带有开口箭头的虚线（上方带有<<derive>>元类型）表示法

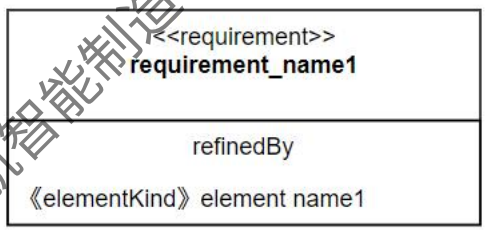


图中表示需求 2 继承自需求 1

5.1.8.4改善关系

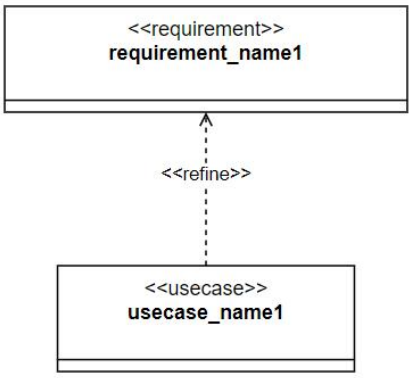
改善关系是另一种依赖关系。对于改善关系的任意一端所能够显示的元素种类，X 语言没有任何限制。然而，一般使用用例对文本的功能性需求进行改善。改善关系表示客户端的元素要比提供方端的元素更加具体。改善关系一般可以通过需求属性栏或者通过带有开口箭头的虚线（上方带有<<refine>>元类型）表示。

方式一：需求属性栏表示法



图中 refinedBy 表示元素 1 改善了需求 1

方式二：带有开口箭头的虚线（上方带有<<refine>>元类型）表示法

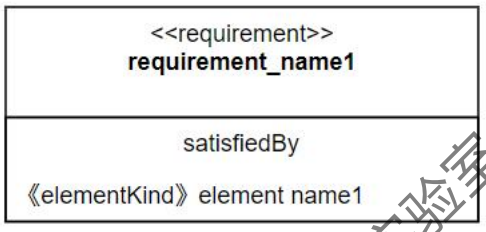


图中表示用例 1 改善了需求 1

5.1.8.5满足关系

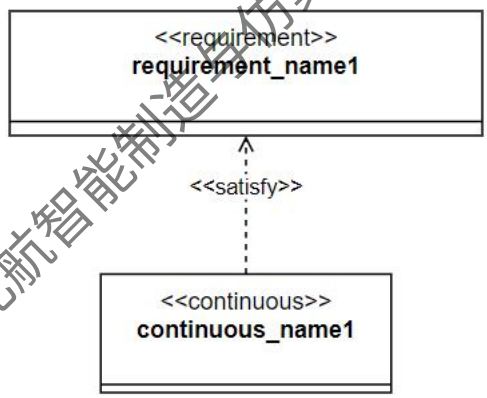
满足关系是另一种依赖关系。这种关系在提供方端必须有一个需求。X 语言没有对客户端所能够出现的元素种类施加限制。然而，客户端元素通常是类（连续类、离散类等）。满足关系是一种断言，说明客户端类的实例会满足提供方端的需求。满足关系一般可以通过需求属性栏或者通过带有开口箭头的虚线（上方带有<<satisfy>>元类型）表示。

方式一：需求属性栏表示法



图中 satisfiedBy 表示元素 1（可以是连续类、离散类等）会满足需求 1。

方式二：带有开口箭头的虚线（上方带有<<satisfy>>元类型）表示法

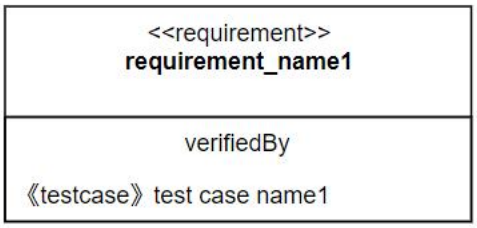


图中表示连续类 1 会满足需求 1

5.1.8.6验证关系

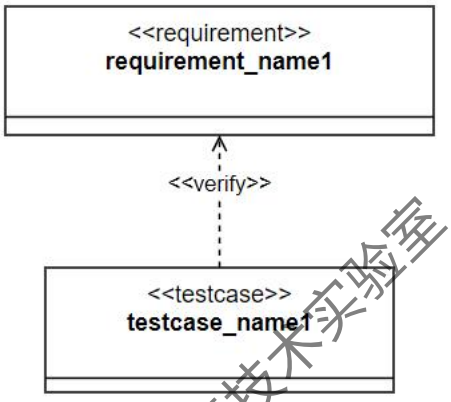
验证关系是另一种依赖关系。和满足关系一样，验证关系必须在提供方端有一个需求。X 语言没有对客户端所能够出现的元素种类施加限制。然而，客户端元素通常是测试案例。测试案例一般会是一种系统行为，当进行仿真的时候，会证明系统是否真正满足了需求。验证关系一般可以通过需求属性栏或者通过带有开口箭头的虚线（上方带有<<verify>>元类型）表示。

方式一：需求属性栏表示法



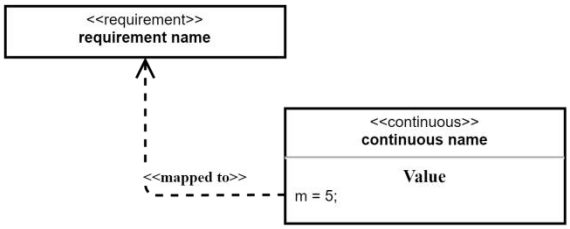
图中 verifiedBy 表示测试案例 1 验证了需求 1

方式二：带有开口箭头的虚线（上方带有<<verify>>元类型）表示法



图中表示测试案例 1 验证了需求 1

映射关系是另一种依赖关系，和验证一样，映射关系必须在提供方端有一个需求。客户端一般是对应模型元素内部具体的属性或行为。映射关系的设定目的是建立需求和实现其对应的模型元素的具体属性或行为的关联关系。映射关系一般可以通过带有开口箭头的虚线（上方带有<<mapped to>>元类型）表示。



5.1.9 需求文本

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|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;

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 用例图

5.2.1 目的

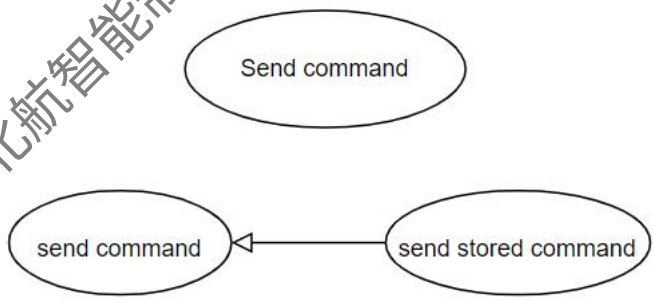
用例图会简洁地传递一系列用例——系统提供的外部可见服务——以及出触发和参与用例的执行者。用例图是系统的一种黑盒视图，因此也很适合作为系统的情境图。

5.2.2 何时创建用例图

用例图是一种分析工具，一般会在系统生命周期的早期创建。系统分析师可能会枚举各种用例，然后在系统概念和操作的开发阶段创建用例图。在某些方法中，分析师会在系统生命周期的需求引出和指定阶段，以基于文本的功能性需求方式创建用例。

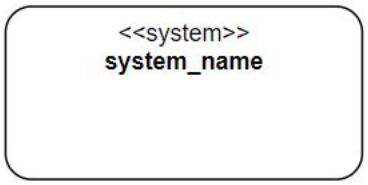
5.2.3 用例

用例图的标识法是一个椭圆形。一般用例的名称是一个动词短语（在椭圆中）。用例可以泛化，也可以特殊化，这意味着你可以创建并显示从一个用例到另一个用例的泛化关系。



5.2.4 系统边界

系统边界代表拥有并执行用例的系统。系统边界的标识法是围绕用例的矩形框。主题的名称—显示在矩形的顶部—必须是一个名词短语。

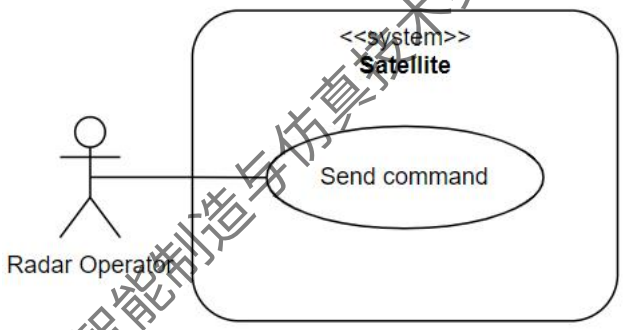


5.2.5 执行者

执行者有两种标识法：火柴棍小人，或者是名称前面带有<<actor>>关键字的矩形。一般建模者会使用火柴棍小人代表人，矩形标识法代表系统。

5.2.6 执行者与用例关联

一般会在执行者和用例之间创建关联。从而表示执行者与系统交互，以触发和参与到用例中。

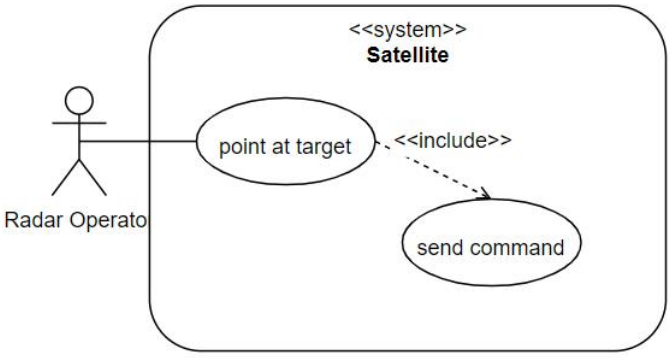


5.2.7 基础用例

基础用例是通过关联关系与主执行者连接在一起的任意用例。这意味着基础用例代表的是主执行者的目标。

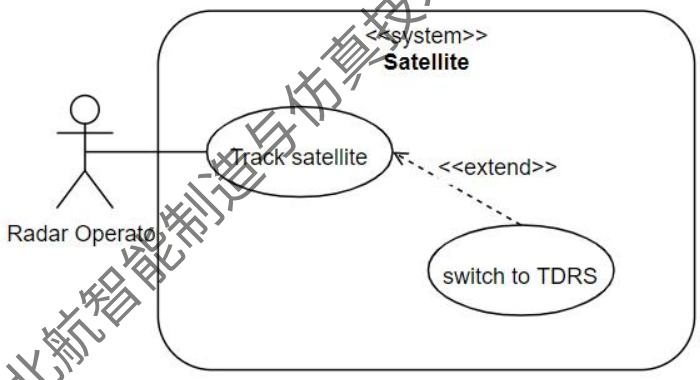
5.2.8 内含用例

内含用例是任意一种用例，它是内含关系的目标—也就是位于箭头端的元素。内含关系的标识法是带有箭头的虚线，在旁边会有关键字<<include>>。



5.2.9 扩展用例

扩展用例是任意一种用例。它是扩展关系的源—位于尾端的元素。扩展关系的标识法是带有箭头的虚线，附近带有关键字<<extend>>。



5.3 定义图

5.3.1 目的

定义图是 X 语言中特定类的结构属性（输入输出端口、参数、状态变量等）的图形表达形式

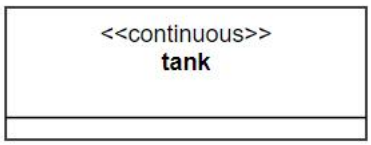
5.3.2 何时创建定义图

经常。当我们对一个系统进行建模的时候，基本上都会创建定义图。定义图通常会和其他图组合去描述 X 语言的特定类（比如连续类、离散类等）。

5.3.3 定义图的元素和关系

5.3.3.1 类

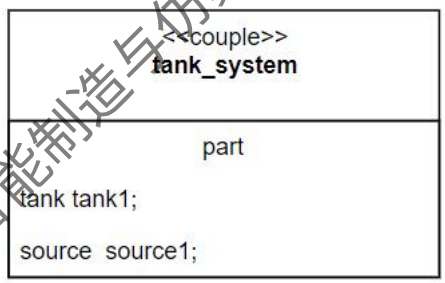
类（包含一般类和特定类）是 X 语言结构中的基本单元。可以使用类为系统中或者系统外部环境中任意感兴趣的实体类型创建模型。类一般是带有元类型<<class|couple|continuous|discrete...>>的矩形。



图中表示一个名为 tank 的连续类模型

5.3.3.2 组成部分属性

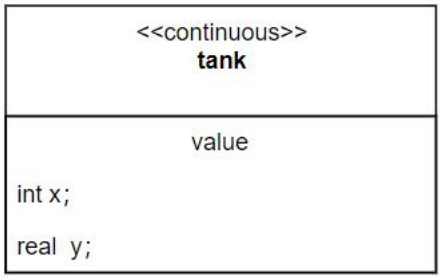
组成部分属性代表类（X 语言中一般只有 couple 类具有该属性）的内部结构。换句话说，类是由组成部分属性构成的。这种关系是一种所属关系。组成部分属性一般是由类的下方带有元类型 part 的结构分隔框表示。



图中表示一个名为 tank_system 的 couple 类模型由 1 个名为 tank1 和 1 个名为 source1 的模型组成。其中，tank1 和 source1 均是在系统某处建立的某特定类 tank 和 source 的实例化。

5.3.3.3 值属性

值属性代表类的状态变量，状态变量的类型可以是整型、实数型、布尔型等。值属性一般是由类的下方带有元类型 value 的结构分隔框表示。

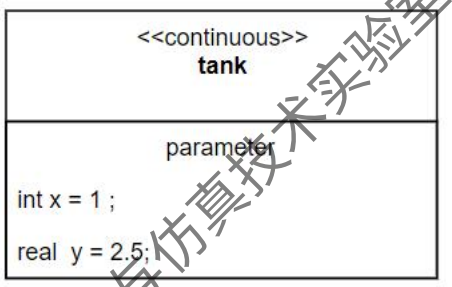


图中表示一个名为 tank 的 continuous 类模型具有 1 个名为 x 的整型变量和 1 个名为 y 的实数型变量。

5.3.3.4参数属性

参数属性代表类的实例化参数，其类型也可以是整型、实数型、布尔型等。

参数属性一般是由类的下方带有元类型 parameter 的结构分隔框表示。



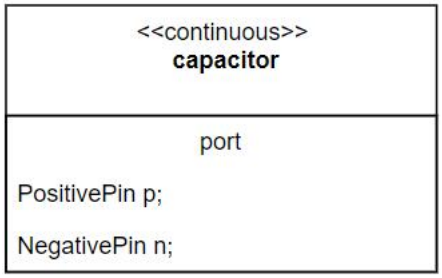
图中表示一个名为 tank 的 continuous 类模型具有 1 个名为 x 值为 1 和 1 个名为 y 值为 2.5 的实例化参数。

