
X 语言建模规范

目 录

1 X 语言概述	7
1.1 X 语言研发背景.....	7
1.2 X 语言核心架构.....	8
2 X 语言关键词、操作符、数据类型及内置函数	12
2.1 关键字.....	12
2.2 操作符.....	13
2.3 数据类型.....	13
2.3.1 int	13
2.3.2 real.....	13
2.3.3 string.....	13
2.3.4 bool.....	14
2.3.5 数组.....	14
2.3.6 list.....	14
2.3.7 map.....	14
2.4 内置函数.....	15
2.4.1 run(plan planName, ...)	15
2.4.2 send(message msg)/ send(outputport o1, outvalue v1).....	15
2.4.3 receive()/receive(inputport1,inputport2,inputport3...)	15
2.4.4 statehold(real time1)	15
2.4.5 entry().....	15
2.4.6 timeover().....	15
2.4.7 trasiition(state s1).....	15
2.4.8 real random (real a,real b, int randtype)	15
2.4.9 void seed (real a)	16
2.4.10connect(connector 1,connector2)/ connect(output,input)	16
2.4.11 mathlib (函数库)	16
3 X 语言类的描述规范	17
3.1 连续类.....	17

3.1.1 简介.....	17
3.1.2 基本结构.....	17
3.1.3 示例代码.....	18
3.2 离散类.....	18
3.2.1 简介.....	18
3.2.2 基本结构.....	18
3.2.3 用法.....	20
3.2.4 建模案例.....	23
3.3 智能体类.....	24
3.4 耦合类.....	24
3.4.1 简介.....	24
3.4.2 基本结构.....	24
3.4.3 用法.....	25
3.4.4 建模案例（水箱模型）.....	26
3.5 连接器类.....	26
3.5.1 简介.....	26
3.5.2 基本结构.....	26
3.5.3 示例代码.....	26
3.6 记录类.....	27
3.6.1 简介.....	27
3.6.2 基本结构.....	27
3.6.3 示例代码.....	27
3.6.4 用法（record 的构造函数）.....	27
3.7 函数类.....	28
3.7.1 基本结构.....	28
3.7.2 示例代码.....	28
4 X 语言详细规范总结.....	29
4.1 词法正则表达式.....	29
4.2 X 语言巴克斯范式.....	29

5 X 语言图形描述规范	30
5.1 需求图.....	30
5.1.1 目的.....	30
5.1.2 何时创建需求图.....	30
5.1.3 利益相关者	30
5.1.4 需求.....	30
5.1.5 需求关系.....	31
5.2 用例图.....	36
5.2.1 目的.....	36
5.2.2 何时创建用例图.....	36
5.2.3 用例.....	37
5.2.4 系统边界.....	37
5.2.5 执行者.....	37
5.2.6 执行者与用例关联.....	37
5.2.7 基础用例.....	38
5.2.8 内含用例.....	38
5.2.9 扩展用例.....	38
5.3 定义图.....	39
5.3.1 目的.....	39
5.3.2 何时创建定义图.....	39
5.3.3 定义图的元素和关系.....	39
5.4 连接图.....	43
5.4.1 目的.....	43
5.4.2 何时创建连接图.....	43
5.4.3 组成部分属性.....	43
5.4.4 连接器.....	44
5.5 方程图.....	44
5.5.1 目的.....	44
5.5.2 何时创建方程图.....	44

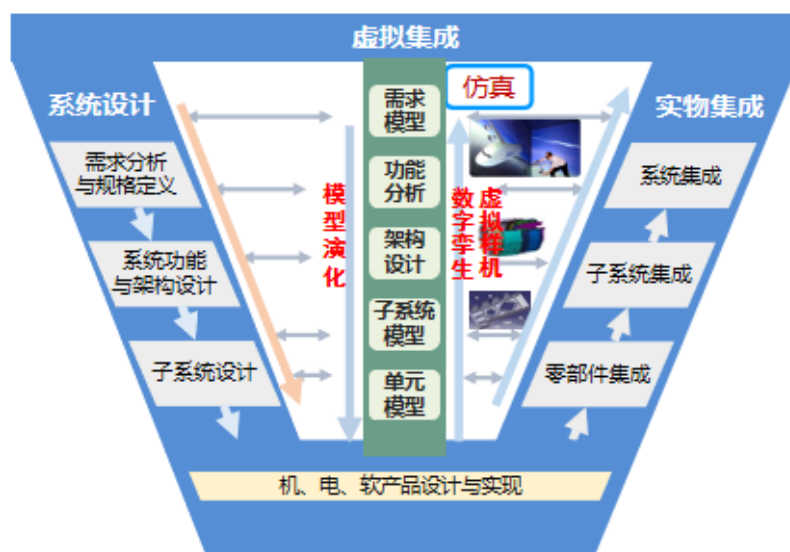
5.5.3	方程类型.....	44
5.6	状态机图.....	46
5.6.1	目的.....	46
5.6.2	何时创建状态机图.....	47
5.6.3	状态.....	47
5.6.4	转换.....	47
5.6.5	事件类型.....	48
5.6.6	伪状态.....	48
5.7	活动图.....	50
5.7.1	目的.....	50
5.7.2	何时创建活动图.....	50
5.7.3	动作.....	50
5.7.4	活动参数.....	50
5.7.5	控制节点.....	50
6 X	语言建模案例分析.....	53
6.1	水箱模型.....	53
6.1.1	背景描述.....	53
6.1.2	建模分析.....	53
6.1.3	系统模型.....	53
6.1.4	子系统模型.....	53
6.2	看门狗模型.....	53
6.2.1	背景描述.....	53
6.2.2	建模分析.....	53
6.2.3	系统模型.....	54
6.2.4	子系统模型.....	54
6.3	电路模型.....	54
6.3.1	背景描述.....	54
6.3.2	建模分析.....	54
6.3.3	系统模型.....	54

6.3.4	子系统模型.....	54
6.4	攻防对抗模型.....	55
6.4.1	背景描述.....	55
6.4.2	建模分析.....	55
6.4.3	系统模型.....	55
6.4.4	子系统模型.....	55
6.5	导弹姿态控制模型.....	55
6.5.1	背景描述.....	55
6.5.2	建模分析.....	55
6.5.3	系统模型.....	56
6.5.4	子系统模型.....	56
6.6	飞机起飞模型.....	56
6.6.1	背景描述.....	56
6.6.2	建模分析.....	56
6.6.3	系统模型.....	56
6.6.4	子系统模型.....	56

