§5　模型检测方法

§5.1　基于状态分析的模型检测

符号模型

以 Kripke 结构 $M =< S, \Delta, I, L >$ 为出发点，我们可以将 S 编号。若 S 有 $\leq 2^n$ 个元素，我们可以用 n 个命题来表示，例如，若 $n = 4$ 且其元素依次记为 $s_0, s_1, s_2, ...$，则 $(a_1, \neg a_2, a_3, \neg a_4)$ 代表 s_{10} 。

状态的集合可以用公式表示。一个可满足的公式有一个或多个模型，每个模型对应一个元素。一个公式则表示其模型所对应的元素的集合。比如公式 $a_1 \wedge \neg a_2 \wedge a_3 \wedge \neg a_4$ 表示 s_{10} ， $\neg a_2 \wedge a_3 \wedge \neg a_4$ 表示 $\{s_2, s_{10}\}$ 。

状态的转化关系的集合一样可以用公式表示。这里，我们还需引进 n 个新变量，记作 $a'_1, ..., a'_n$ 。比如状态转换关系 (s_2, s_{10}) 可以表示为

$$\neg a_1 \wedge \neg a_2 \wedge a_3 \wedge \neg a_4 \wedge a'_1 \wedge \neg a'_2 \wedge a'_3 \wedge \neg a'_4$$

状态转换关系的集合 $\{(s_2, s_{10}), (s_{10}, s_{10})\}$ 可以表示为

$$(\neg a_1 \wedge \neg a_2 \wedge a_3 \wedge \neg a_4 \wedge a'_1 \wedge \neg a'_2 \wedge a'_3 \wedge \neg a'_4) \vee (a_1 \wedge \neg a_2 \wedge a_3 \wedge \neg a_4 \wedge a'_1 \wedge \neg a'_2 \wedge a'_3 \wedge \neg a'_4)$$

这个公式可以简化为

$$(\neg a_2 \wedge a_3 \wedge \neg a_4 \wedge a'_1 \wedge \neg a'_2 \wedge a'_3 \wedge \neg a'_4)$$

在这个框架下， Kripke 结构中 L 的没有很好的表示的方法。由于从给定的 L ，对任意 $p \in AP$ ，我们可以计算 $L'(p) = \{s \mid L(s) \models p\}$ 且根据 L' ，我们可以计算 $L(s) = \{p \in AP \mid s \in L'(p)\}$ ，可以说 L 和 L' 作为系统模型中描述状态满足哪些基本命题的作用是等价的。这样，在这个框架下，我们可以用 L' 取代 L 作为系统描述的一部分。由于 $L'(p)$ 是一个状态集合， $L'(p)$ 可以用一个公式来表示。

由公式的运算，我们可以得到集合的运算。集合的交集就是公式的合取。集合的并集就是公式的析取。集合的子集关系就是公式的蕴含关系。集合的补集就是公式的否。

设 $AP = \{p_1, ..., p_k\}$ 上的 Kripke 结构 $M =< S, \Delta, I, L >$ 有 2^n 个状态。则 M 可以用以下 $k+2$ 个命题逻辑公式描述。

$$\varphi(a_1, ..., a_n, a'_1, ..., a'_n), \psi_0(a_1, ..., a_n), \psi_1(a_1, ..., a_n), \cdots, \psi_k(a_1, ..., a_n)$$

有了一个表示系统的方法，重要的是进一步的分析。给定一个公式 $\zeta(a_1, ..., a_n)$ ，其后区状态集合可以由以下公式得到。

$$\exists a_1 ... a_n . (\varphi(a_1, ..., a_n, a'_1, ..., a'_n) \wedge \zeta(a_1, ..., a_n))$$

从应用来讲，需要把它化简成只包含 $a'_1, ..., a'_n$ 的公式 $\varphi'(a'_1, ..., a'_n)$ ，然后再将其中的 $a'_1, ..., a'_n$ 替换为 $a_1, ..., a_n$ ，