5.3.3.5端口

端口是代表结构边缘不同交互点的一种属性，通过那些点外部实体可以和那个结构交互（一般是交换事件、能量、数据等）。

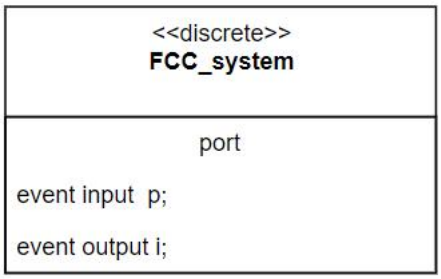
X 语言中有两种端口，一种是基于连接器定义的遵循广义基尔霍夫定律的端口，一种是基于事件交换的事件端口。

端口一般是由类的下方带有元类型 port 的结构分隔框表示。



图中表示的是一个名为 capacitor 的连续类模型具有两个分别有 PositivePin 和

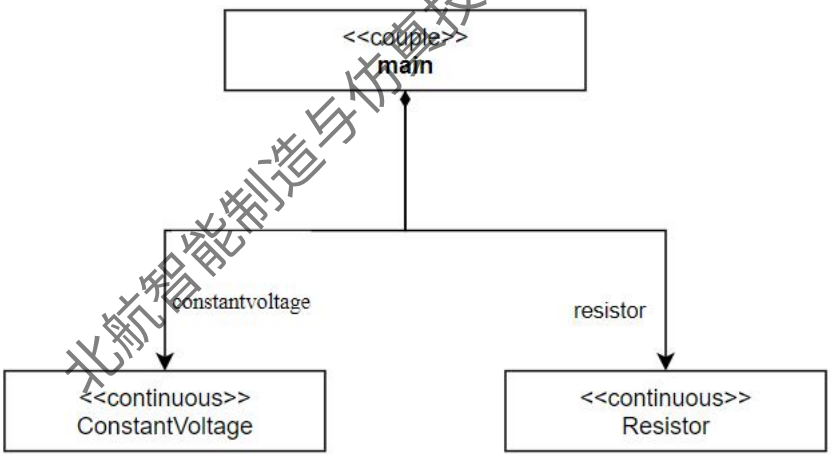
NegativePin 实例化的连接器类型端口 p 和 n。



图中表示的是一个名为 FCC_system 的离散类模型具有两个名为 p 和 i 的事件输入和事件输出端口。

5.3.3.6组合关联

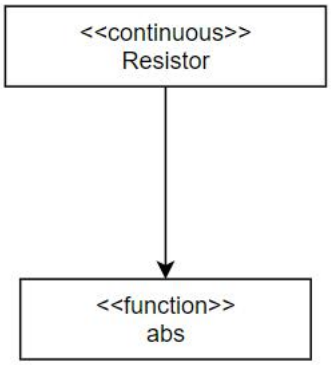
组合关联是组成部分属性的另一种表示法。两个类之间的组合关联表示结构上的分解。组合端的类实例由一些组成部分端类的实例组合而成。定义图中组合关联的标识法是两个模块之间的实线，在组合端有实心的菱形。



图中表示一个名为 main 的耦合类模型由一个名为 ConstantVoltage 的连续类模的实例化模型 constantvoltage 和一个名为 Resistor 的连续类的实例化模型 resistor 组成（注：耦合类的组合关联也包含了子类的引用）。

5.3.3.7引用关联

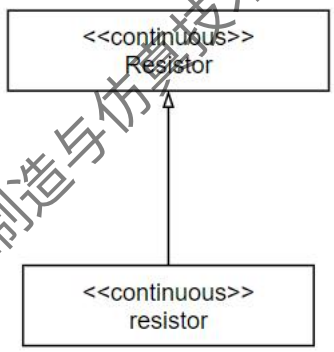
引用关联是类之间存在的一种连接关系，表示两个类之间的调用关系。定义图中引用关联的标识法是两个模块之间的实线箭头。



图中表示一个名为 Resistor 的连续类模型调用了一个名为 abs 的函数类模型

5.3.3.8泛化

泛化是经常会在定义图中显示的另一种关系。这种关系表示两种元素之间的继承关系：一个更加一般化的元素，叫做超类型，以及一个更具体的元素，叫做子类型。泛化的标识法是一条实线，在超类型的一端带有空心的三角箭头。



图中表示名为 resistor 的连续类模型是名为 Resistor 的连续类模型的泛化（继承了其所有的属性）

5.4 连接图

5.4.1 目的

创建连接图是为了指定单个类（一般是耦合类）的内部结构。连接图会表达类内部的组成部分及其是如何交互才能够创建有效实例的。

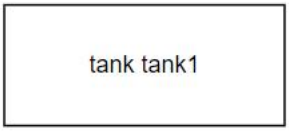
5.4.2 何时创建连接图

当我们需要建立复合系统（耦合类）的内部组成部分及其交互逻辑的时候，会创建

连接图。连接图一般会和定义图组合构建耦合类去描述一个复合系统。

5.4.3 组成部分属性

连接图的组成部分属性等同于对应定义图中组成部分分隔框中的组成部分属性。连接图中的组成部分的标识法是带有实现边框的矩形。显示在矩形边框中的字符串的格式和定义图中的组成部分分隔框中显示的字符串一致。

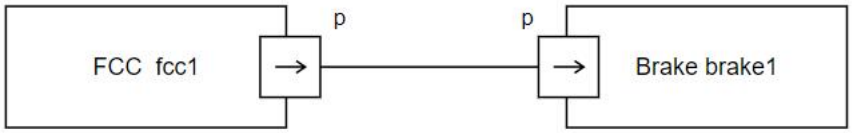


图中表示的是 2.3.3.2 中 tank_system 的一个组成部分：名为 tank1 的 tank 类的实例化模型。

5.4.4 连接器

连接图的连接器等同于对应定义图中端口分隔框中的端口属性。连接图中的连接器的标识法有两种：第一种是附着于对应组成部分上的小矩形框；另一种是附着于对应组成部分上的带有方向箭头小矩形框。

图中表示的是 source1 的连接器类的实例化端口 p 与 tank1 的连接器类的实例化端口 p 有能量或者数据的交互。



图中表示的是 fcc1 的事件输出端口 p 与 brake1 的事件输入端口 p 有事件的交互。

5.5 方程图

5.5.1 目的

方程图是一种用于描述连续系统的连续行为的图。一般基于微分方程组来描述连续系统的随时间连续变化的行为约束。

5.5.2 何时创建方程图

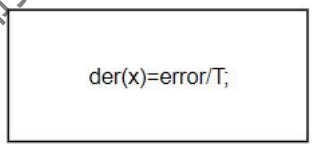
当我们所建立涉及到具有随时间变化的连续行为且物理特性明确的系统时，会创建方程图。方程图一般会和定义图组合构建连续类去描述一个连续系统。

5.5.3 方程类型

X 语言的方程类型一共包括基础方程、初始方程、条件方程、事件方程以及循环方程五种。下面将逐一介绍。

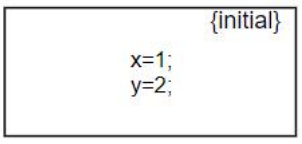
5.5.3.1 基础方程

基础方程一般都是微分代数方程（组）的形式。



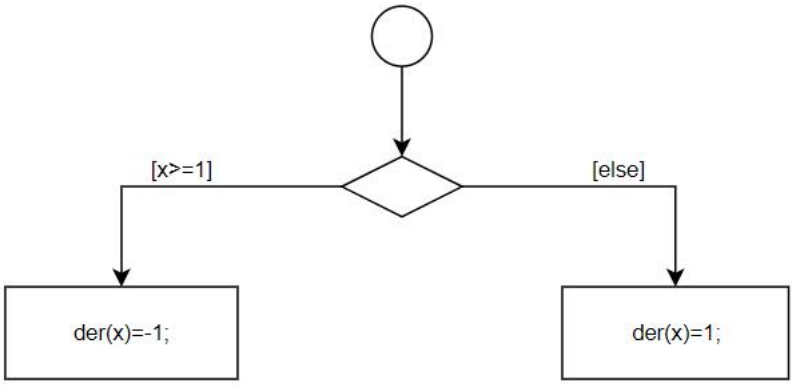
5.5.3.2 初始方程

初始方程一般都是代数方程的形式。其图形建模右上方带有{initial}标志。



5.5.3.3 条件方程

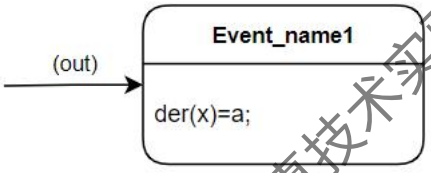
条件方程是带有 if-else 判断的微分代数方程形式。二分支的描述如下；（多分支描述同理）



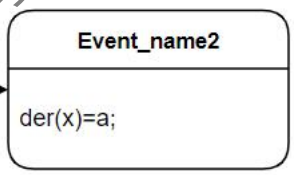
5.5.3.4事件方程

事件方程有两种形式：外部事件以及内部事件触发的两种行为方程。

外部事件：

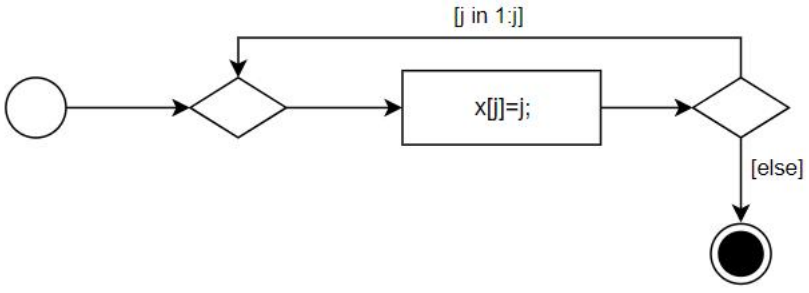


内部事件：



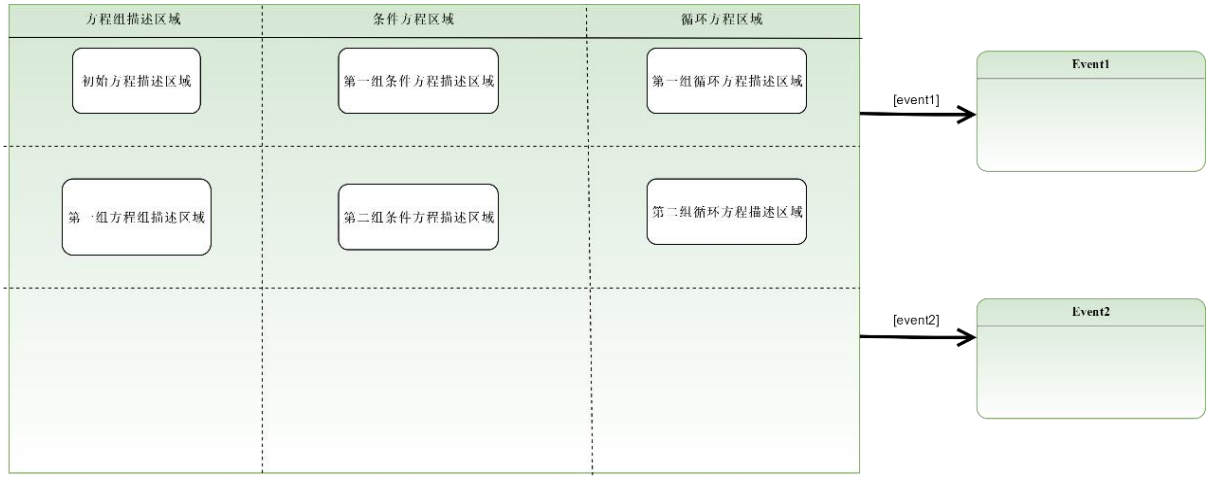
5.5.3.5循环方程

循环方程是一种方便描述具有相同描述形式的微分代数方程的描述形式。



5.5.3.6方程图架构

当所要描述的连续系统的行为由上述多种描述形式的方程组合描述时，以下图的架构进行其方程图的构建。



5.6 状态机图

5.6.1 目的

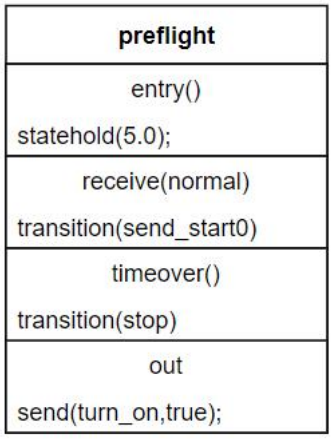
状态机图是一种用于描述离散系统的离散行为的图。一般基于事件推进系统状态的转换机制来描述离散系统的由事件触发的离散变化的行为约束。

5.6.2 何时创建状态机图

当我们所建立涉及到由事件触发的离散行为的系统时，会创建状态机图。状态机图一般会和定义图组合构建离散类去描述一个离散系统。

5.6.3 状态

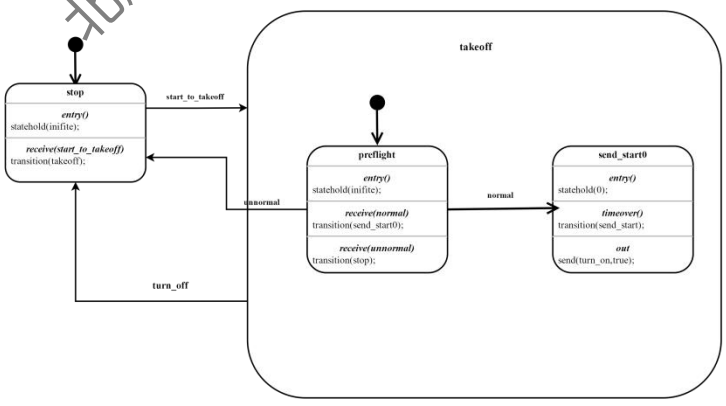
一个系统有时会拥有一系列定义好的状态，在系统操作过程中可以处于那些状态。一个正常的状态至少会包括处于状态的持续时间、接受外部事件以及内部事件触发的行为和输出。



图中表示系统在 preflight 状态在没有任何事件触发时持续时间为 5s，当接受到名为 normal 事件时，会转移到名为 send_start0 的状态；当内部事件触发即 5s 过后，会转移到名为 stop 的状态并将输出事件端口 turn_on 的值赋为 true 进行输出。

5.6.4 复合状态

复合状态一般内部会包含若干子状态，当复合状态处于非活动状态时，它所有的子状态都是非活动的。当复合状态是活动的，那么它的子状态中有一个会是活动的。在活动的状态下，复合状态会对事件进行响应，从一种子状态转移到另一种子状态。子状态之间的转换形式和状态之间的转换形式一样。另外，复合状态可能从其边界跳出，也可能从特定的内嵌子状态跳出。