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（MSBSE），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

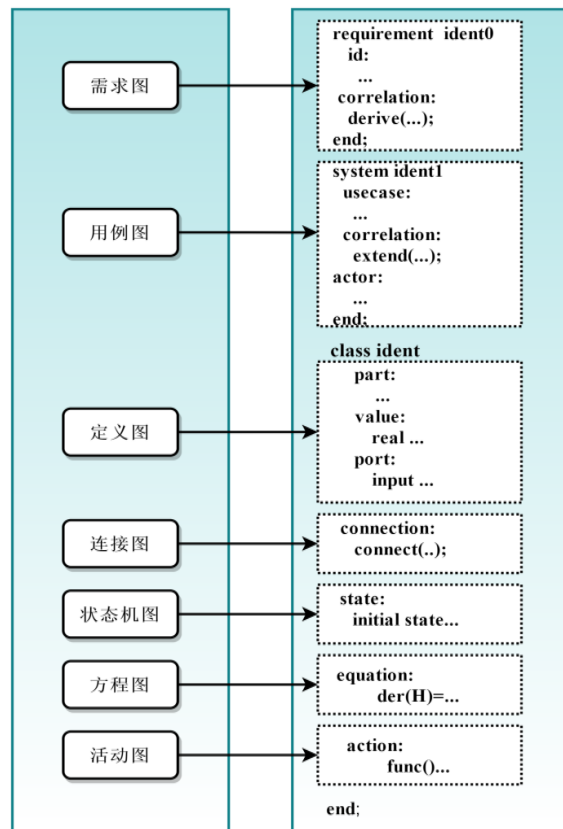
基于物理建模语言（如 Modelica、Matlab/Simulink 等）并配合集成标准规范（FMI、HLA 等），实现对复杂产品系统设计和仿真的集成；方法二是通过建立系统建模语言（如 SysML、IDEF 等）和物理建模语言（如 Modelica、Matlab/Simulink 等）的映射关系实现系统模型和物理模型的自动转换，达到对复杂产品系统设计和仿真的集成。

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。

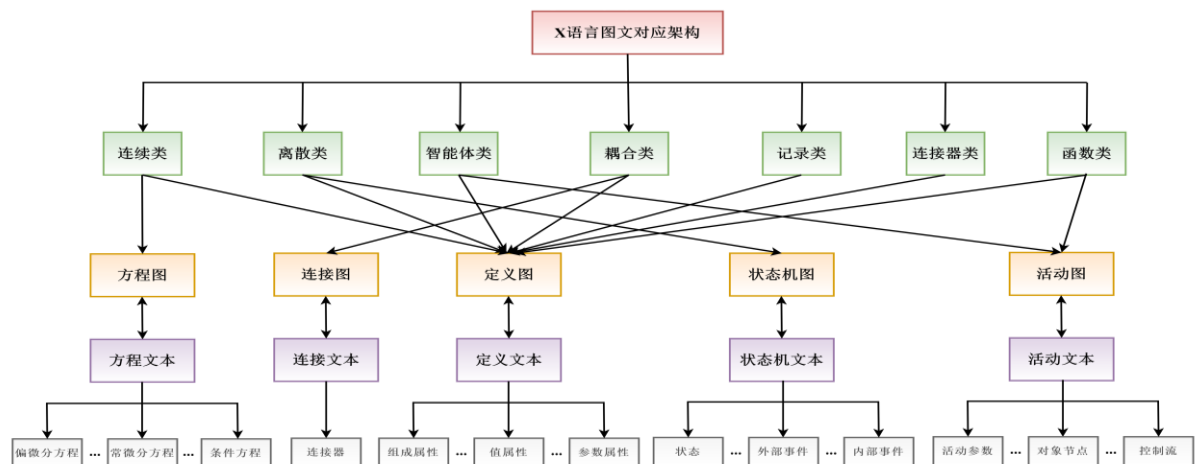
基于此，我们提出了一种面向复杂系统支持 MBSE 的新一代一体化建模仿真语言—X 语言。X 语言全面支持基于模型的系统工程（MBSE），在产品概念设计阶段提供规范的图形化建模描述，还可将规范的图形化模型自动 编译转换成文本化的底层仿真模型，在仿真引擎的驱动下，支持全系统、全流程、多视角的无缝集成仿真，实现从概念模型设计、 系统架构设计、多物理域模型到仿真模型的统一的一体化描述和仿真。