使之成为正常的表示状态的公式。存在量词可以用以下等式消去

$$\exists x.\varphi = (\varphi|_{x=0} \vee \varphi|_{x=1})$$

设 $n = 4$, $\varphi(a_1, ..., a_n, a'_1, ..., a'_n) = (\neg a_2 \wedge a_3 \wedge \neg a_4 \wedge a'_1 \wedge \neg a'_2 \wedge a'_3 \wedge \neg a'_4)$ 。

若 $\zeta(a_1, ..., a_n) = a_1 \wedge \neg a_2 \wedge a_3 \wedge \neg a_4$ ，则

$$\exists a_1...a_n.(\varphi(a_1, ..., a_n, a'_1, ..., a'_n) \wedge \zeta(a_1, ..., a_n)) = a'_1 \wedge \neg a'_2 \wedge a'_3 \wedge \neg a'_4$$

即 $\zeta(a_1, ..., a_n) = a_1 \wedge \neg a_2 \wedge a_3 \wedge \neg a_4$ 的后区公式为 $a_1 \wedge \neg a_2 \wedge a_3 \wedge \neg a_4$ 。

若 $\zeta(a_1, ..., a_n) = a_1 \wedge a_2 \wedge a_3 \neg a_4$ ，则

$$\exists a_1...a_n.(\varphi(a_1, ..., a_n, a'_1, ..., a'_n) \wedge \zeta(a_1, ..., a_n)) = false$$

即 $\zeta(a_1, ..., a_n) = a_1 \wedge a_2 \wedge a_3 \wedge \neg a_4$ 的后区为空。

公式 $\zeta(a_1, ..., a_n)$ 的前区状态集合可以由以下公式得到。

$$\exists a'_1...a'_n.(\varphi(a_1, ..., a_n, a'_1, ..., a'_n) \wedge \zeta(a'_1, ..., a'_n))$$

从理论上讲，根据前区和后区的求法，我们可以用不动点算法求得系统的可达集合，然后计算其与表示性质的公式是否有子集关系。

二叉决策图

由于不动点算法需计算两个集合是否相等，即两个逻辑公式是否等价，而计算一般逻辑公式是否等价的复杂性很高。若不能降低计算逻辑公式是否等价的代价，则这种算法的实用性有限。

从集合的比较来讲，把所有元素有序罗列，则两集合的等价就非常简单。但是若元素很多，最好能够在用一个公式表示多个元素的同时便于比较。首先可以考虑在逻辑的框架下用罗列的办法。比如 $\{s_{10}, s_2\}$ 可以表示为以下公式的集合。

$$a_1 \wedge \neg a_2 \wedge a_3 \wedge \neg a_4$$
$$\neg a_1 \wedge \neg a_2 \wedge a_3 \wedge \neg a_4$$

这样两个集合的比较就很简单。由于运算中，需要求补等运算， $\{s_{10}, s_2\}$ 的补集为以下公式的集合。

$\neg a_1 \wedge \neg a_2 \wedge \neg a_3 \wedge \neg a_4$	$a_1 \wedge \neg a_2 \wedge \neg a_3 \wedge \neg a_4$
$\neg a_1 \wedge \neg a_2 \wedge \neg a_3 \wedge a_4$	$a_1 \wedge \neg a_2 \wedge \neg a_3 \wedge a_4$
$\neg a_1 \wedge \neg a_2 \wedge a_3 \wedge a_4$	$a_1 \wedge \neg a_2 \wedge a_3 \wedge a_4$
$\neg a_1 \wedge a_2 \wedge \neg a_3 \wedge \neg a_4$	$a_1 \wedge a_2 \wedge \neg a_3 \wedge \neg a_4$
$\neg a_1 \wedge a_2 \wedge \neg a_3 \wedge a_4$	$a_1 \wedge a_2 \wedge \neg a_3 \wedge a_4$
$\neg a_1 \wedge a_2 \wedge a_3 \wedge \neg a_4$	$a_1 \wedge a_2 \wedge a_3 \wedge \neg a_4$
$\neg a_1 \wedge a_2 \wedge a_3 \wedge a_4$	$a_1 \wedge a_2 \wedge a_3 \wedge a_4$