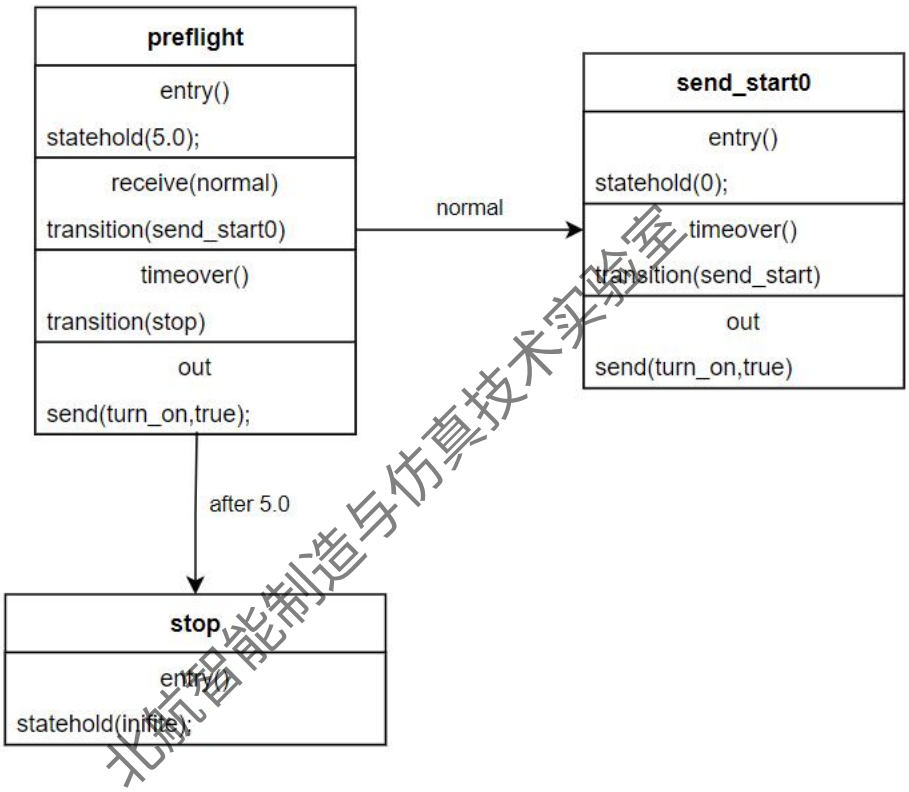


图中的 takeoff 就是一个复合状态，当其收到事件 start_to_takeoff 时，takeoff 被激活，其内部子状态标识的初始状态 preflight 被激活，当其收到事件 normal 时会正常在复合状态内部进行状态转换到 send_start0 状态，然后当其收到事件 unnormal 时会跳出复合状态 takeoff 状态转换到 stop 状态。另外，无论此时处于复合状态 takeoff 内部的任一子状

态，当收到事件 turn_off，会直接跳出当前状态到 stop 状态。

5.6.5 转换

转换代表的是从一种状态向另一种状态的改变。转换的标识法是带有开放箭头的实线，从源顶点画向目标顶点。转换的实线上会标注触发的事件（内部事件则会以 after time 的形式，外部事件会以事件名的形式）



图中的转换箭头分别表示系统处于 preflight 状态时接受外部事件 normal 转移至 send_start0 的状态；在 5s 过后会转移至 stop 状态。

5.6.6 事件类型

X 语言中的状态事件有两种：外部事件、内部事件。

外部事件

外部事件一般都是由外部系统输入的信号事件。信号事件代表能够接受信号实例的目标系统接受它的过程。如果状态机拥有具有信号事件触发器的转换，那么执行状态机的系统就必须拥有具有相同名称的接收。

内部事件

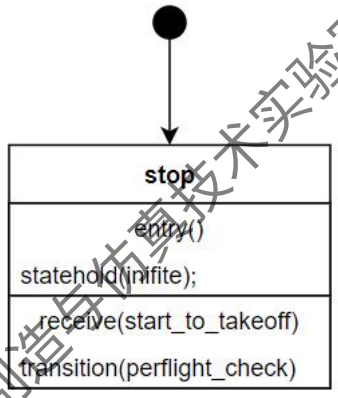
内部事件一般都是内部的时间事件。时间事件很直观，它代表时间中的实例。当那个时刻在系统操作过程中到来的时候，时间事件发生。

5.6.7 伪状态

伪状态和状态的区别是，状态机可以在状态中暂停，但无法在伪状态中暂停。之所以向状态机添加伪状态，是为了在状态之间的转换上指定控制逻辑。

5.6.7.1 初始伪状态

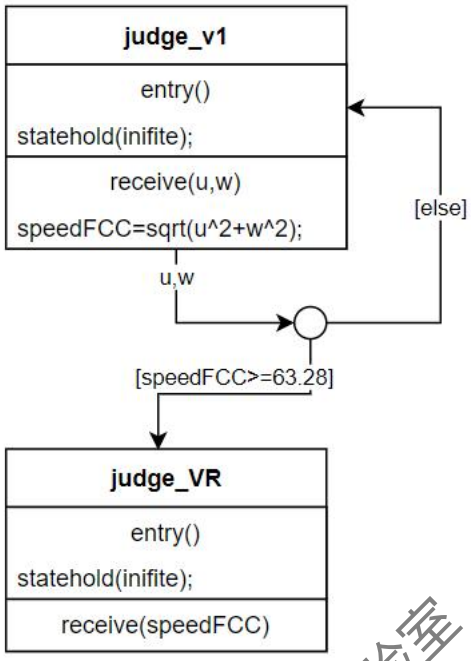
初始伪状态表示状态开始执行时的第一个状态。初始伪状态在 X 语言只是一种标识，标志着初始状态即将开始。初始伪状态的标识法是一个小型的实心圆。



图中表示系统的初始状态为 stop

5.6.7.2 连接伪状态

连接伪状态可以把状态间的多个转换组合成一个复合转换。连接伪状态的标识法也是一个小型的实心圆。和初始伪状态不同的是，连接伪状态必须拥有一个或多个输入转换，以及一个或多个输出转换。



图中表示在 judge_v1 状态接受到外部事件（u,w）时转移到连接伪状态再进行条件判断即当 speedFCC>=63.28 时，转移至状态 judge_VR，否则转移至自身状态。

5.7 活动图

5.7.1 目的

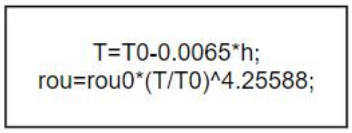
活动图是一种用于描述行为和事件发生序列的行为图。一般用于描述函数类和智能体类中 plan 的算法。

5.7.2 何时创建活动图

当我们所建立系统涉及到需调用函数或者建立智能体类中的 plan 时，会创建活动图。活动图一般会和定义图组合构建函数类去描述一个系统中调用的函数。

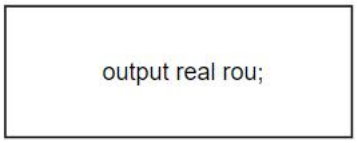
5.7.3 动作

动作是一种可以存在于活动之中的节点，它是为活动基本的功能单元建模的节点。一个动作代表某种类型的处理或转换，它会在系统操作过程中活动被执行的时候发生。动作的标识法是矩形。在 X 语言中，活动一般都是文本形式的语句描述。



5.7.4 活动参数

活动参数从总体上表示活动的一种输入或者输出。活动参数的标识法矩形，描述内容一般带有 input/output。

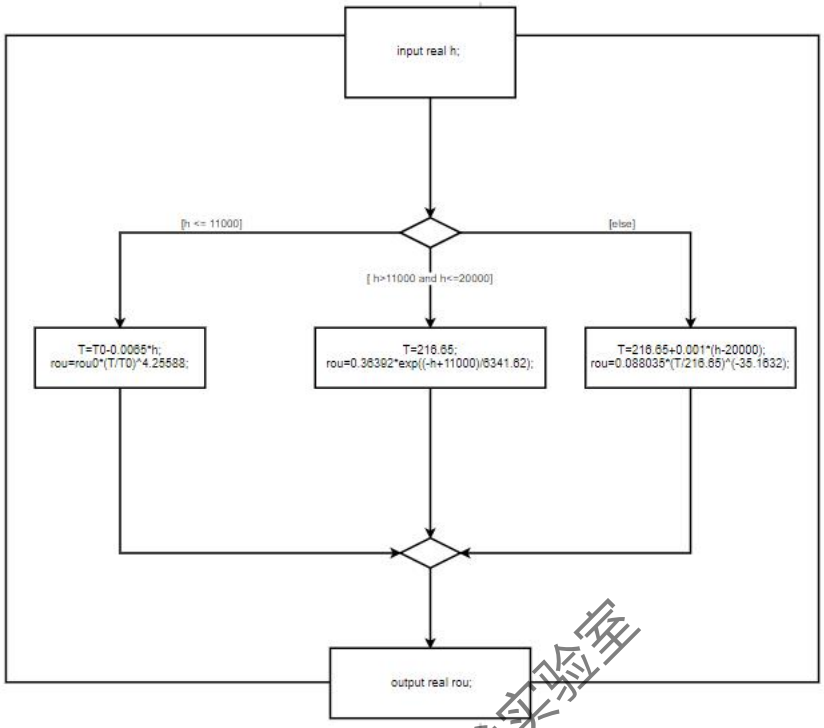


5.7.5 控制节点

使用控制节点，可以引导活动沿着路径执行。控制节点有 2 种类型：决定节点、合并节点。

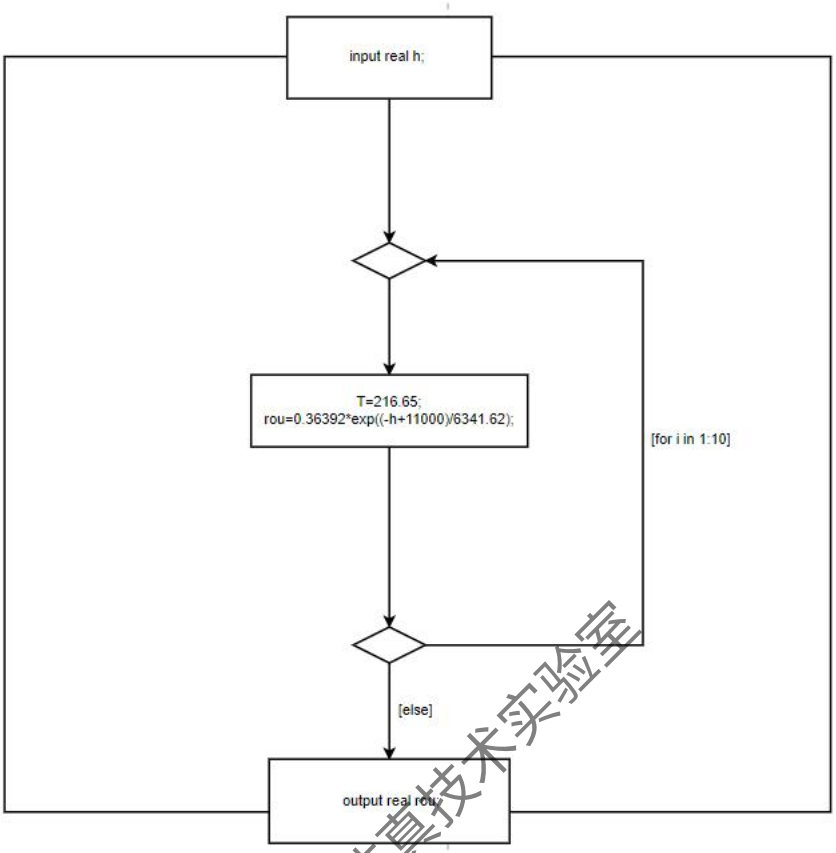
5. 7. 5. 1 决定节点

决定节点标记活动中可替换序列的开始。其标识法是一个空心的菱形。决定节点必须拥有单一的输入边，一般拥有两个或多个输出边，每个输出边会带有布尔表达式，显示为方括号中间的字符串。



5.7.5.2 合并节点

合并节点标记活动中可选序列的结尾。其标识法和决定节点相同：空心菱形。合并节点拥有两条或多条输入边，而只拥有一条输出边。一般合并节点会和决定节点组合使用，在活动中对循环建模。



5.8 模型结构图

5.8.1 目的

模型结构图是 X 语言中系统模型文件架构（文件结构）的图形表达形式。系统模型的组织方式由系统的层级关系。系统模型并没有唯一正确的结构，不同的方法会建议不同的模型结构，项目的目标不同，同一模型结构产生的效果也不同。一旦确定对系统有效的模型结构，那么创建模型结构图将会提供对该项目的一种易于理解的视图。

5.8.2 何时创建模型结构图

当项目开始时，基于对系统的以及利益相关者关注的问题，我们将会首先定义一个模型结构图用于描述系统的初始架构。然后随着设计的变化，新的架构也会被确定下来，从而在模型结构图上进行跟新补充，把更高层次的结构元素分解成更低层次的结构元素。

5.8.3 模型结构图的元素和关系

模型是模型结构图的基本元素。模型可以定义为具体的 X 语言类型，用于描述具体的某种流程或者物品；也可以是抽象的元素，用于描述具有某一特征或者性质的对象。模型在模型结构图中使用带有模型名称的类似文件夹的图形，模型名称标注在左上角的标签。

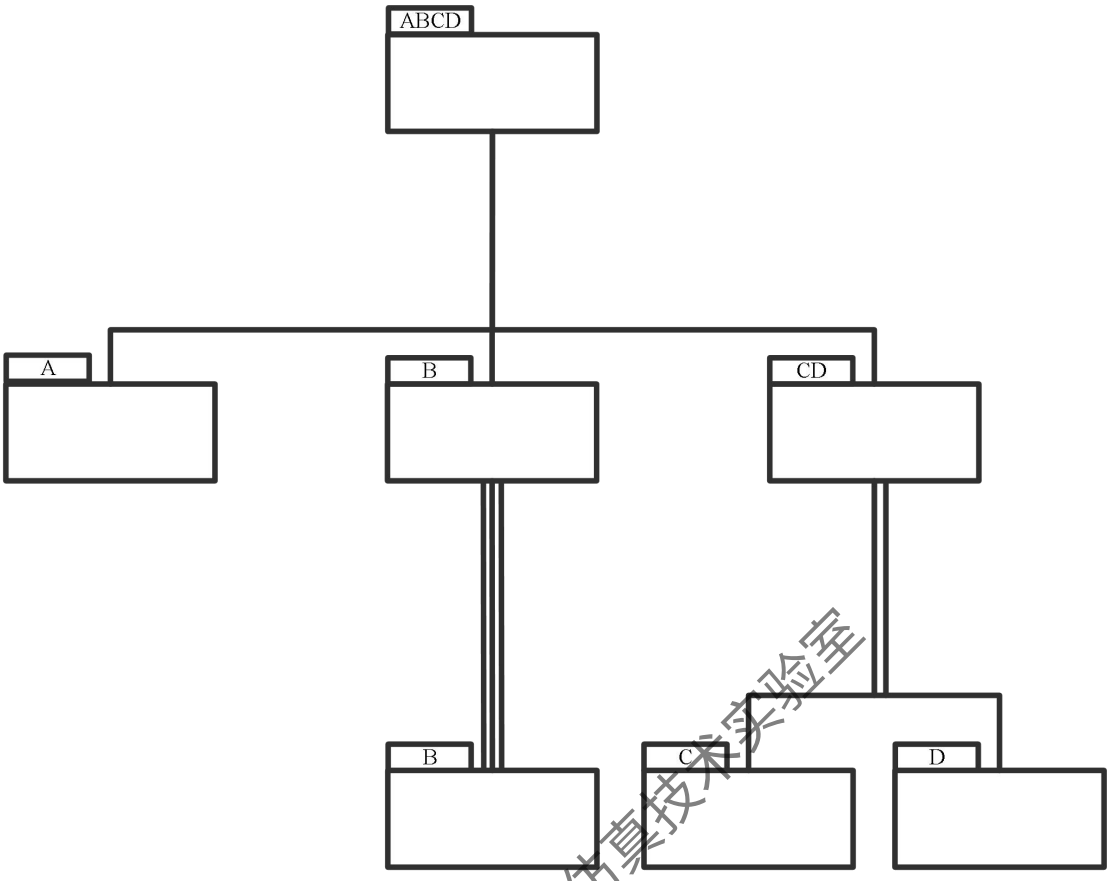
图中表示一个名为 ABCD 的实体

5.8.3.1 实体架构组合方式

1) 连线组合

模型元素在模型结构图使用三种方式进行组合连接。分别是：方面、多方面和特例化。

方面连接表达后续模型描述的是一个模型的某一个方面，即描述上一个模型节点由后续的模型节点组合而成，用 | 表示。类似地，多方面连接表示其前一个节点由相同类型的多个模型组成，用 ||| 表示，模型的数量可以在需要进行系统实例化的时候进行确定。特例化连接由一条双垂直线连接到一个模型，用 ‖ 表示，该连接描述模型的特殊化，即一个模型的不同实现方式，一个模型可能具备多种特例化，在进行系统实例化的时候也还会确定出唯一的特化。三种组合方式如下图所示。