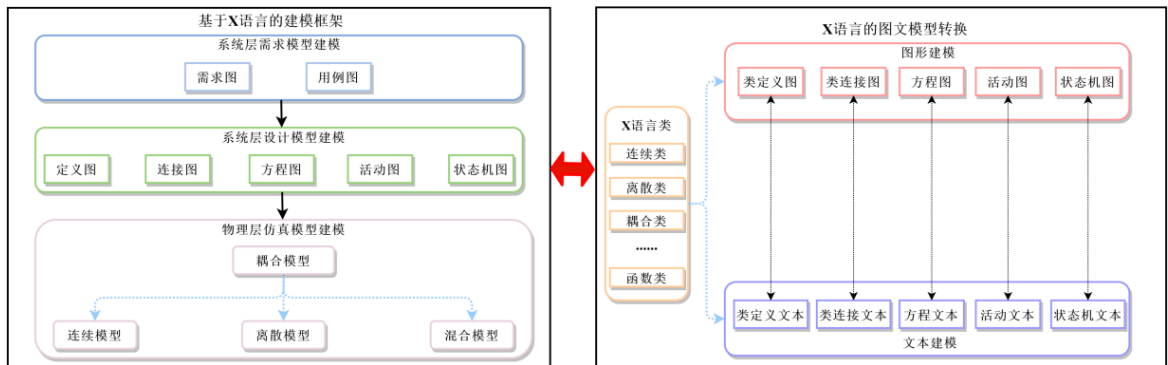
1.2 X 语言核心架构

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。



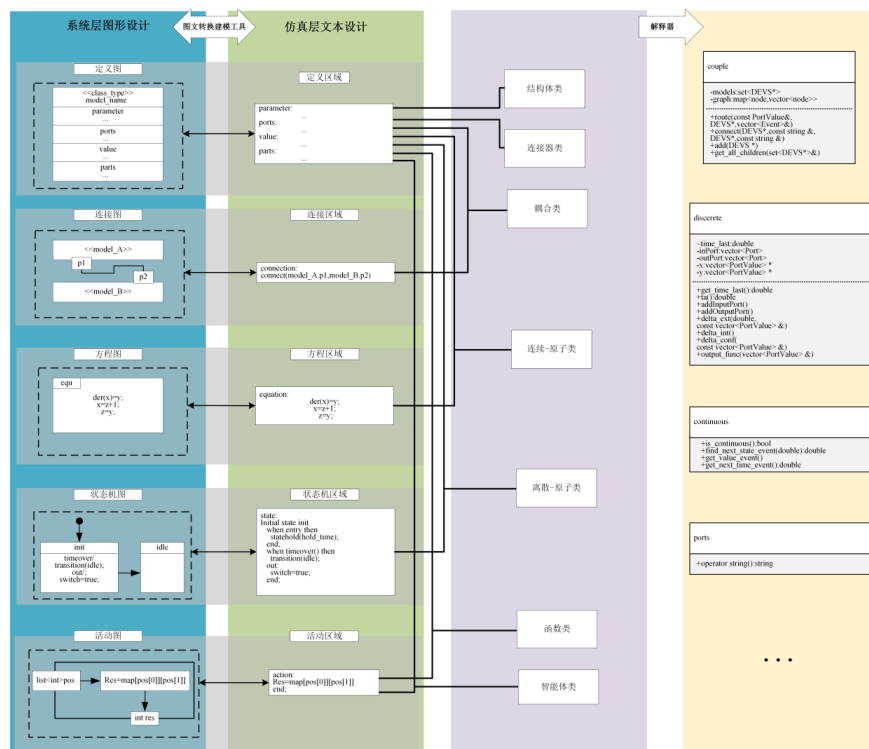
X 语言深度融合现有建模语言 SysML、Modelica 的思想及描述规范，基于 DEVS（离散事件系统规范），实现系统架构和物理特性的一体化建模，并基于统一的仿真引擎实现全系统模型的仿真验证。X 语言是一种面向对象的语言，在 X 语言中定义了七种特定类（连续、离散、耦合等），可实现对多特征，多粒度模型的描述。每种类都具有图形和文本两种建模形式，二者之间一一对应，可相互转换。





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。

面向复杂系统统一建模模型框架方法图



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

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 insertVariable)`。

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 trasion(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。`import` 指的是下文需要对外部类进行实例化，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。

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。

3.1.3 示例代码

```
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 initial state 加初始状态机名字组成。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 initial 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 用法

3.2.3.1 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）

3.2.3.2 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。

3.2.3.3 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  $\subseteq V_{state}$ 
...
    transition(state2);
out
    send(port1,value); //port1  $\subseteq X$ 
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。

3.2.3.4 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 i 变量由 1 变化到 10 进行循环。

3.2.3.5 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 expression2 then
    ...
end;
```

3.2.4 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

3.4 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 `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 LiquidSource;
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。

3.5.3 示例代码

```
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 示例代码

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

3.6.4 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。

3.7 函数类

3.7.1 基本结构

函数类以关键字 `function` 统领。包含头部份，定义部分以及行为部分。头部份可以导入外部函数类。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。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。

3.7.2 示例代码