为了使计算补集能够简单, 我们将所有元素的信息都存在一个集合的表示之中， $\{s_{10}, s_2\}$ 可以表示为以下公式的集合。

$$
\begin{array}{ll}
\neg a_1 \wedge \neg a_2 \wedge \neg a_3 \wedge \neg a_4 = false & a_1 \wedge \neg a_2 \wedge \neg a_3 \wedge \neg a_4 = false \\
\neg a_1 \wedge \neg a_2 \wedge \neg a_3 \wedge a_4 = false & a_1 \wedge \neg a_2 \wedge \neg a_3 \wedge a_4 = false \\
\neg a_1 \wedge \neg a_2 \wedge a_3 \wedge \neg a_4 = true & a_1 \wedge \neg a_2 \wedge a_3 \wedge \neg a_4 = true \\
\neg a_1 \wedge \neg a_2 \wedge a_3 \wedge a_4 = false & a_1 \wedge \neg a_2 \wedge a_3 \wedge a_4 = false \\
\neg a_1 \wedge a_2 \wedge \neg a_3 \wedge \neg a_4 = false & a_1 \wedge a_2 \wedge \neg a_3 \wedge \neg a_4 = false \\
\neg a_1 \wedge a_2 \wedge \neg a_3 \wedge a_4 = false & a_1 \wedge a_2 \wedge \neg a_3 \wedge a_4 = false \\
\neg a_1 \wedge a_2 \wedge a_3 \wedge \neg a_4 = false & a_1 \wedge a_2 \wedge a_3 \wedge \neg a_4 = false \\
\neg a_1 \wedge a_2 \wedge a_3 \wedge a_4 = false & a_1 \wedge a_2 \wedge a_3 \wedge a_4 = false
\end{array}
$$

这样的一个集合可用有序二元决策图表示。

可以看出这样的一个表示方法有很多冗余部分。将冗余部分简化可得以下二元决策图。

§5.2 基于路径分析的模型检测

§5.3 限界模型检测

§5.4 工具

§5.4.1 SMV 的使用

例子

```
MODULE main
VAR
    x: boolean;
    y: boolean;
    t: boolean;
    p0 : process aa(x,y,t);
    p1 : process bb(x,y,t);
ASSIGN
    init(x) := 0;
    init(y) := 0;
    init(t) := 0;
SPEC
    (AG !(p0.a=s2 & p1.b=t2))
SPEC
    (AG (p0.a=s1 → AF p0.a=s2))
SPEC
    (AG ((p0.a=s1 & p1.b!=t1 & p1.b!=t2) → !E[!p0.a=s2 U p1.b=t2]))
```

```
MODULE aa(x,y,t)
VAR
        a: {s0,s1,s2,s3,s4};
ASSIGN
        init(a) := s0;
        next(a) := case
                a=s0: s1;
                a=s1 & (x=0 | t=0): s2;
                a=s2: s3;
                a=s3: s0;
                1: a;
            esac;
        next(x) := x;
        next(y) := case
                a=s0 | a=s3: 1;
                a=s2: 0;
                a=s1 & (x=0 | t=0): y;
                1: y;
            esac;
        next(t) := case
                a=s0 | a=s3: 1;
                a=s2: t;
                a=s1 & (x=0 | t=0): t;
                1: t;
            esac;
FAIRNESS
        running
```

```
MODULE bb(x,y,t)
VAR
        b: t0,t1,t2,t3,t4;
ASSIGN
        init(b) := t0;
        next(b) := case
                b=t0: t1;
                b=t1 & (y=0 | t=1): t2;
                b=t2: t3;
                b=t3: t0;
                1: b;
            esac;
        next(y) := y;
        next(x) := case
            b=t0 | b=t3: 1;
            b=t2: 0;
            b=t1 & (y=0 | t=1): x;
            1: x;
            esac;
        next(t) := case
                b=t0 | b=t3: 0;
                b=t2: t;
                b=t1 & (y=0 | t=1): t;
                1: t;
            esac;
 FAIRNESS
        running
```