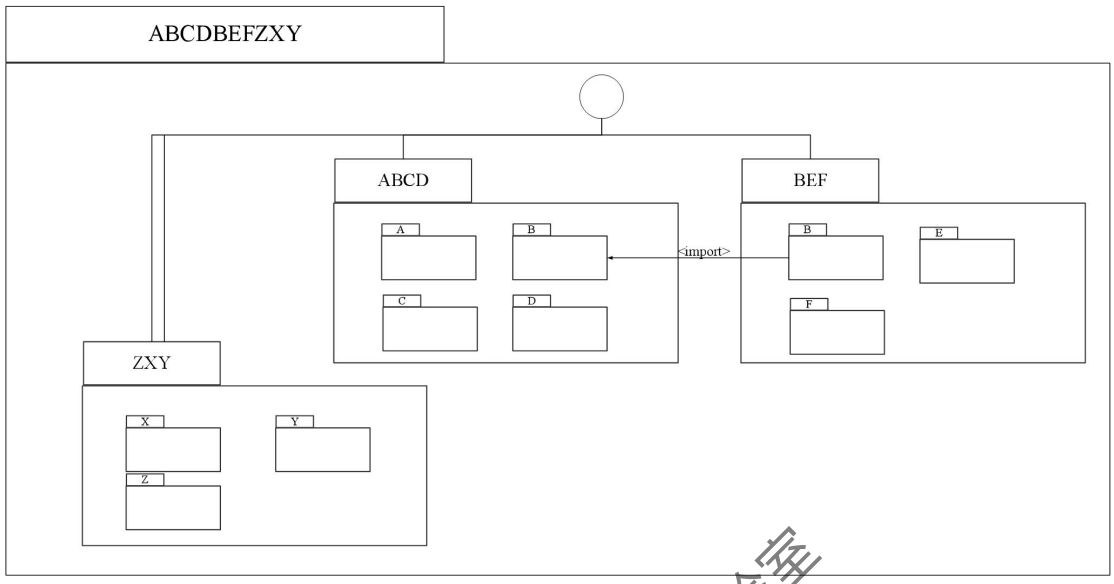


图中描述了三种类型的连接方式

图中的 ABCD 实体包括 A、B、CD 三个方面，其中 B 可以包括很多个，CD 方面存在 C 和 D 两种特殊的描述。

2) 图形内嵌组合方式

除了直接连接方式之外，X 语言模型结构图也提供了模型的内嵌方式，方便建模人员进行模型文件管理。在该描述方式中，如果一个模型内定义了其他模型，那么这些模型都是这个上层模型的方面或者是特化，具体的关系则由连接线表示，只不过连接线从直接连接变成了连接该模型的本体，使用圆形图标表示。如下图的 ABCD、BEF 是 ABCDBEFZXY 的两个方面，然后 ZXY 是 ABCDBEFZXY 一个特化。特别的，如果一个模型内的子模型没有任何连接线，则表示这些子模型都只为该模型的方面。如 A、B、C、D 都为 ABCD 的方面。

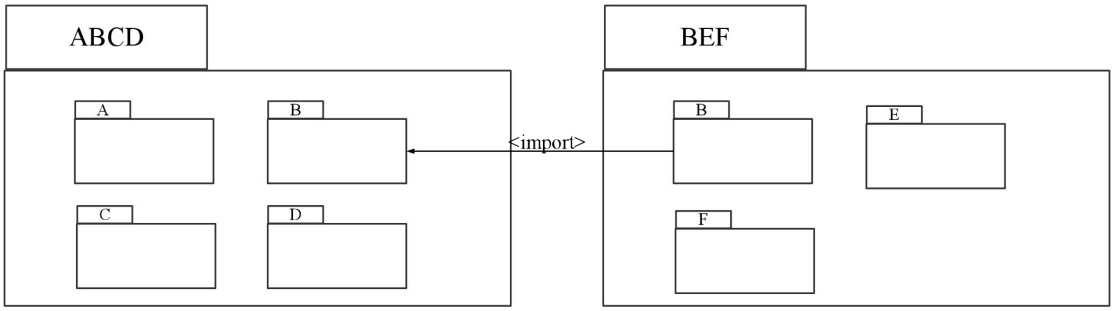


5.8.4 命名空间

一般来说，实体是其显示在内容区域的所有子实体的所有者（包含者）一一除非在图上显式地显示了另一个命名空间的包含关系。X 语言支持使用限定名称标识法来对模型所处的命名空间进行描述。如字符串 ABCD.D，。被命名的元素 D 位于字符串的末尾。字符串中的“.”表示 D 包包含在实体 ABCD 中。

5.8.5 模型引用

因为创建模型层级关系的时候，可能会从其他模型和包引人包（及其内容）到其他系统中。X 语言架构图提供了一种机制，来表达一个包引人了另一个包的内容。这种引入关系，标识法是带有开放箭头的实线，并且在线的附近带有关键字<import>，如下图所示，ABCD 从 BEF 中引入了 B 实体。



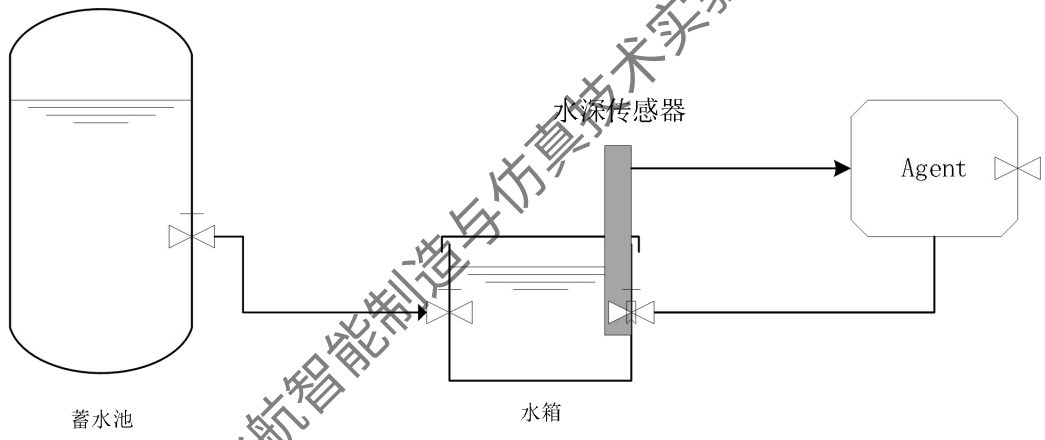
6 X 语言建模案例分析

6.1 水箱模型

6.1.1 背景描述

模型由三个部分构成，分别是水源，水箱和智能体（PI 控制器）。水箱获得水源，然后由 PI 控制器控制水位稳定在一定范围。

蓄水池模型在时间小于 150s 的时候输出水流量较小，在 150s 之后输出增大为三倍，所以可以看到水池水位在经过初始的稳定后在 150s 左右又再次增加并再次由 PI 控制器控制稳定。

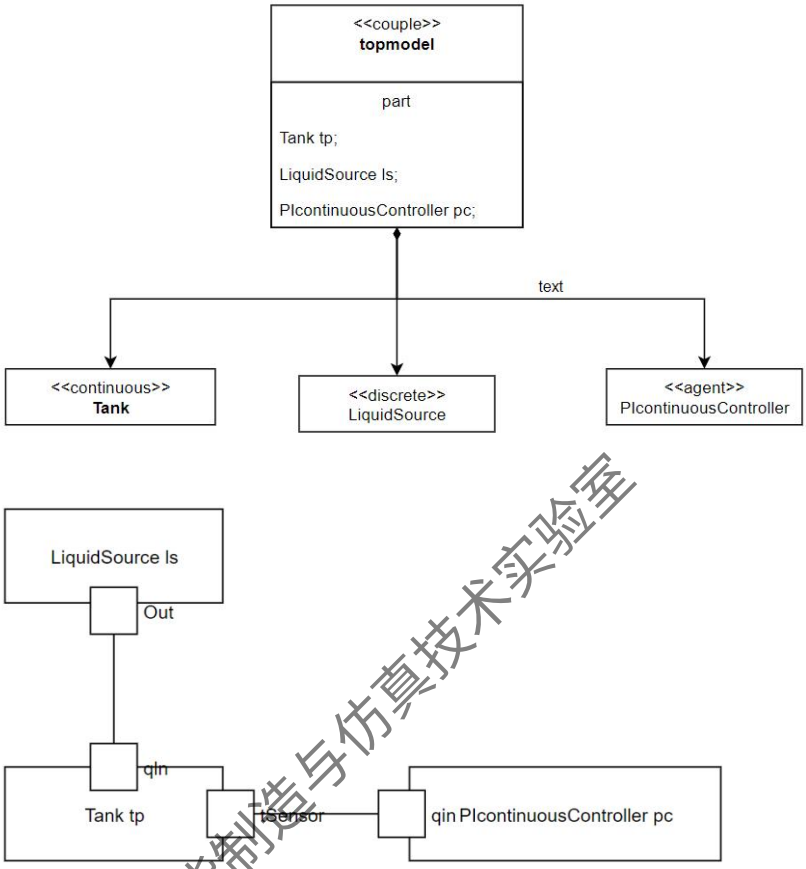


6.1.2 建模分析

首先建立顶层系统模型 topmodel，描述整个模型系统所包含的所有子系统模型，即水箱 tank，水源 LiquidSource 和智能体控制器 PIcontinuousController 模型。同时在顶层模型中还应描述各子系统模型的连接关系。

对于子系统模型，水箱模型中主要描述水箱容积等一些具体参数和水流流入流出的一些方程式，水源模型主要描述了蓄水池在各种输出水流量之间的状态转换，智能体模型描述了水位控制的相关方程。

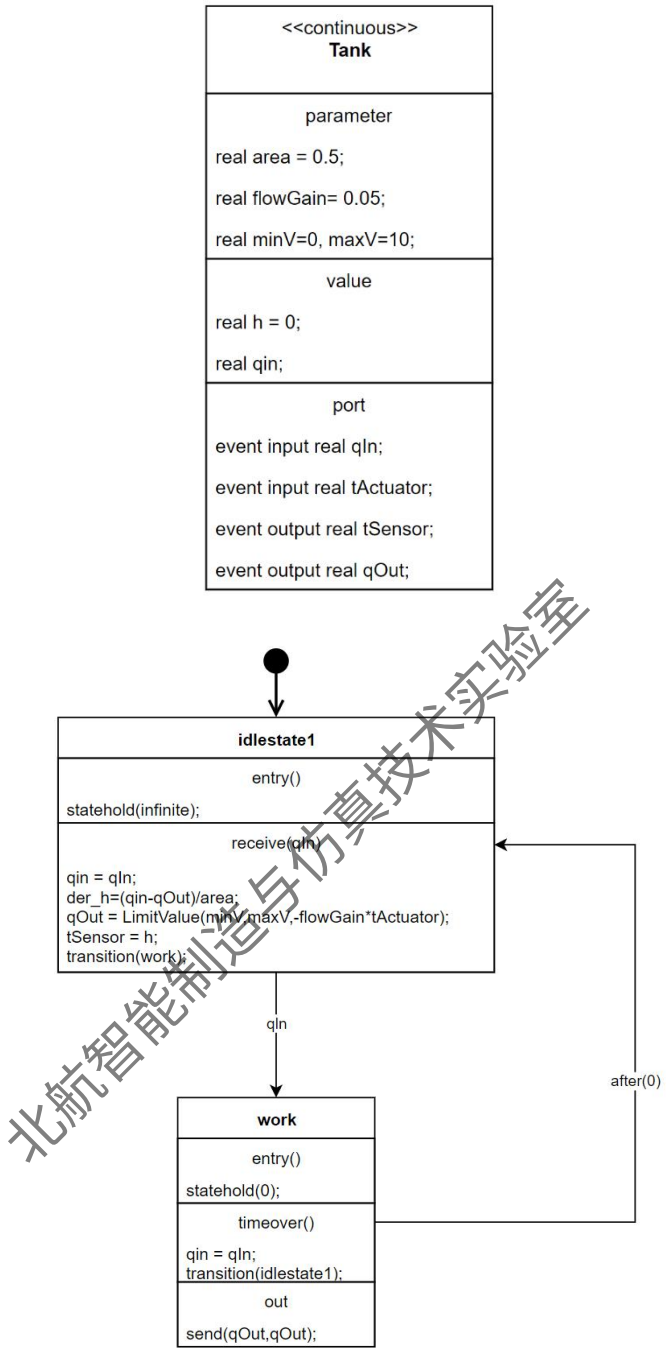
6.1.3 系统模型




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

6.1.4 子系统模型

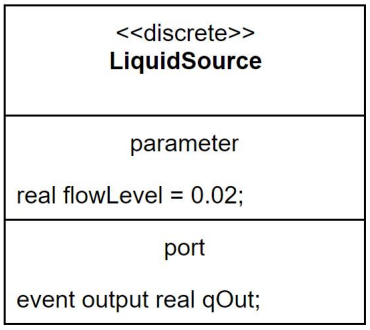
水箱模型 Tank：

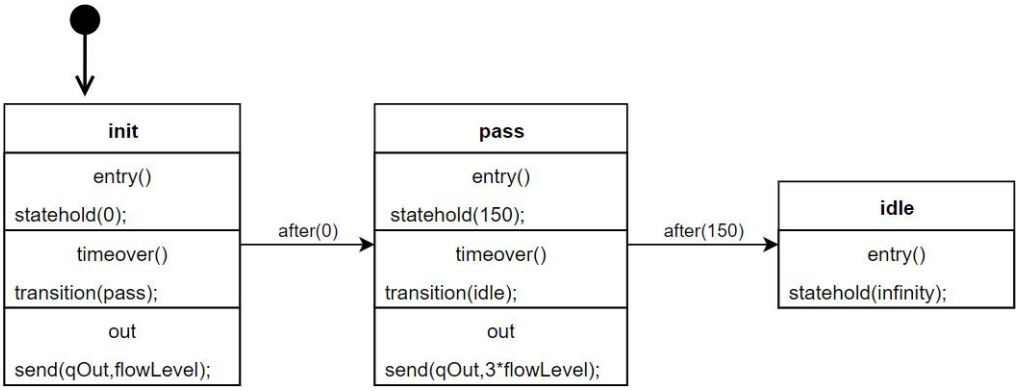



```
continuous Tank
    import LimitValue;
    parameter:
        real area = 0.5;
        real flowGain= 0.05;
        real minV=0, maxV=10;
    value:
        real h = 0;
        real qin;
    port:
```

```
        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;
```