```
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;
```

4 X 语言详细规范总结

4.1 词法正则表达式

4.2 X 语言巴克斯范式

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。

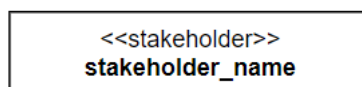
5.1.2 何时创建需求图

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的。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，是整个项目开发的需求提出者。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 需求

需求的标识法是一个矩形，在名称之前有元类型《requirement》。需求有 4 个属性：id、type、level、text，这四种属性的类型都是 string。其中，id 表示该需求的编

号、type 表示需求的类型（一般包括一般性需求、功能性需求以及非功能性需求）、level 表示需求的层级（一般包括利益相关者需求、系统级需求以及组件级需求）、text 一般通过自然语言描述需求的具体内容。

具体形式如下：

<<requirement>> requirement_name
Id id="text string"
Type type=General/Functional/Nonfunctional
Level level =stakeholderReq/systemReq/componentReq
Text text="text string"

5.1.5 需求关系

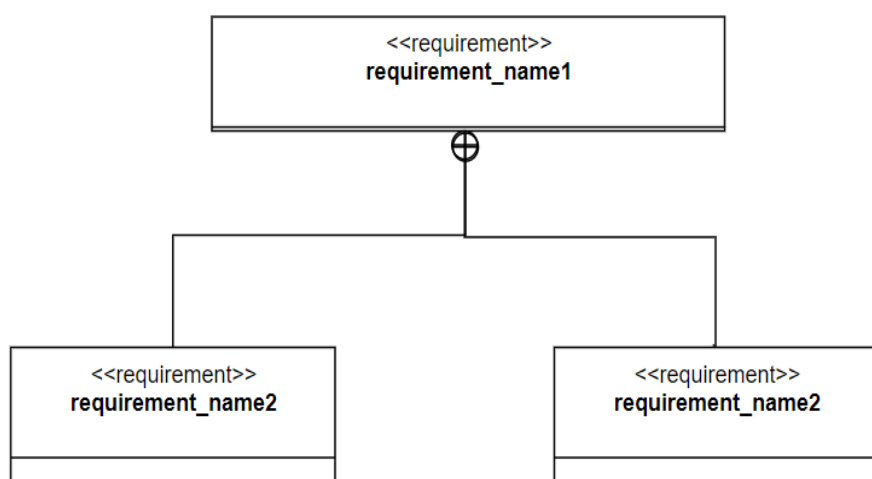
在系统模型中记录需求很有用。然而，需求和其他模型元素直接的关系有更大的价值。在建模过程中，一般可能会使用六种需求关系：包含、跟踪、继承、改善、满足以及验证。

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。

5.1.5.1 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：

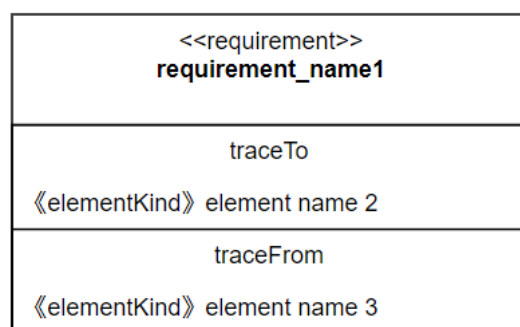


图中表示需求 1 包含需求 2 和需求 3。

5.1.5.2 跟踪关系

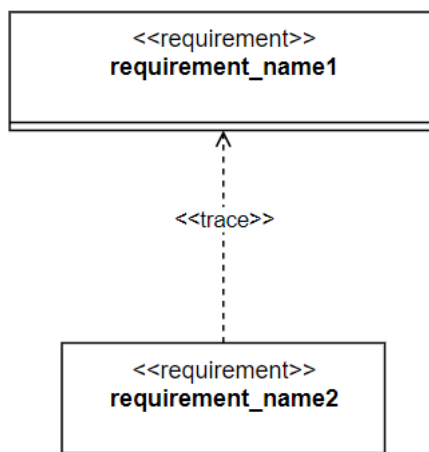
正式情况下，跟踪关系是一种依赖关系。跟踪关系是一种弱关系。它只是表达了一种基本的依赖关系：对提供方元素的修改可能会导致对客户端元素修改的需要。跟踪关系一般可以通过需求属性栏或者通过带有开口箭头的虚线（上方带有《trace》元类型）表示。

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

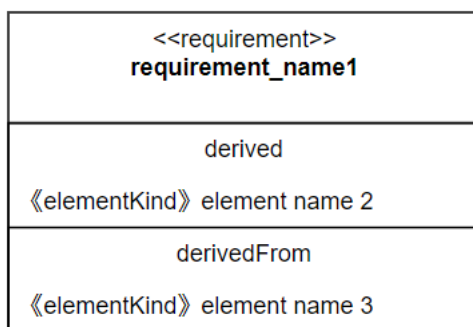


图中表示需求 2 会跟踪回需求 1

5.1.5.3 继承需求关系

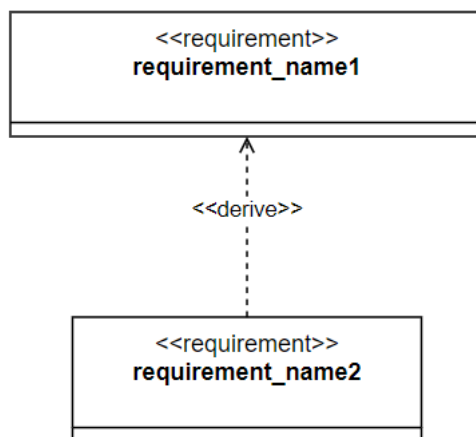
继承需求关系是另一种依赖关系。这种关系必须在客户端和提供方端都有需求。继承需求关系表示客户端的需求继承了提供方的需求。继承需求关系一般可以通过需求属性栏或者通过带有开口箭头的虚线（上方带有《derive》元类型）表示。拥有多级继承关系完全是合法的，并且依赖关系是可传递的。因此，如果基本的需求发生变更，那么下游的影响会贯穿整个继承需求关系链。

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

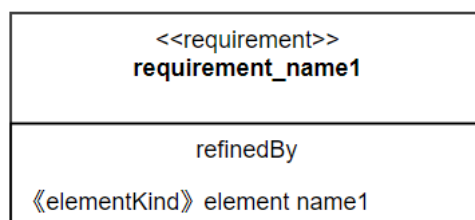


图中表示需求 2 继承自需求 1