若将 $p1$ 的

```
 FAIRNESS
        running
```

去掉, 则第二个性质不成立。这是由模型可以一直运行 $p0$ 的某些动作而不改变任何变量的状态造成的。

再将 $p0$ 加上

```
 FAIRNESS
        (x=0 | t=0)
```

或

```
 FAIRNESS
        running
```

改为

```
 FAIRNESS
        a=s2
```

则性质又会成立。

例子

```
MODULE main
VAR
        vv: boolean;
        v0: boolean; v1: boolean; v2: boolean;
        p0 : aa(v0); p1 : bb(v0,v1,v2);
ASSIGN
        init(vv) := 1;
        next(vv) := (v0&v1);
SPEC (AG AF(v2=0))
SPEC (AG AF(v2=1))
SPEC AF (vv=1 & AX v2=0)
SPEC AF (vv=1 & AX v2=1)
SPEC AG (vv=1 & AX v2=1 → AX A[vv=0 U (vv=1 & AX v2=0)])
SPEC vv=1 & AG (vv=1 → AX vv=0 & AX AX AX AX vv=1)
```

```
MODULE aa(v0)
ASSIGN
        init(v0) := 0;
        next(v0) := !v0;
```

```
MODULE bb(v0,v1,v2)
ASSIGN
        init(v1) := 0;
        init(v2) := 0;
        next(v1) := (v0|v1)&(!v0|!v1);
        next(v2) := ((v0&v1)|v2)&(!(v0&v1)|!v2);
```

如果将 aa(v0) 中的 init(v0) 去掉则 v0 的初始值不确定，因此性质 6 不成立。若我们将 v0 的初始值作为性质 6 的前提条件或程序运行的前提条件则性质 6 能成立。具体的做法就是将性质 6 改写成

v0=0 → vv=1 & AG (vv=1 → AX vv=0 & AX AX AX AX vv=1)

§5.4.2 SPIN 的使用

例子

```
bool x,y,t;
mtype = { s0,s1,s2,s3 };
mtype = { t0,t1,t2,t3 };
byte a,b;

active proctype p0()
{
    a=s0;
l01:
    atomic {y=1; t=1; a=s1;}
    atomic {x==0 || t==0; a=s2;}
    atomic {y=0; a=s3;}
    goto l01;
}

active proctype p1()
{
    b=t0;
l11:
    atomic{ x=1; t=0; b=t1;}
    atomic{ y==0 || t==1; b=t2;}
    atomic{ x=0; b=t3;}
    goto l11;
}
```

考虑性质

$$\Box!(a == s2 \&\& b == t2)$$

对应的自动机结构

```
#define p0s2 a==s2
#define p1t2 b==t2
never { /* !([] ! (p0s2 && p1t2 )) */
T0_init:
    if
    :: ((p0s2) && (p1t2)) → goto accept_all
    :: (1) → goto T0_init
    fi;
accept_all:
    skip
}
```

$$\Box(a == s1 \rightarrow \Diamond a == s_2)$$

对应的自动机结构

#define p0s1 a==s1
#define p1t2 b==t2
never { /* !([] (p0s1 → <>p0s2)) */
T0_init:
if
:: (! ((p0s2)) && (p0s1)) → goto accept_S4
:: (1) → goto T0_init
fi;
accept_S4:
if
:: (! ((p0s2))) → goto accept_S4
fi;
}

$\Box((a == s1 \&\& !b == t1 \&\& !b == t2) \rightarrow !(!a == s2 U b == t2))$

对应的自动机结构

#define p0s1 a==s1
#define p0s2 a==s2
#define p1t1 b==t1
#define p1t2 b==t2
never { /* !([]((p0s1 && ! p1t1 && ! p1t2) → !(!p0s2 U p1t2))) */
T0_init:
if
:: (! ((p0s2)) && ! ((p1t1)) && ! ((p1t2)) && (p0s1)) → goto T0_S4
:: (1) → goto T0_init
fi;
T0_S4:
if
:: ((p1t2)) → goto accept_all
:: (! ((p0s2))) → goto T0_S4
fi;
accept_all:
skip
}