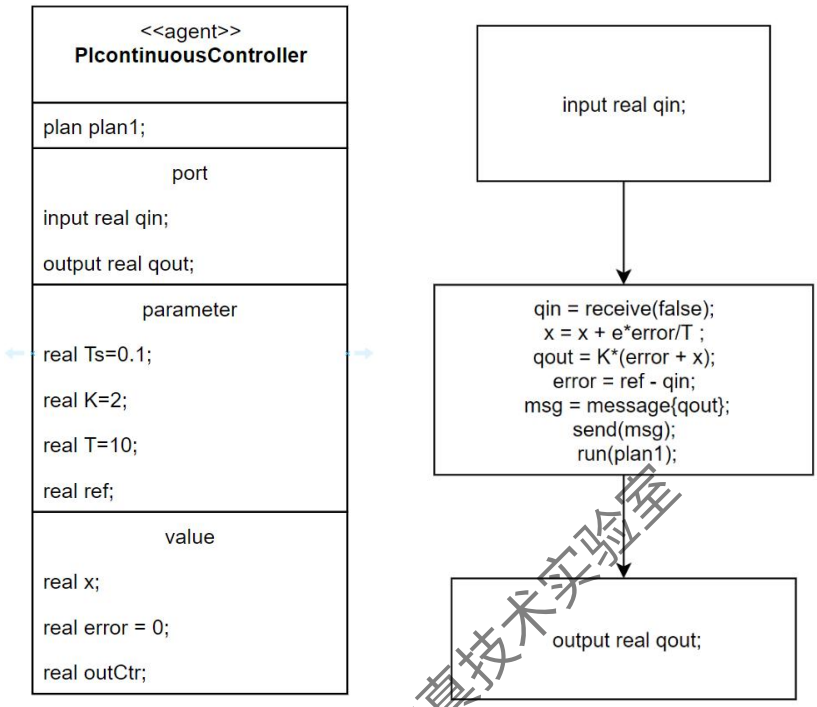

水源模型 LiquidSource：






```
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;
```

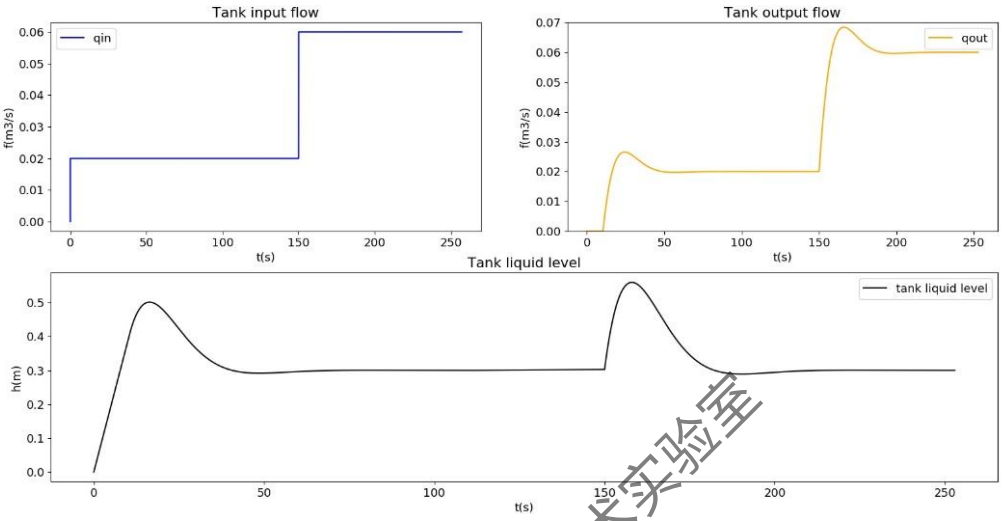
智能体模型 PIcontinuousController：




```
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 仿真结果



6.2 看门狗模型

6.2.1 背景描述

看门狗系统用于提高程序的抗干扰能力。若单片机在特殊干扰下出现程序“跑飞”时，看门狗程序可实现单片机的自动复位，避免程序陷入死循环。看门狗系统常与单片机的输入/输出引脚相连，程序控制的信号通过此引脚控制看门狗系统的状态变化。若系统陷入死循环，看门狗电路接收信号出现异常，将会触发报警机制进行系统程序自动复位。此过程状态转移情况如图2.1所示。

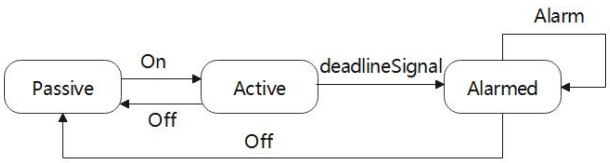


图 2.1 看门狗系统状态转移情况

看门狗系统包括看门狗电路以及与之相连接的三个接口端，控制三路信号的输入，控制系统在不同状态之间进行转移。它有两种基本状态，被动和主动，但也可以进入第三种状态，在收到deadline事件时发出警报。当进入第三种状态时，看门狗立即发出警

报。

看门狗系统包括看门狗模型本身以及三个事件发生器，如图2.2所示。

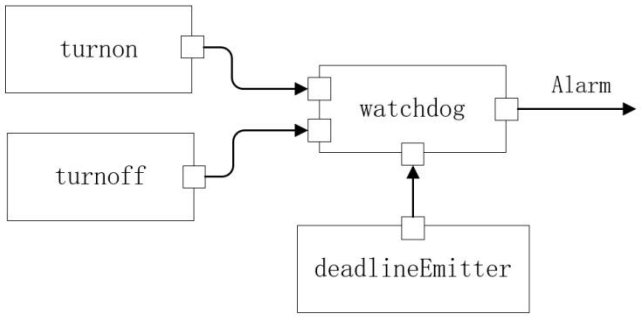
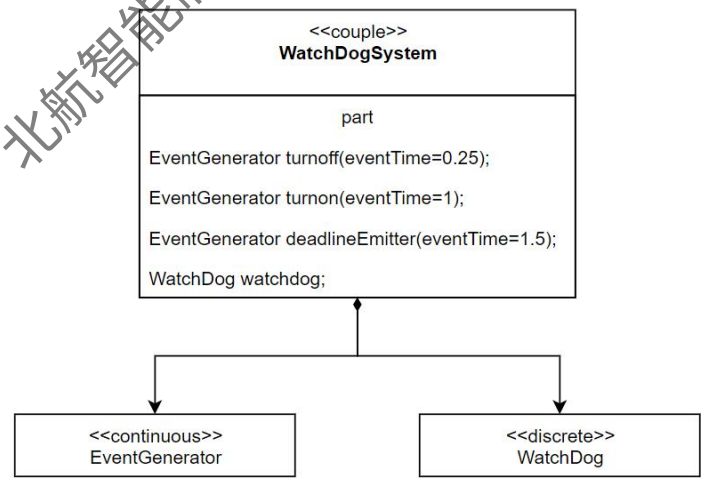


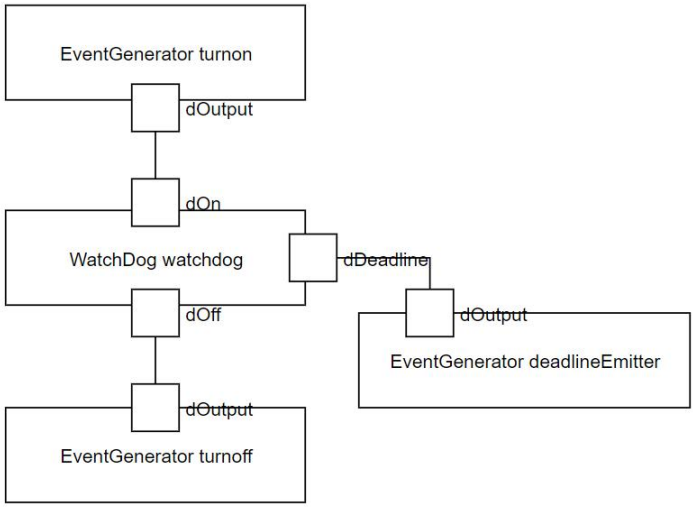
图 2.2 看门狗系统模型

6.2.2 建模分析

首先建立顶层系统模型 WatchDogSystem，用耦合类将离散模块和连续模块整合在一起，并在此顶层模型中表述各子系统模型之间的关系。

6.2.3 系统模型

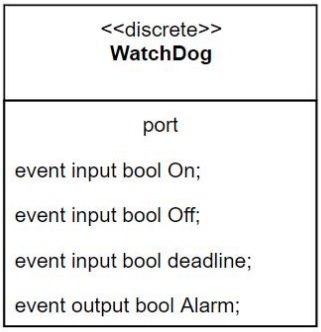


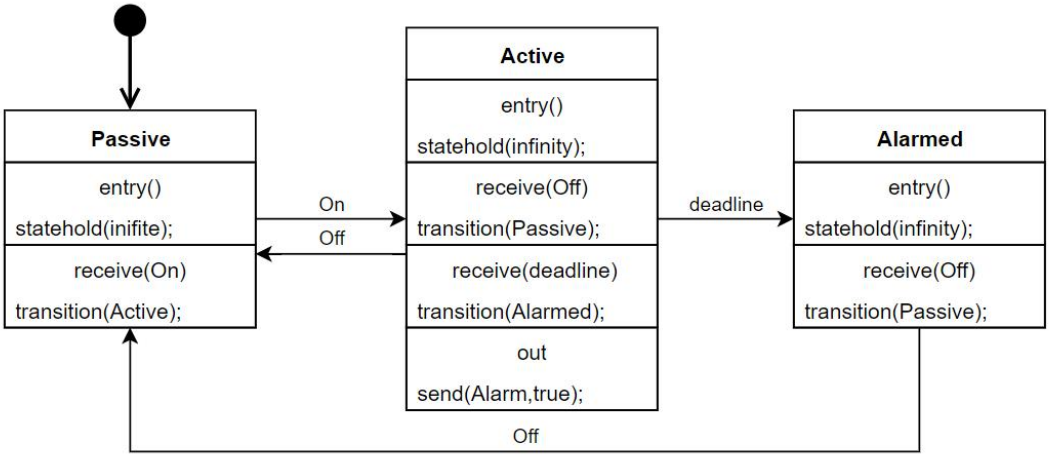



```
couple WatchDogSystem
    import EventGenerator;
    import WatchDog;
part:
    EventGenerator turnoff(eventTime=0.25);
    EventGenerator turnon(eventTime=1);
    EventGenerator deadlineEmitter(eventTime=1.5);
    WatchDog watchdog;
connection:
    connect(turnon.dOutput, watchdog.dOn);
    connect(turnoff.dOutput,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


6.2.4 子系统模型

WatchDog：

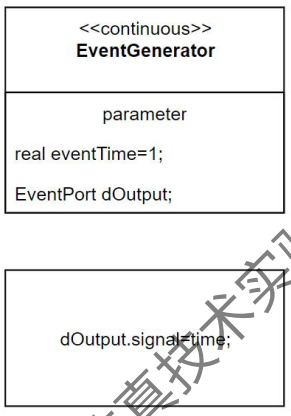





```
discrete WatchDog
port:
    event input bool On;
    event input bool Off;
    event input bool deadline;
    event output bool Alarm;
state:
initial state Passive
    when entry() then
        statehold(inifite);
    end;
    when receive(On) then
        transition(Active);
    end;
end;
state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

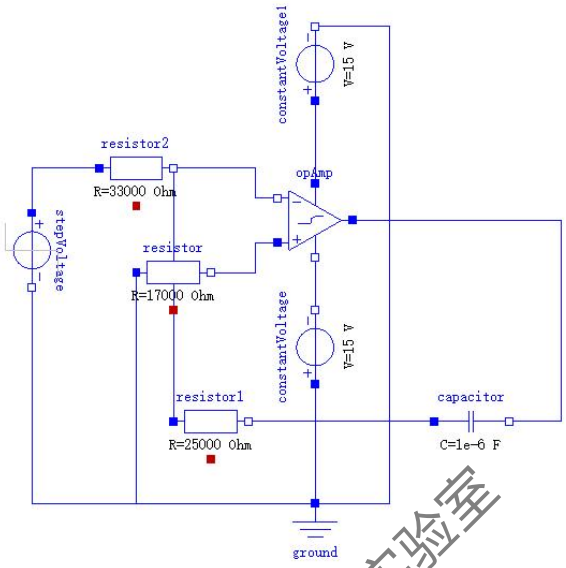

6.2.5 仿真结果

6.3 电路模型

6.3.1 背景描述

对如下电路模型进行建模仿真。本案例为全局连续类案例。主要由 continuous 对各

个模块进行建模。



建模思路为：主要利用 continuous 对各个电路器件进行建模，在顶层电路耦合模块中对各个电路器件类进行实例化，并连接形成最终的电路模型。

6.3.2 系统模型