5.1.5.4 改善关系

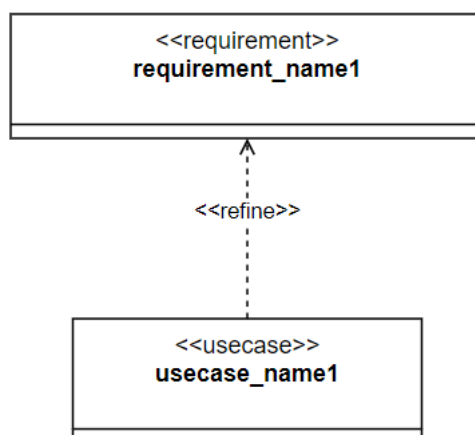
改善关系是另一种依赖关系。对于改善关系的任意一端所能够显示的元素种类，X 语言没有任何限制。然而，一般使用用例对文本的功能性需求进行改善。改善关系表示客户端的元素要比提供方端的元素更加具体。改善关系一般可以通过需求属性栏或者通过带有开口箭头的虚线（上方带有《refine》元类型）表示。

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

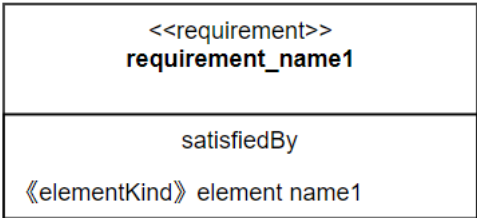


图中表示用例 1 改善了需求 1

5.1.5.5 满足关系

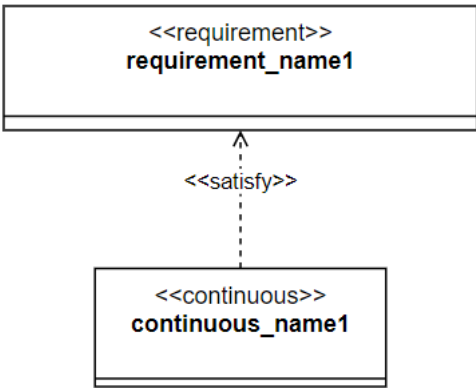
满足关系是另一种依赖关系。这种关系在提供方端必须有一个需求。X 语言没有对客户端所能够出现的元素种类施加限制。然而，客户端元素通常是类（连续类、离散类等）。满足关系是一种断言，说明客户端类的实例会满足提供方端的需求。满足关系一般可以通过需求属性栏或者通过带有开口箭头的虚线（上方带有《satisfy》元类型）表示。

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

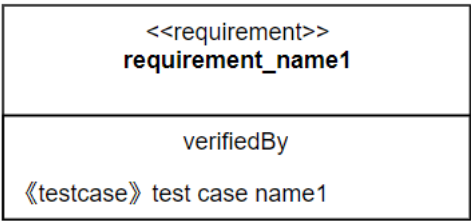


图中表示连续类 1 会满足需求 1

5.1.5.6 验证关系

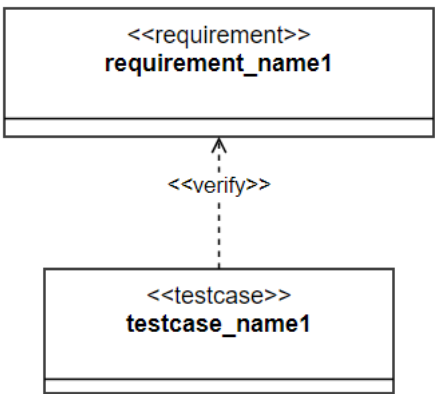
验证关系是另一种依赖关系。和满足关系一样，验证关系必须在提供方端有一个需求。X 语言没有对客户端所能够出现的元素种类施加限制。然而，客户端元素通常是测试案例。测试案例一般会是一种系统行为，当进行仿真的时候，会证明系统是否真正满足了需求。验证关系一般可以通过需求属性栏或者通过带有开口箭头的虚线（上方带有《verify》元类型）表示。

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

5.2 用例图

5.2.1 目的

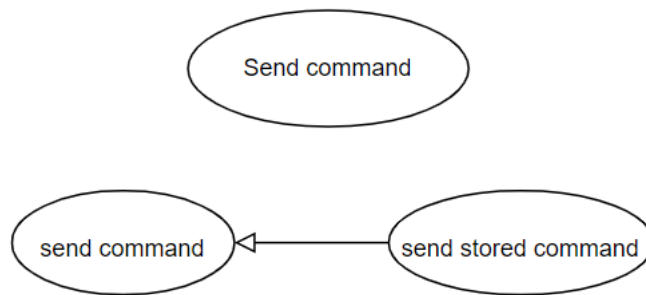
用例图会简洁地传递一系列用例——系统提供的外部可见服务——以及出触发和参与用例的参与者。用例图是系统的一种黑盒视图，因此也很适合作为系统的情境图。

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。

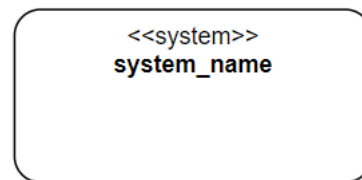
5.2.3 用例

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

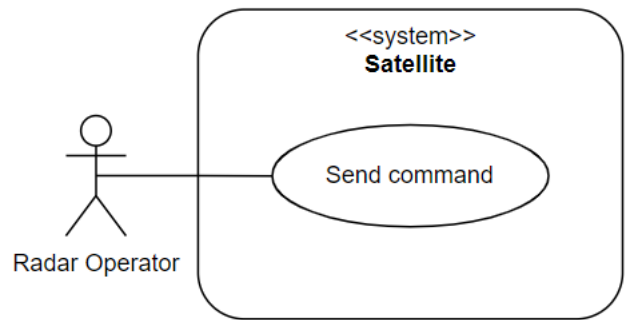
执行者有两种标识法：火柴棍小人，或者是名称前面带有《actor》关键字的矩形。一般建模者会使用火柴棍小人代表人，矩形标识法代表系统。



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。

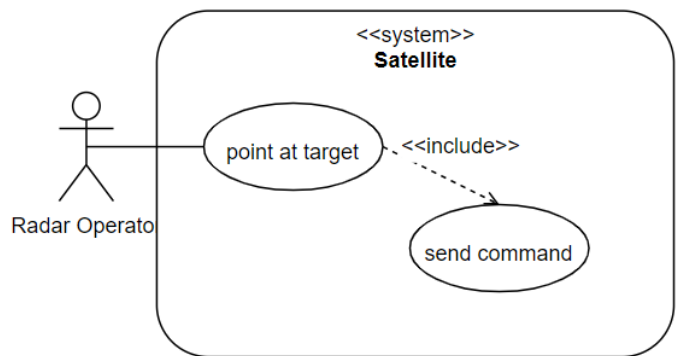


5.2.7 基础用例

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。

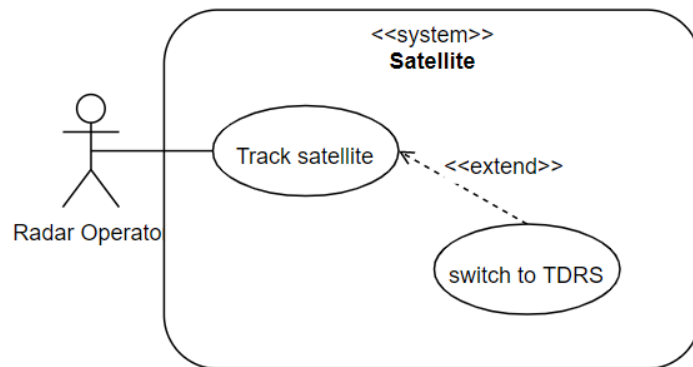
5.2.8 内含用例

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

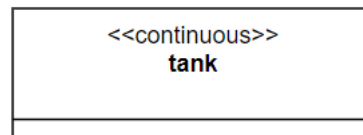
5.3.2 何时创建定义图

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。

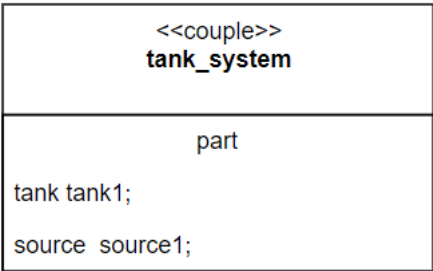