性质

$$\Box(a == s0 \rightarrow \Diamond a == s_2)$$

通不过验证

加 weak fairness 则 $\Box(a == s0 \rightarrow \Diamond a == s_2)$ 顺利通过验证。

例子

```
chan r = [4] of {byte};
chan s = [4] of {byte};
byte a;
byte b;
active proctype p0()
{ byte x;
    do
    :: atomic{ if :: x=0; :: x=1; :: x=2; :: x=3; fi; r!(a+x); }
    :: atomic{ s?a; if :: a==20; break; else; fi; }
od
}
active proctype p1()
{ byte y;
do
    :: s!b;
    :: atomic{ r?y; if :: b+1==y; b++; :: b+1!=y; fi; }
od
}
```

性质 $\Box(a \leq b)$ 可以顺利通过验证。而如果我们验证 $\Box(b > a)$ 则会出现问题。

协议建模和验证例子

本节介绍一个文件传输协议的例子。以计算机网络体系结构的 ISO-OSI 参考模型为出发点，我们有物理层、数据链路层、网络层、运输层、会话层、表示层、应用层。一个简单的文件传输协议可涉及表示层、会话层、数据链路层的功能。此外验证的模型还包括文件系统和用户。在这个例子，我们只介绍数据链路层的流量控制功能的设计。首先定义常量和全局变量：

```
#define true 1
#define false 0
#define M 4
#define W 2
#define QSZ 2
mtype = {ack,sync_ack,sync,data}
chan ses_to_flow[2] = [QSZ] of { byte, byte };
chan flow_to_ses[2] = [QSZ] of { byte, byte };
chan dll_to_flow[2] = [QSZ] of { byte, byte };
chan flow_to_dll[2] = [QSZ] of { byte, byte };
```

定义流量控制进程的名称和局部变量。

```
proctype fc(bit n)
{    bool busy[M];
     bool received[M];
     byte q,s,p,m;
     byte window;
     byte type;
     byte I_buf[M],O_buf[M];
     bool x;
```

定义发送步骤前先定义一个初始化动作。

```
#define clean(buffer) s=M; do :: (s>0) ->s=s-1; buffer[s]=false; :: (s==0) ->break; od
```

定义发送步骤。

```
     do
     :: (window<W && len(ses_to_flow[n])>0 && len(f low_to_dll[n])<QSZ) ->
        ses_to_flow[n]?type,x;
        window=window+1; busy[s]=true; O_buf[s]=type;
        if
        :: (type==sync) ->flow_to_dll[n]!type,x; clean(busy); window=0;
        :: (type==sync_ack) ->flow_to_dll[n]!type,x; clean(received);
        :: (type!=sync && type!=sync_ack) ->flow_to_dll[n]!type, s; s=(s+1)%M;
        fi
```

定义接收步骤前先定义以下动作。

```
#define receive() I_buf[m]=type; received[m]=true; received[(m-W+M)%M]=false
#define acked(m) ((0<p-m)&&(p-m<=W)) || ((0<p-m+M)&&(p-m+M<=W))
#define rereceive() if :: acked(m) ->flow_to_dll[n]!ack,m; :: else; fi
```

定义接收步骤。

```
      (do)
     :: dll_to_flow[n]?type,m ->
        if
        :: (type!=ack && type!=sync && type!=sync_ack) ->
           if
           :: (received[m]==false) ->receive();
           :: (received[m]==true) ->rereceive();
           fi
        :: (type==ack) ->busy[m]=false;
        :: (type==sync) ->
           if
           :: (I_buf[q]!=sync) ->I_buf[q]=sync; flow_to_ses[n]!type,m;
           :: (I_buf[q]==sync); flow_to_dll[n]!sync_ack,m;
           fi
        :: (type==sync_ack) ->
           if
           :: (I_buf[q]!=sync_ack) ->I_buf[q]=sync_ack; flow_to_ses[n]!type,m;
           :: (I_buf[q]==sync_ack);
           fi
        fi
```