```
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 子系统模型

电容模块

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

电阻模块

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

大地模块

```
continuous Ground
```

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

阶跃电压源模块

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

连续电压源模块

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


运算放大器模块

```
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;
```

接口或基准定义模块，放于名为 Interfaces 的文件中，含有以下 6 个类

正极模块

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

负极模块

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

一端口模块

```
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;
```

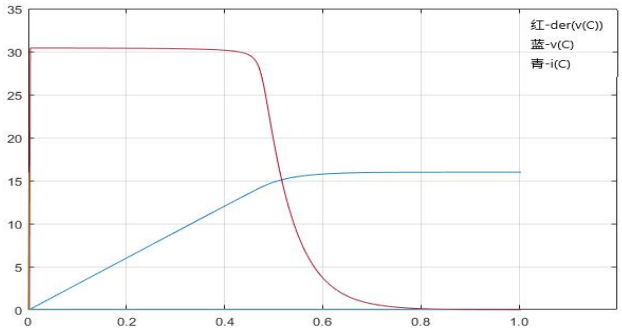
信号源模块

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

电压源模块

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

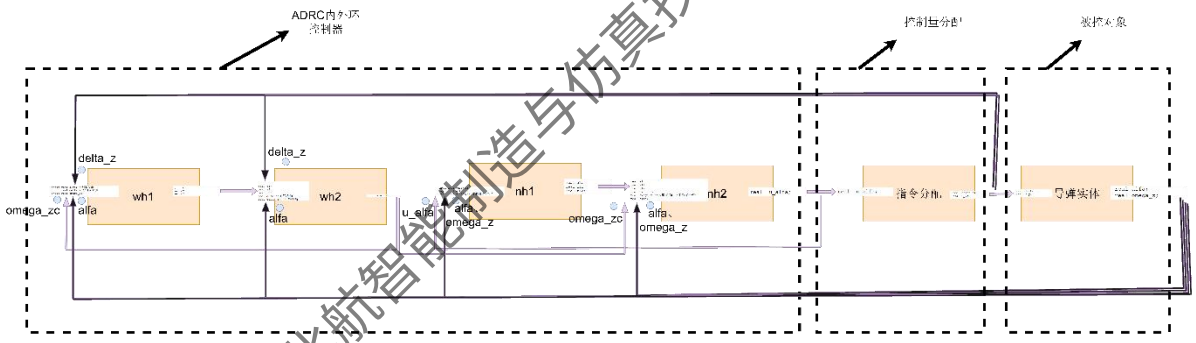
6.3.4 仿真结果



6.4 导弹姿态控制模型

6.4.1 背景描述

对如下导弹姿态控制模型进行建模仿真。本案例为全局迭代类案例，即各个模块时间的数据具有迭代传递关系。主要由 discrete 对各个模块进行建模。



ADRC 导弹姿态控制模型示意图

6.4.2 系统模型

```
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 子系统模型

内环 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;
```

内环 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 //前向
                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 //反馈
                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;
```

外环 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 待定
        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//计算一遍，然后再继续完成需要的东西
                e4 = alfa__c - alfa;//##e4 = alfa__c - z21;  求出 e4
                u__alfa = beta41 * fal(e4, a4, delta4);//求出 u_alfa

                der_alfa = u__alfa;
                alfa = alfa+time_step*der_alfa;//更新 alfa
                cs2 = -3.2553 * alfa - 0.3828;
                f__alfa = cs1 * sin(alfa) + cs2 * alfa * cos(alfa) + cs3 * delta_z *
cos(alfa);//cs1 //cs2   cs3
                //状态观测器
                der_z21 = z22 - bata31 * e3 + f__alfa + omega_zc;
                z21 = z21+time_step*der_z21;//更新 z21
                e3 = z21 - alfa;
                der_z22 = -beta32 * fal(e3, a3, delta3);
                z22 = z22+time_step*der_z22;//更新 z22
                transition(send_message);
```

```
            end;
        end;
        state work
            when entry() then
                statehold(infinite);
            end;
            when receive(alfa__c) then//计算一遍，然后再继续完成需要的东西
                e4 = alfa__c - alfa;//##e4 = alfa__c - z21;  求出 e4
                u__alfa = beta41 * fal(e4, a4, delta4);//求出 u_alfa
                der_alfa = u__alfa;
                alfa = alfa+time_step*der_alfa;//更新 alfa
                cs2 = -3.2553 * alfa - 0.3828;
                f__alfa = cs1 * sin(alfa) + cs2 * alfa * cos(alfa) + cs3 * delta_z *
cos(alfa);//cs1    //cs2    cs3
                //状态观测器
                der_z21 = z22 - bata31 * e3 + f__alfa + omega_zc;
                z21 = z21+time_step*der_z21;//更新 z21
                e3 = z21 - alfa;
                der_z22 = -beta32 * fal(e3, a3, delta3);
                z22 = z22+time_step*der_z22;//更新 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;
```

外环 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 待定
        real cs3 = -0.082;
    value:
        real e4;
        real u__alfa;
        real f__alfa;
        real cs2;
        real der_alfa;//导数
    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    //前向输入
            //误差反馈控制器
```

```
                e4 = alfa__c - alfa;//##e4 = alfa__c - z21;   alfa 不用更新
                u__alfa = beta41 * fal(e4, a4, delta4);
                der_alfa = u__alfa;
                alfa = alfa+time_step*der_alfa;//更新 alfa
                //连接点
                //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   //反馈输入
                transition(work);
            end;
            when receive(delta_z) then   //反馈输入
                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;
```

导弹实体

```
    discrete daodanshiti
        import fal;
    parameter:
        real time_step = 0.01;
        real cs11=0.3;//c_x 不知道，待定
        real cs13=-0.082;
        real cs14=-1/(255*1000);
        real cs23=-0.32;
```

```
        real cs24=1.26/306.3;////此参数涉及 d，无法确定
    value:
        real delta_z;
        real T_y;
        real cs21;
        real cs22;
        real cs12;
        //导数
        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;
                //误差反馈控制器
                der_omega_z = cs21 * alfa + cs22 * delta_z + cs23 * omega_z + cs24 *
T_y;
                omega_z = omega_z+time_step*der_omega_z;//更新 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;//更新 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;
```

指令分配

```
    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    //前向
                //误差反馈控制器
                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;
```

指令输入

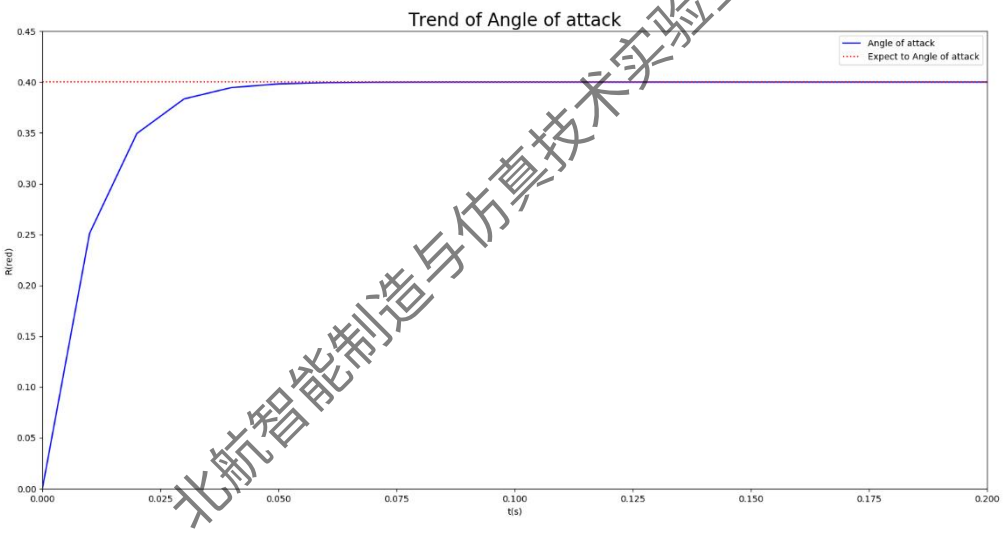
```
discrete inputstart
parameter:
    real alfa__c = 0.4;
port:
    event output real alfa__c_out;
state:
    initial state start
        when entry() then
            statehold(0);
        end;
        when timeover() then
            transition(work);
        out
            send(alfa__c_out,alfa__c);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

函数

```
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;
```

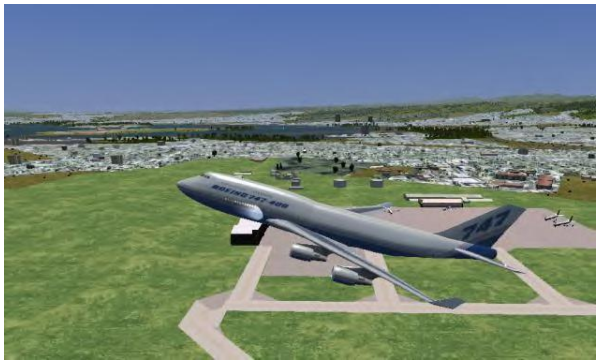
6.4.4 仿真结果



6.5 飞机起飞模型

6.5.1 背景描述

以世界范围内广泛应用的典型民机波音 747-400 为研究对象，针对 B747 具体飞行场景，即起飞场景，基于 X 语言实现从飞机起飞场景的需求分析、功能分析、系统设计乃至物理性能仿真验证，从而解决传统设计方法中需求难以追溯和早期系统功能架构设计验证困难等问题。



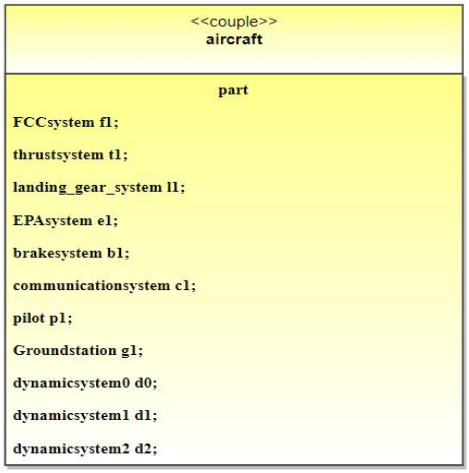
6.5.2 建模分析

基于 X 语言需求图和用例图对飞机起飞场景的需求、功能进行分析；基于 X 语言定义图、连接图构建满足起飞场景需求及功能的子系统组件及其之间的交互逻辑进行系统设计；基于 X 语言定义图、状态机图进行子系统组件的结构、行为构建。最后生成仿真文本进行功能、性能的仿真，达到验证设计的合理性的目的。

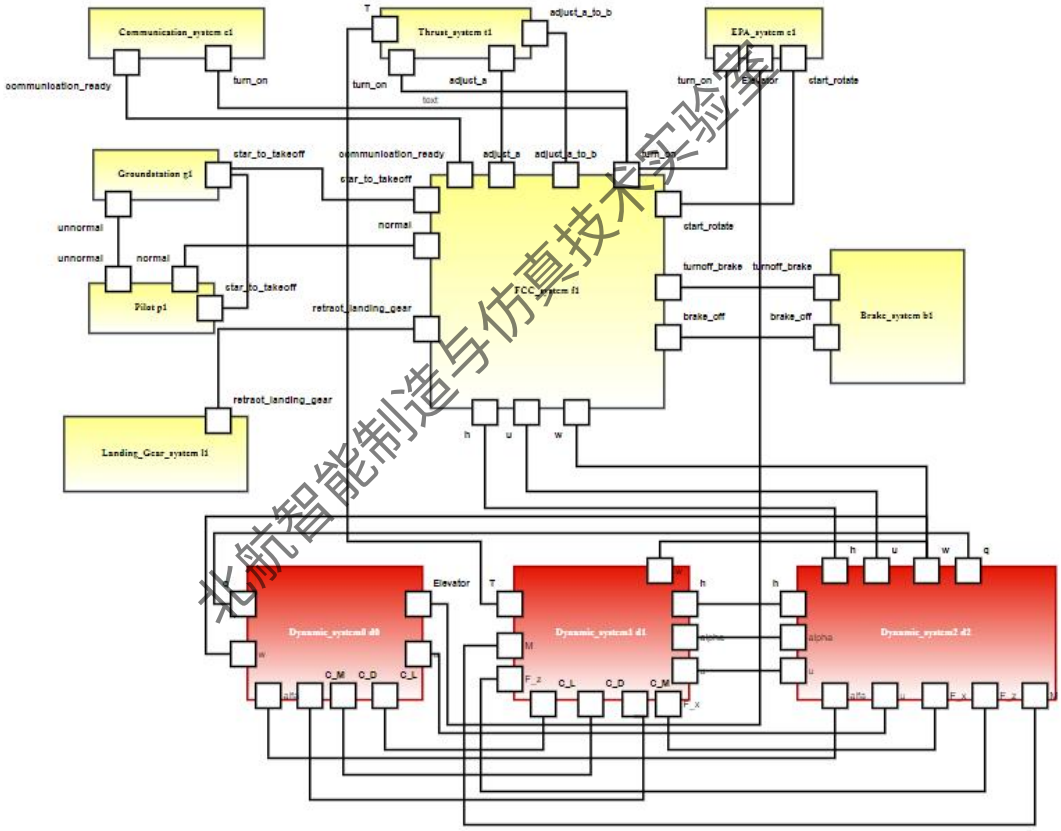
6.5.3 系统模型

基于上述功能架构分析，整个波音 747-400 起飞过程会包括多个飞机子系统。在 X 语言中，遵循自顶向下的建模理念，基于耦合类来建立整个飞行系统内部各子系统的组成及其之间的交互关系。X 语言耦合类在图形建模层面是由定义图和连接图进行描述（如下图所示）。其中，定义图中描述了整个飞行系统在起飞过程中所涉及的子系统；连接图描述了各子系统之间的信号或数据的交互关系。另外，在建立完飞行系统的图形模型之后可自动生成 X 语言的文本模型（如图所示）。

定义图：



连接图：



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;
```

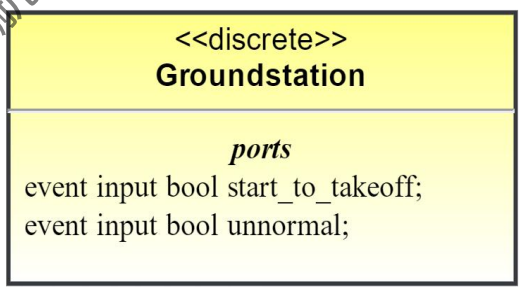
6.5.4 子系统模型

由所建立的飞行系统模型，我们定义了飞行过程中各子系统的信号或数据交互逻辑。在上述分析的基础上，基于 X 语言建立各子系统的结构和行为逻辑。

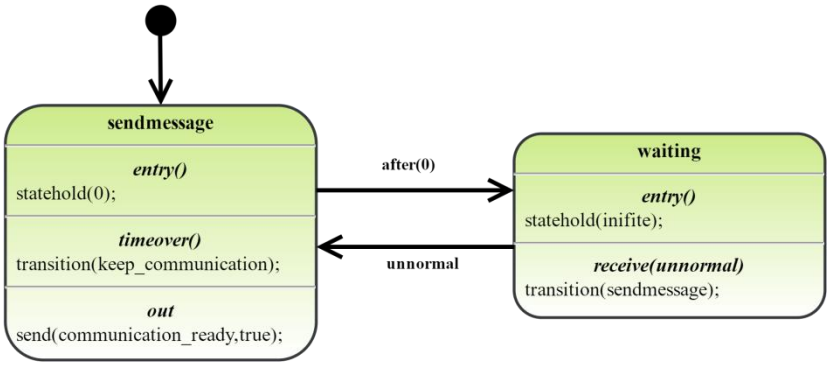
6.5.4.1 基站

基站在整个飞行过程中，与飞行员进行通讯并发送起飞指令的功能。这里，基于 X 语言的离散类来实现基站的结构和行为描述。X 语言离散类在图形建模层面是由定义图和状态机图进行描述（如下图所示）。其中，定义图中描述了基站的输入输出信号；状态机图描述了基站的状态和行为逻辑。另外，在建立完基站的图形模型之后可自动生成 X 语言的文本模型（如图所示）。

定义图：



状态机图：



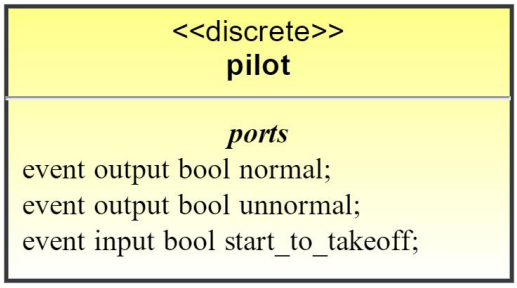