图中表示一个名为 tank 的连续类模型

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

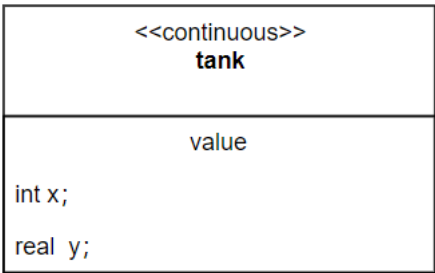
句话说，类是由组成部分属性构成的。这种关系是一种所属关系。组成部分属性一般是由类的下方带有元类型 **part** 的结构分隔框表示。



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

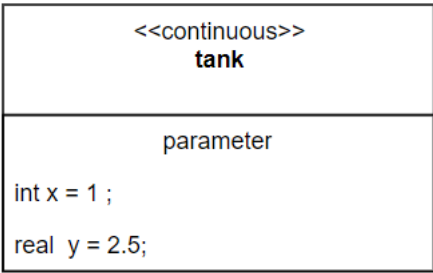
值属性代表类的状态变量，状态变量的类型可以是整型、实数型、布尔型等。值属性一般是由类的下方带有元类型 **value** 的结构分隔框表示。



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**

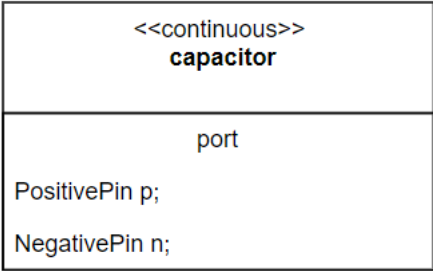
值为 2.5 的实例化参数。

5.3.3.5 端口

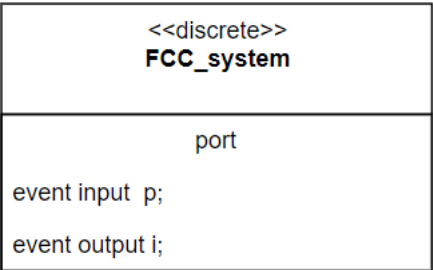
端口是代表结构边缘不同交互点的一种属性，通过那些点外部实体可以和那个结构交互（一般是交换事件、能量、数据等）。

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。



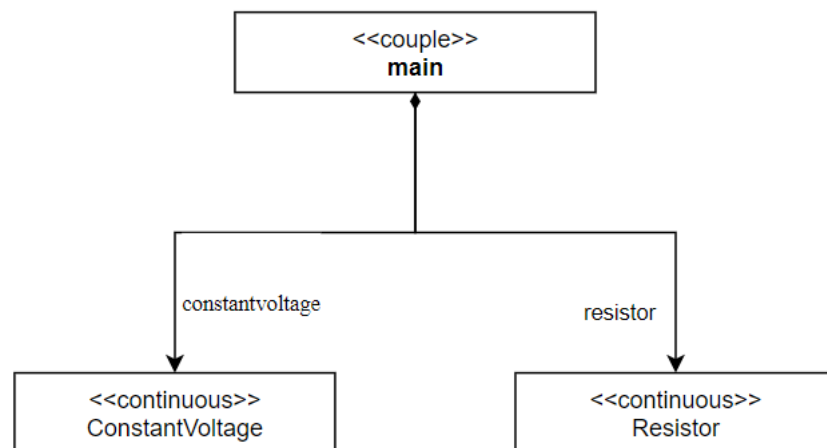
图中表示的是一个名为 **capacitor** 的连续类模型具有两个分别有 **PositivePin** 和 **NegativePin** 实例化的连接器类型端口 **p** 和 **n**。



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

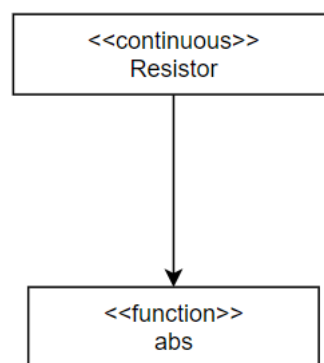
组合关联是组成部分属性的另一种表示法。两个类之间的组合关联表示结构上的分解。组合端的类实例由一些组成部分端类的实例组合而成。定义图中组合关联的标识法是两个模块之间的实线，在组合端有实心的菱形。



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

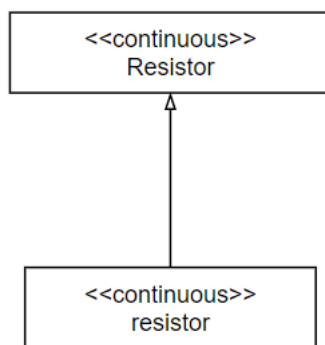
引用关联是类之间存在的一种连接关系，表示两个类之间的调用关系。定义图中引用关联的标识法是两个模块之间的实线箭头。



图中表示一个名为 **Resistor** 的连续类模型调用了—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

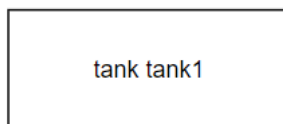
创建连接图是为了指定单个类（一般是耦合类）的内部结构。连接图会表达类内部的组成部分及其是如何交互才能够创建有效实例的。

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 and 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

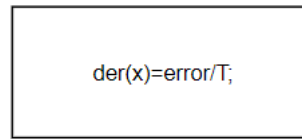
当我们所建立涉及到具有随时间变化的连续行为且物理特性明确的系统时，会创建方程图。方程图一般会 and 定义图组合构建连续类去描述一个连续系统。

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。

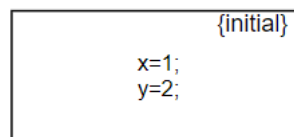
5.5.3.1 基础方程

基础方程一般都是微分代数方程（组）的形式。



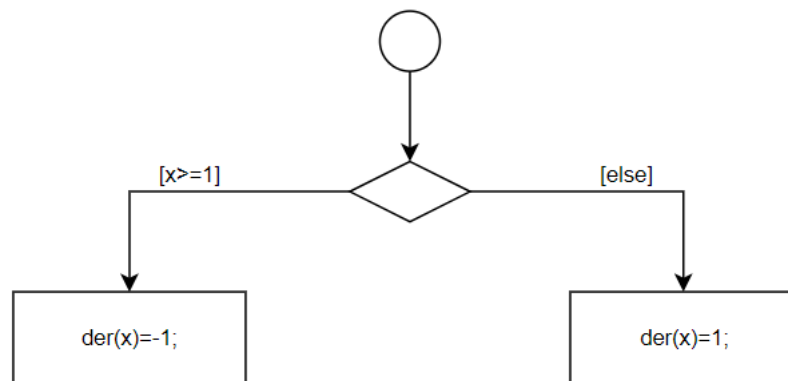