定义超时和接收完成后的步骤。

```
    :: (window>0 && busy[q]==false) ->window=window-1; q=(q+1)%M;
    :: (received[p]==true &&
        len(flow_to_ses[n])<QSZ && len(flow_to_dll[n])<QS Z) ->
        flow_to_ses[n]!I_buf[p];
        flow_to_dll[n]!ack,p;
        p=(p+1)%M;
    :: (timeout &&
        len(flow_to_dll[n])<QSZ && window >0 && busy[q ]==true) ->
        flow_to_dll[n]!O_buf[q],q;
    od
}
```

fc(0) 和 fc(1) 之间的联系由 flow_to_dll[n] 和 dll_to_flow[1-n] 的连接来完成。可以定义一个进程如下：

```
proctype datalink()
{   byte type, seq;
    do
    :: flow_to_dll[0]?type,seq; if :: dll_to_flow[1]!type,seq :: skip fi;
    :: flow_to_dll[1]?type,seq; if :: dll_to_flow[0]!type,seq :: skip fi;
    od
}
```

流量控制进程的目的是保证一端所接收到的信息和另一端发出的信息完全一致。为了验证这一点，定义了流量控制进程和进程之间的连接之后，我们还需定义测试用的收发信息的进程。由理论得知，在任意长的信息流中只需测试某三类不同信息且其中两类信息只需出现一次，并测试这两类信息另一端发出的顺序与发送的顺序相同即可。这三类信息由 red,white,blue 代表。

这三类信息 red,white,blue 的类型和发送信息的进程定义如下。

```
mtype = {red,white,blue}

proctype test_sender(bit n)
{   byte val;
    ses_to_flow[n]!sync,val;
    do
    :: flow_to_ses[n]?sync_ack,val ->break
    :: timeout ->ses_to_flow[n]!sync,val
    od;
    do :: ses_to_flow[n]!white; :: ses_to_flow[n]!red ->break; od;
    do :: ses_to_flow[n]!white; :: ses_to_flow[n]!blue ->break; od;
    do :: ses_to_flow[n]!white; :: break; od
}
```

变量 val 在这个例子中没有发挥作用，若应用中有多个会话回合，则 val 的值可用于区分不同会话回合。接收信息的进程可定义如下。

```
proctype test_receiver(bit n)
{    byte val;
     flow_to_ses[n]?sync,val;
     ses_to_flow[n]!sync_ack,val;
     do
     :: flow_to_ses[n]?white,val;
     :: flow_to_ses[n]?red,val ->break;
     :: flow_to_ses[n]?blue,val ->assert(0);
     od;
     do
     :: flow_to_ses[n]?white,val;
     :: flow_to_ses[n]?blue,val ->break;
     :: flow_to_ses[n]?red,val ->assert(0);
     od;
     do
     :: flow_to_ses[n]?white,val;
     :: flow_to_ses[n]?red,val ->assert(0);
     :: flow_to_ses[n]?blue,val ->assert(0);
     od
}
```

系统模型的初始化如下。

```
init
{    run datalink();
     run fc(0); run fc(1);
     run test_sender(0); run test_receiver(1);
}
```

以上系统描述可用 SPIN 来验证。在没有做优化的情况下, 用 SPIN.4.1.2 验证以上模型, 约需 8.8 分钟和 900MB 内存。若将所有步骤当做原子动作则验证约需 3.5 分钟和 280MB 内存。系统还可以有其它验证方法。比如我们可定义接收信息的进程如下

```
proctype test_receiver(bit n)
{    byte type,val;
     flow_to_ses[n]?sync,val;
     ses_to_flow[n]!sync_ack;
     do
     :: flow_to_ses[n]?type,val;
     od;
}
```

然后验证系统是否满足

$$!([]z||(zU([]w||(wU(r)))))$$

其中 z, w, r, b 的定义如下：

z test_receiver:type==0
w test_receiver:type==white
r test_receiver:type==red

当然这公式描述的只是接收到的信息的顺序关系, 不能验证重复接收和丢包问题。将所有步骤当做原子动作则验证约需 8.5 分钟和 330MB 内存。

§5.4.3 VERDS 的使用