```
discrete Groundstation
    port:
        event input bool unnormal;
        event output bool start_to_takeoff;
    state:
        initial state sendmessage
            when entry() then
                statehold(0);
            end;
            when timeover() then
                transition(waiting);
            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 飞行员

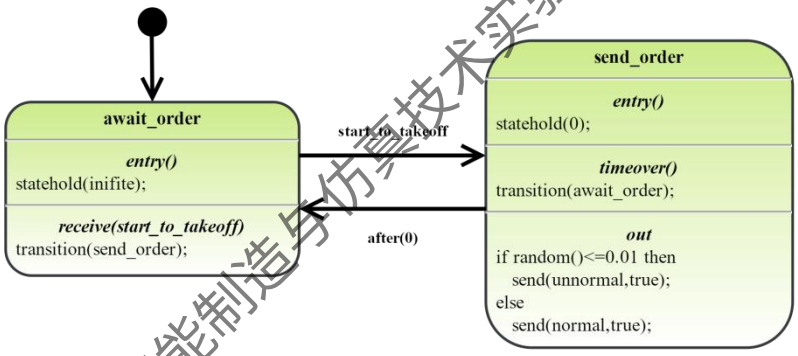
飞行员在整个过程中，在接受到基站的起飞指令后，会进行飞机检查发送是否正常的信号指令。这里，基于 X 语言的离散类来实现飞行员的结构和行为描述。X 语言离散类在图形建模层面是由定义图和状态机图进行描述（如下图所示）。其中，定义图中描

述了飞行员的输入输出信号；状态机图描述了飞行员的状态和行为逻辑。另外，在建立完飞行员的图形模型之后可自动生成 X 语言的文本模型（如图所示）

定义图：



状态机图：




```
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);
```

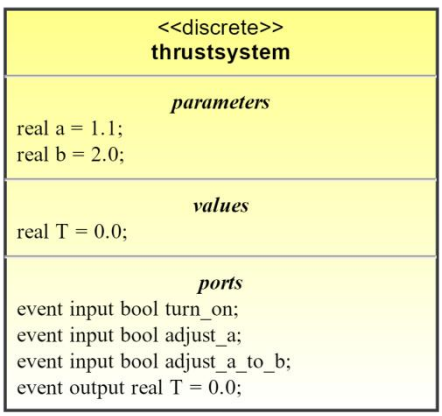
```
                end;
                when timeover() then
                    transition(await_order);
                out
                    if random()<=0.01 then
                        send(unnormal,true);
                    else
                        send(normal,true);
                 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

6.5.4.3 飞行控制计算机系统

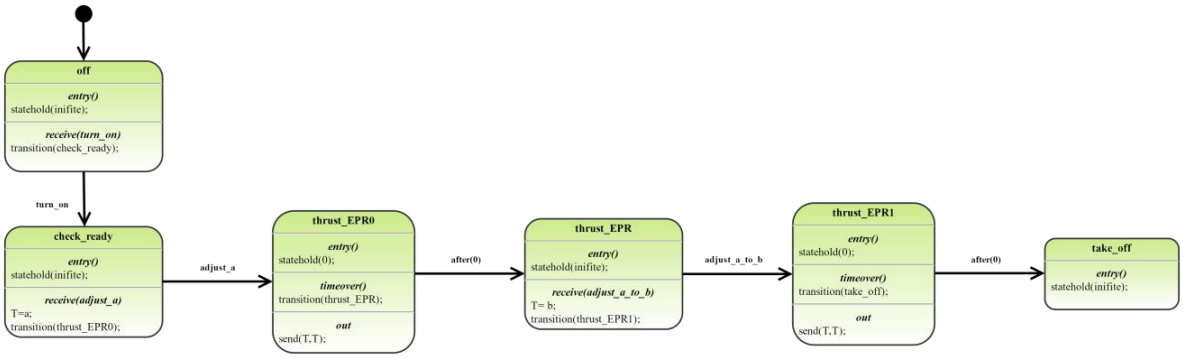
飞行控制计算机系统在整个过程中，作为主控系统，协调控制着整个起飞过程其他子系统的逻辑行为。飞控系统在接受起飞指令后，会进入预检测状态，确认无误后，开始准备起飞，在接受到通讯系统正常指令后，开始给制动系统发送指令释放制动器，在接受制动释放指令后，发送推力指令 1 给推力系统，3s 后发送推力指令 2 给推力系统，当速度大于抬轮速度 VR 时，发送升降舵调整指令给电力系统，当飞行高度大于 12 米时，发送收起起落架指令给起落架系统完成起飞过程。

这里，基于 X 语言的离散类来实现飞控系统的结构和行为描述。X 语言离散类在图形建模层面是由定义图和状态机图进行描述（如下图所示）。其中，定义图中描述了飞控系统的输入输出信号；状态机图描述了飞控系统的状态和行为逻辑。另外，在建立完飞控系统的图形模型之后可自动生成 X 语言的文本模型（如图所示）。

定义图：



状态机图：




```
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;
```

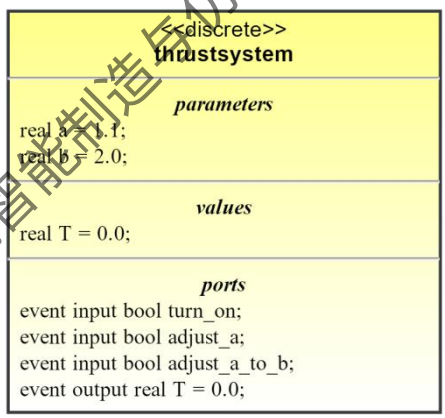
6.5.4.4推力系统

推力系统在整个过程中，在收到启动指令后，进入检查准备状态，当收到推力系统 1 指令时，将推力控制信号调为 1.1；在收到推力系统 2 指令时，将推力控制信号调为 2。

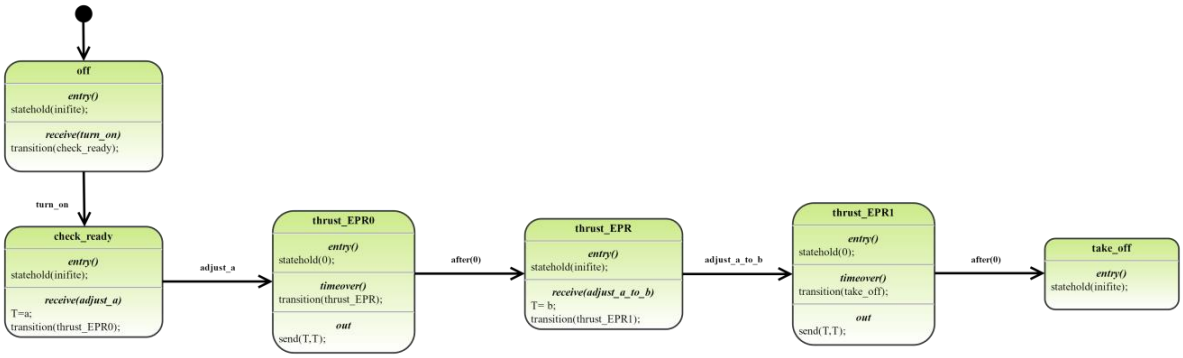
这里，基于 X 语言的离散类来实现推力系统的结构和行为描述。X 语言离散类在图形建模层面是由定义图和状态机图进行描述（如下图所示）。其中，定义图中描述了推力系统的输入输出信号；状态机图描述了推力系统的状态和行为逻辑。另外，在建立完推力系统的图形模型之后可自动生成 X 语言的文本模型（如图所示）。

图形建模

定义图：



状态机图：



文本建模

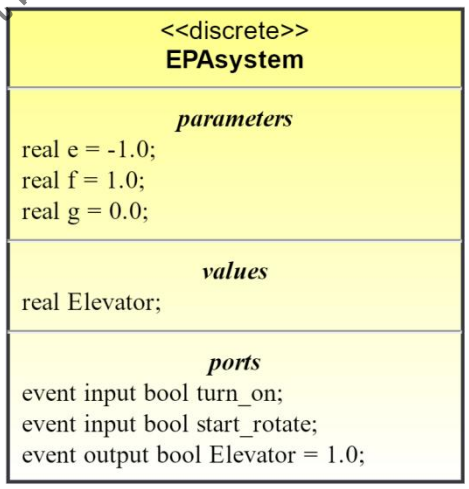
```
discrete thrustsystem
    port:
        event input bool turn_on;
        event input bool adjust_a;
        event input bool adjust_a_to_b;
        event output real T = 0.0;
    parameter:
        real a = 1.1;
        real b = 2.0;
        real c = 909000;
    value:
        real t0;
        real 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;
```

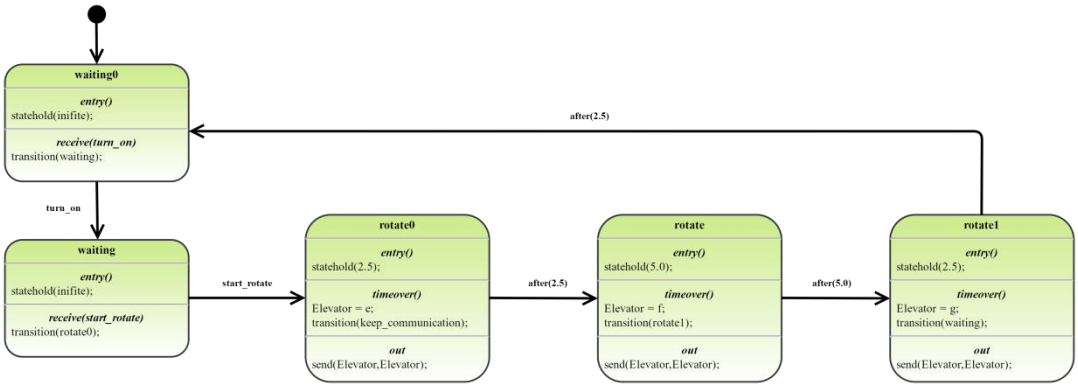
6.5.4.5电力系统

图形建模

定义图：



状态机图：



文本建模：

```
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;
```

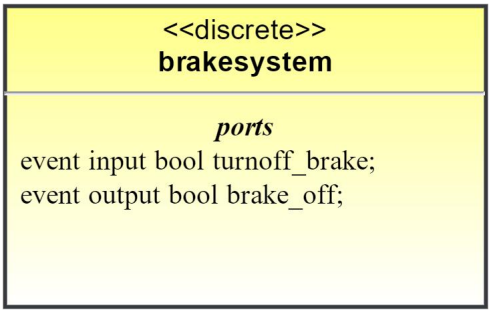
6.5.4.6起落架系统

起落架系统在整个过程中，在收到起落架收起指令后，调整起落架控制信号进行收起状态。

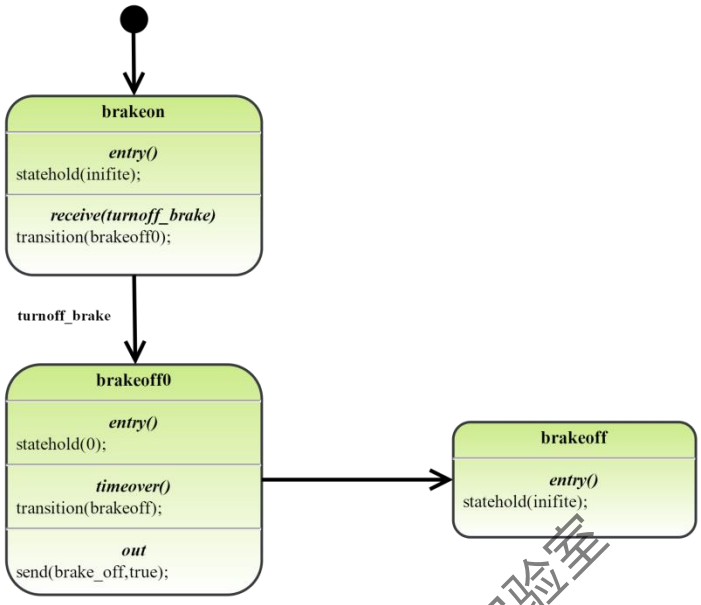
这里，基于 X 语言的离散类来实现起落架系统的结构和行为描述。X 语言离散类在图形建模层面是由定义图和状态机图进行描述（如下图所示）。其中，定义图中描述了起落架系统的输入输出信号；状态机图描述了起落架系统统的状态和行为逻辑。另外，在建立完起落架系统的图形模型之后可自动生成 X 语言的文本模型（如图所示）。

图形建模

定义图：



状态机图：



文本建模：

```
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;
```

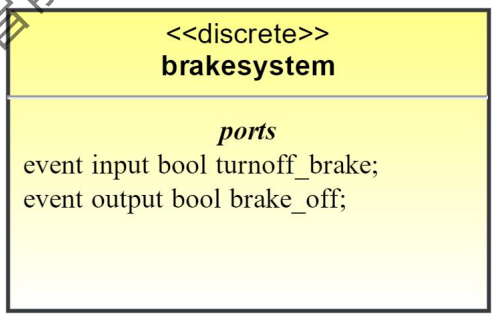
6.5.4.7制动系统

制动系统在整个过程中，在收到制动释放指令后，释放制动并发送制动已释放指令调整到制动释放状态。

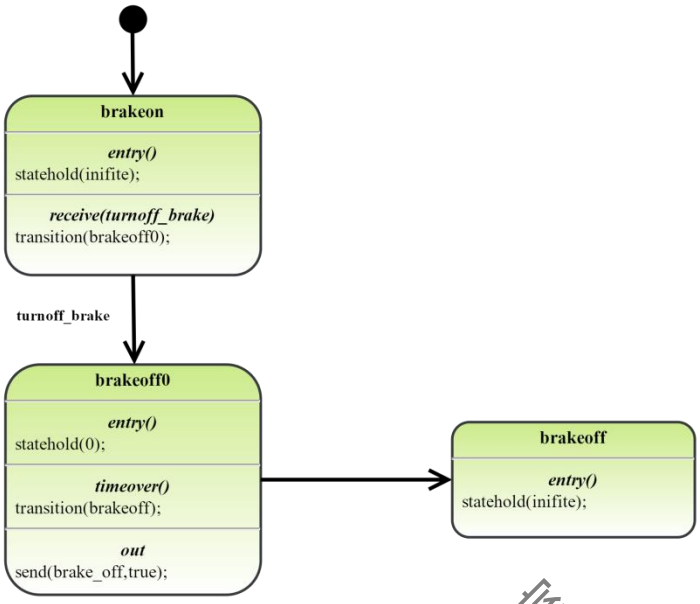
这里，基于 X 语言的离散类来实现制动架系统的结构和行为描述。X 语言离散类在图形建模层面是由定义图和状态机图进行描述（如下图所示）。其中，定义图中描述了制动系统的输入输出信号；状态机图描述了制动系统的状态和行为逻辑。另外，在建立完制动系统的图形模型之后可自动生成 X 语言的文本模型（如图所示）。

图形建模

定义图：



状态机图：



文本建模