5.5.3.2 初始方程

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

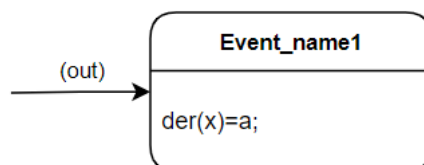
条件方程是带有 if-else 判断的微分代数方程形式。二分支的描述如下：（多分支描述同理）



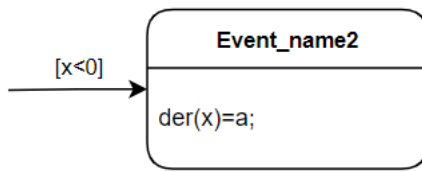
5.5.3.4 事件方程

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：

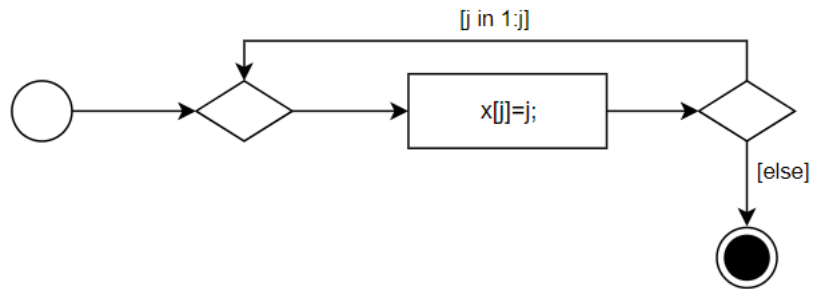


内部事件：



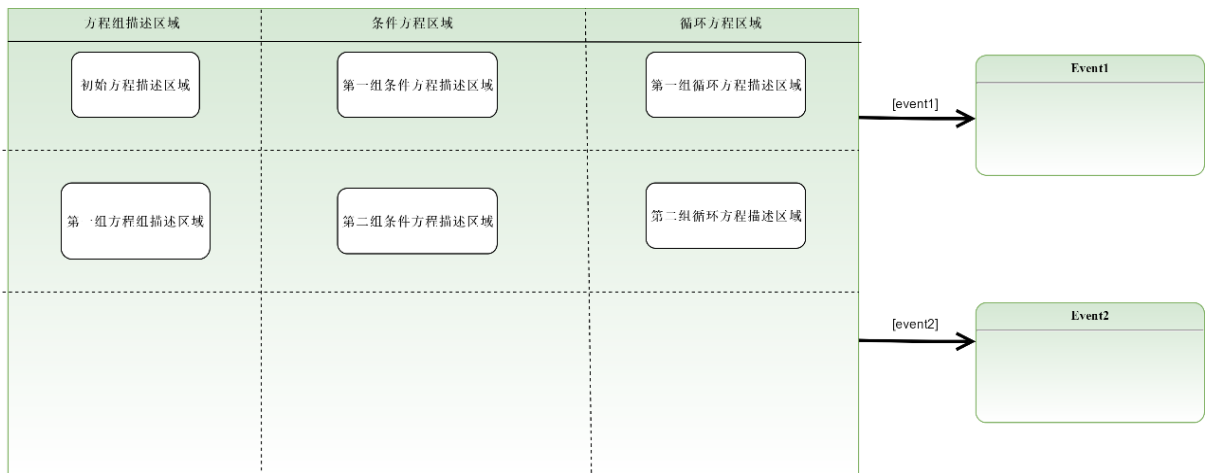
5.5.3.5 循环方程

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 and 定义图组合构建离散类去描述一个离散系统。

5.6.3 状态

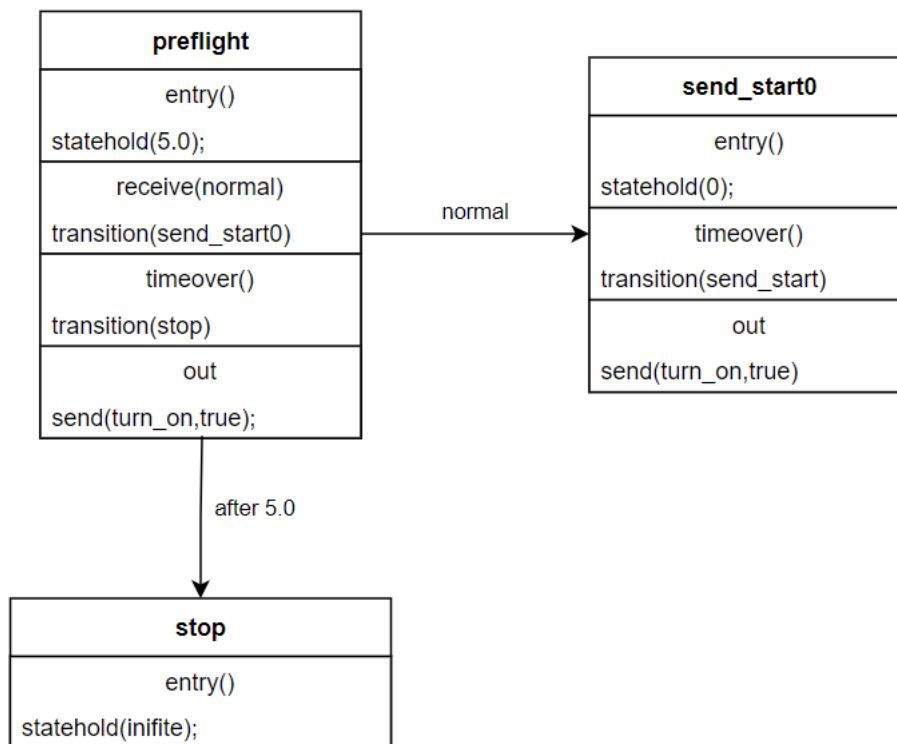
一个系统有时会拥有一系列定义好的状态，在系统操作过程中可以处于那些状态。一个正常的状态至少会包括处于状态的持续时间、接受外部事件以及内部事件触发的行为和输出。

preflight
entry() statehold(5.0);
receive(normal) transition(send_start0)
timeover() transition(stop)
out send(turn_on,true);

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 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。

5.6.5 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

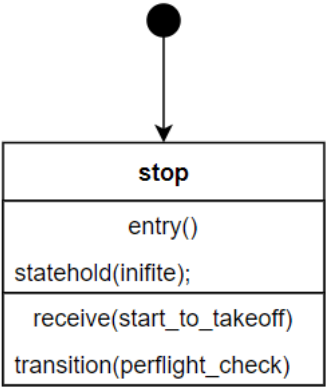
内部事件一般都是内部的时间事件。时间事件很直观，它代表时间中的实例。当那个时刻在系统操作过程中到来的时候，时间事件发生。

5.6.6 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。

5.6.6.1 初始伪状态

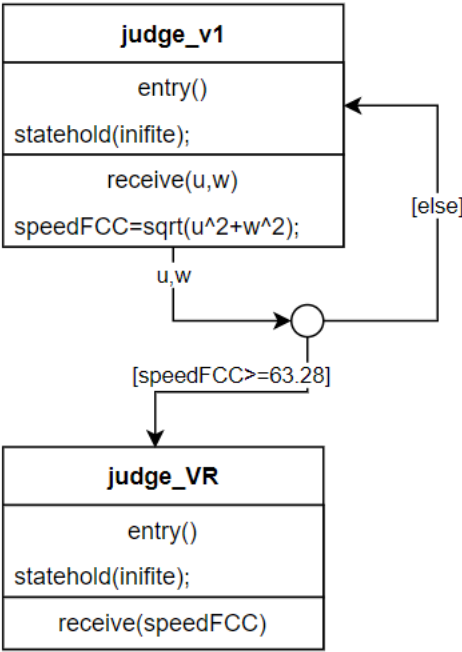
初始伪状态表示状态开始执行时的第一个状态。初始伪状态在 X 语言只是一种标识，标志着初始状态即将开始。初始伪状态的标识法是一个小型的实心圆。



图中表示系统的初始状态为 stop

5.6.6.2 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

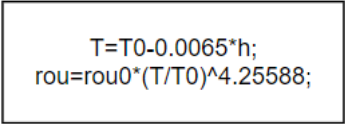
活动图是一种用于描述行为和事件发生序列的行为图。一般用于描述函数类和智能体类中 `plan` 的算法。

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 and 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

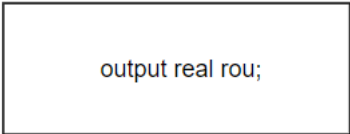
动作是一种可以存在于活动之中的节点，它是为活动基本的功能单元建模的节点。一个动作代表某种类型的处理或转换，它会在系统操作过程中活动被执行的时候发生。动作的标识法是矩形。在 X 语言中，活动一般都是文本形式的语句描述。

A rectangular box representing an activity node, containing two lines of text: `T=T0-0.0065*h;` and `rou=rou0*(T/T0)^4.25588;`

```
T=T0-0.0065*h;
rou=rou0*(T/T0)^4.25588;
```

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`。

A rectangular box representing an activity parameter, containing the text `output real rou;`

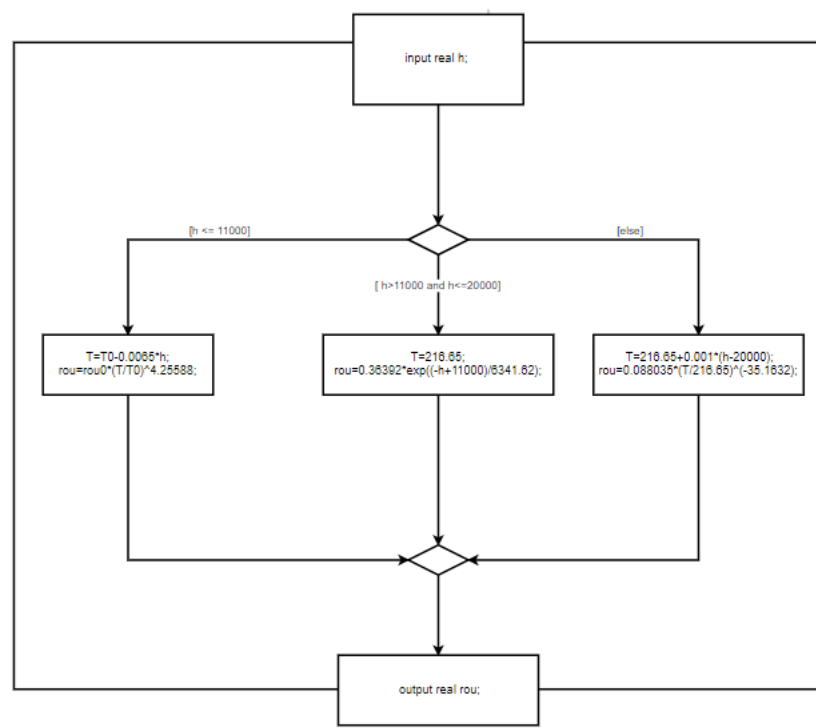
```
output real rou;
```

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。

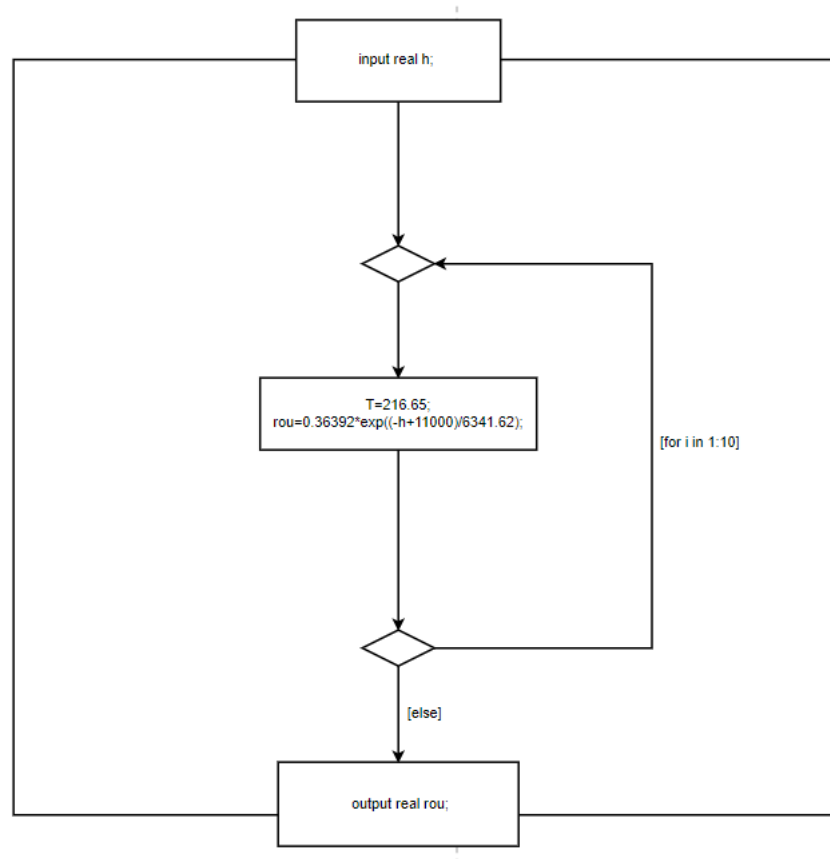
5.7.5.1 决定节点

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。



6 X 语言建模案例分析

6.1 水箱模型

6.1.1 背景描述

6.1.2 建模分析

6.1.3 系统模型

6.1.4 子系统模型

6.2 看门狗模型

6.2.1 背景描述

6.2.2 建模分析

6.2.3 系统模型

6.2.4 子系统模型

6.3 电路模型

6.3.1 背景描述

6.3.2 建模分析

6.3.3 系统模型

6.3.4 子系统模型

6.4 攻防对抗模型

6.4.1 背景描述

6.4.2 建模分析

6.4.3 系统模型

6.4.4 子系统模型

6.5 导弹姿态控制模型

6.5.1 背景描述

6.5.2 建模分析

6.5.3 系统模型

6.5.4 子系统模型

6.6 飞机起飞模型

6.6.1 背景描述

6.6.2 建模分析

6.6.3 系统模型

6.6.4 子系统模型