```
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;
```

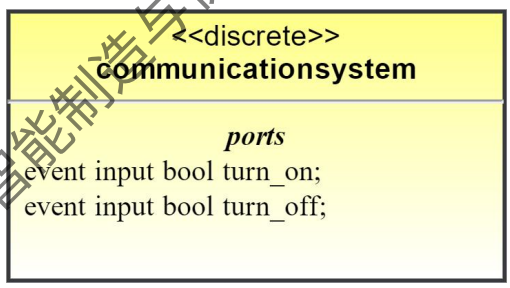
6.5.4.8通信系统

通信系统在整个过程中，在收到启动指令后，3s 后，发送通信正常指令并调整到通讯保持状态。

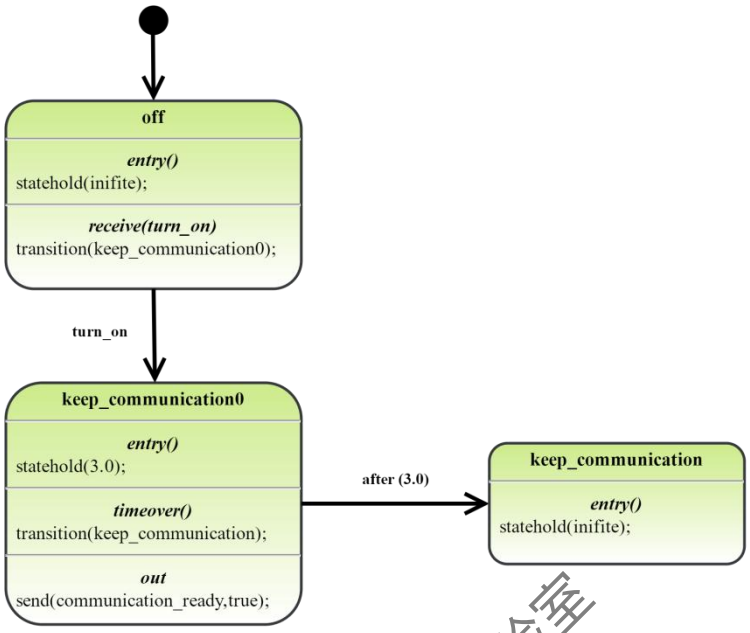
这里，基于 X 语言的离散类来实现通信系统的结构和行为描述。X 语言离散类在图形建模层面是由定义图和状态机图进行描述（如下图所示）。其中，定义图中描述了通信系统的输入输出信号；状态机图描述了通信系统的状态和行为逻辑。另外，在建立完通信系统的图形模型之后可自动生成 X 语言的文本模型（如图所示）。

图形建模

定义图：



状态机图：



文本建模

```
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;
```

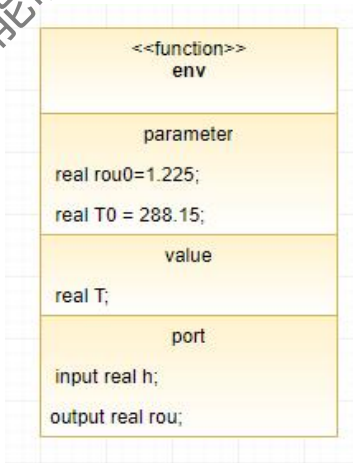
6.5.4.9 飞行环境系统

飞行环境系统的建立主要是为了使得仿真场景更加的真实。由于飞行的高度的变化，大气密度是不断变化的，然而大气密度对动压（动压的变化会导致飞机空气动力和空气动力矩的变化）计算有影响。综上所述，飞行环境系统的输入为飞行仿真中的实时飞行高度，输出为大气密度 ρ 。

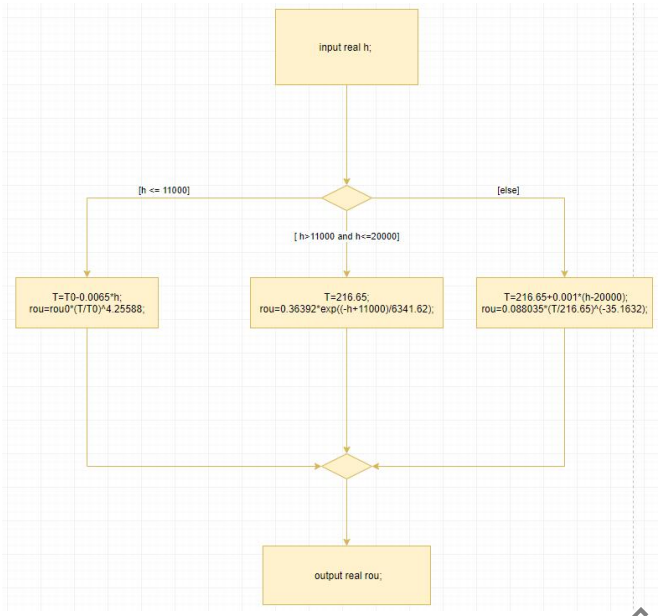
这里，基于 X 语言的函数类来实现飞行环境系统的结构和行为描述。X 语言函数类在图形建模层面是由定义图和活动图进行描述（如下图所示）。其中，定义图中描述了飞行环境系统的输入输出；活动图描述了飞行环境系统的行为逻辑。另外，在建立完飞行环境系统的图形模型之后可自动生成 X 语言的文本模型（如图所示）。

图形建模

定义图：



活动图：



文本建模

```
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 动力学系统

飞机起飞过程包括滑跑、抬轮、爬升三个阶段，整个过程主要受到空气作用于飞机的升力 T 和阻力 D、地面的支持力 F_N、摩擦力 F_f、重力 G 和推力 T。

飞机在滑跑阶段，与地面接触，此时存在地面的支持力 F_N 和摩擦力 F_f。当飞机滑轮离开地面时，支持力 F_N 和摩擦力 F_f 会瞬间消失。通过对飞机起飞过程受力分析，在机体坐标系下，支持力 F_N、X 轴方向和 Z 轴方向的合力如下式所示。

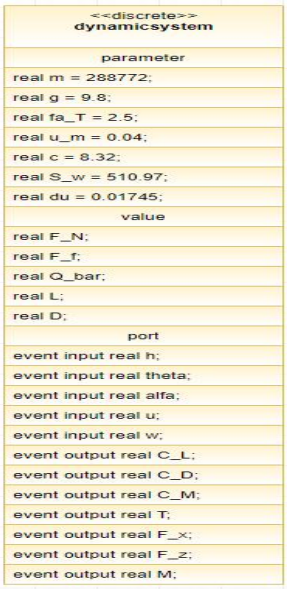
$$\begin{cases} F_N = -[G + D\sin(\theta - \alpha) - T\sin(\varphi_T + \theta) - L\cos(\theta - \alpha)] \\ F_x = T\cos\varphi_T - F_N\sin\theta + F_f\cos\theta - G\sin\theta - L\sin\alpha + D\cos\alpha \\ F_Z = -T\sin\varphi_T + F_N\cos\theta + F_f\sin\theta + G\cos\theta - L\cos\alpha + D\sin\alpha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

其中，φ_T为发动机安装角；θ为飞机俯仰角；α为飞机迎角。

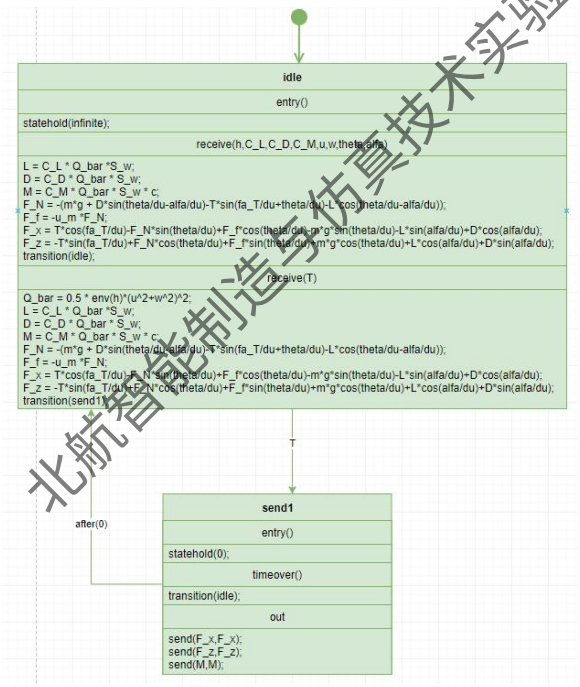
这里，基于 X 语言的离散类来实现飞行环境系统的结构和行为描述。X 语言离散类在图形建模层面是由定义图和状态机图进行描述（如下图所示）。其中，定义图中描述了飞行动力学系统的输入输出信号；状态机图描述了飞行动力学系统的状态和行为逻辑。另外，在建立完飞行动力学系统的图形模型之后可自动生成 X 语言的文本模型（如图所示）。

图形建模

定义图：



状态机图：



文本建模

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 运动学系统

由于飞机起飞场景纵向方向的运动变化是其核心，因此本文仅对飞机的纵向运动规律进行分析，如下式所示，飞机起飞过程的运动学方程为：

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

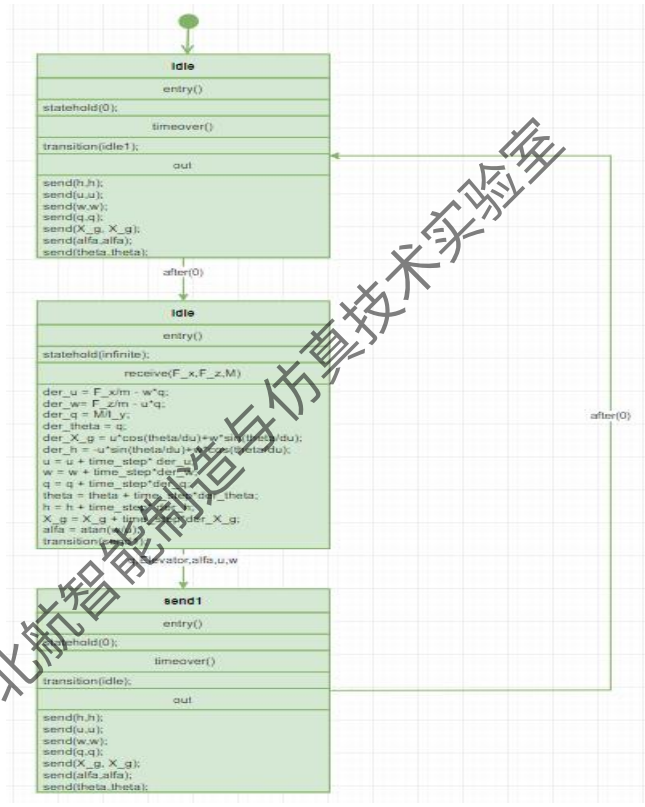
其中， u、w分别为 X 轴和 Z 轴方向速度；$\dot{u}$、$\dot{w}$分别为 X 轴和 Z 轴方向加速度；m 是飞机的重量；θ和q分别为俯仰角和俯仰角速率；X和Y分别为 X 轴和 Z 轴方向飞行距离。

这里，基于 X 语言的离散类来实现飞行环境系统的结构和行为描述。X 语言离散类在图形建模层面是由定义图和状态机图进行描述（如下图所示）。其中，定义图中描述了飞行运动学系统的输入输出信号；状态机图描述了飞行运动学系统的状态和行为逻辑。另外，在建立完飞行运动学系统的图形模型之后可自动生成 X 语言的文本模型（如图所示）。

图形建模

定义图：

状态机图：



文本建模

```
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 仿真结果

图 4.1 至图 4.4 展示了飞机在起飞的过程中各种性能参数（飞行高度、水平飞行距离、俯仰角以及速度等）的变化情况。结合图 4.1-4.3，我们可以发现飞机在速度快达到 80m/s 时进行抬轮动作；当飞机飞行离地面 25m 高度时，飞行速度接近 85m/s。另外，由图 4.4 可知，飞机在起飞过程中，飞机的俯仰角和迎角在起飞早期保持一致，大约在增大到 6°左右逐渐分离，后续俯仰角继续增大保持在 15°左右，迎角逐渐减小至 0，这说明在起飞过程中，当俯仰角增大到 6°左右，飞机正在离开地面并最后以 15°左右的俯仰角爬升。

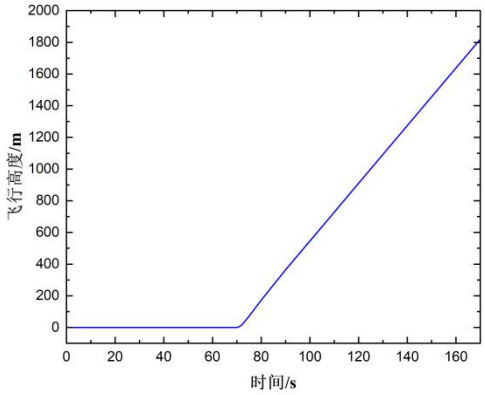


图 4.1 飞行高度随时间变化仿真结果

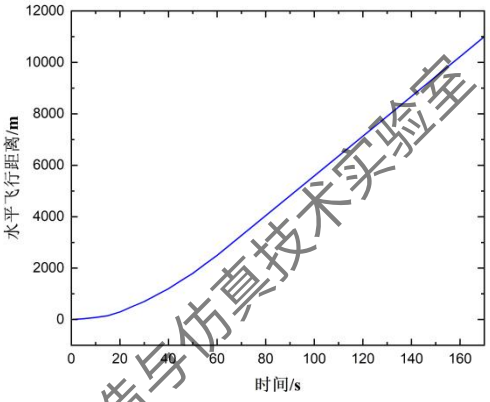


图 4.2 水平飞行距离随时间变化仿真结果

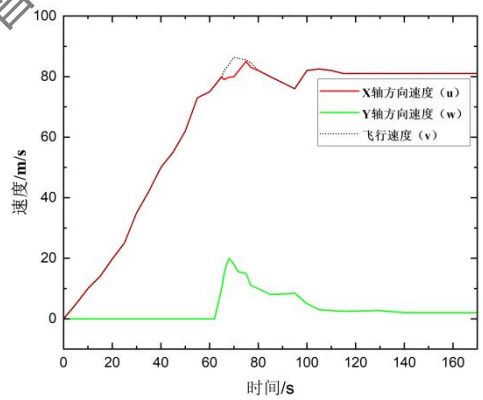


图 4.3 速度随时间变化仿真结果

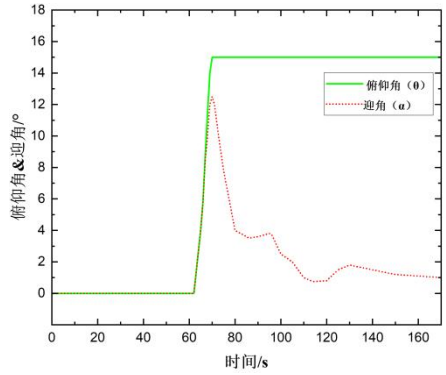


图 4.4 俯仰角&迎角随时间变化仿真结果