

谓词抽象

前言

本篇将介绍谓词抽象（Predicate Abstraction）算法。它是程序验证的经典算法之一，属于上近似算法，在证明程序正确性方面有着独到的优势。同时，由于它也分析程序的执行路径，所以也可以用于寻找错误路径。一般而言，谓词抽象不需要额外提供不变式来处理循环，而且谓词分析完成正确性验证后的结果，可以用于生成不变式。在本篇中，我们将较为细致地介绍谓词抽象的基本方法，并给出一个综合验证案例来辅助讲解。本篇篇幅很长，内容很多，也较为详细，希望读者耐心、认真、反复阅读。

谓词抽象

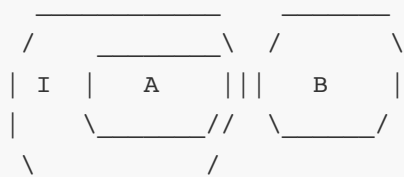
在介绍谓词抽象算法之前，我们先介绍少量非常必要的预备知识。其中最核心的就是插值（Interpolant）的计算。

Craig插值

给定公式 A 和公式 B ，倘若公式 $A \wedge B$ 是不可满足的。那么我们就可以计算公式 A 和公式 B 的克雷格插值（Craig Interpolant，在下文中我们称为**Craig插值**），得到一个新的公式 I ，满足：

- $A \Rightarrow I$ 有效，即 $A \wedge \neg I$ 不可满足
- $I \wedge B \Rightarrow false$ ，即 $B \wedge I$ 不可满足
- I 仅含有公式 A 和公式 B 公共的符号，以及理论本身的符号

```
1. A & B = false
2. A => I = true    3. I => B = false
```



从某个角度来说， A 和 B 的插值 I 所表达的，是 A 的一些关键性质，正是因为这些性质，才使得 A 和 B 相矛盾。插值 I 抽象掉了（Abstract Away） A 中与导致矛盾无关的部分。所以 I 实际上是对 A 一种泛化（Generalization）。以程序分析为例，倘若 A 表示一条路径约束， B 表示某一属性，那么 $A \wedge B$ 不可满足表示该路径中属性不可达。那么我们可以通过计算 A 和 B 的插值 I ，使用 I 来表示和该路径有相似原因导致属性不可达的路径集合，从而实现泛化，使我们能分析更多的路径。

插值计算

我们可以使用SMT求解器来计算插值。目前主流的SMT求解器中，对Craig插值计算支持得比较好的有MathSat，以及SMTInterpol。Z3的Release版本未支持计算插值，但是unstable版本中iZ3分支支持插值计算。[在线版iZ3](https://rise4fun.com/iZ3)具有插值计算功能，我们可以直接使用。

iZ3 Online: <https://rise4fun.com/iZ3>

我们以如下的SMT-LIB公式为例，简要介绍以下SMT-LIB中计算插值的语法。首先使用 `(set-option :produce-interpolants true)` 开启插值计算。使用 `(assert (! A :named name))` 的形式，添加一个名字（named:）为 name 的公式 A。下面的例子中，我们添加了 f1 和 f2 两个公式。调用SMT求解 `(check-sat)` 后，若结果为**不可满足**（UNSAT），则可继续调用 `(get-interpolant f1 f2)` 计算 f1 和 f2 的插值。

```
(set-option :produce-interpolants true)
(declare-const a Int) (declare-const b Int)
(declare-const c Int) (declare-const d Int)
(assert (! (and (= a b) (= a c)) :named f1))
(assert (! (and (= b d) (not (= c d))) :named f2))
(check-sat)
(get-interpolant f1 f2)
```

将以上公式交给在线版iZ3求解后，返回：

```
unsat
(= b c)
```

显然，插值产生的新谓词 `(= b c)` 满足插值结果所需满足的几条要求。

归纳序列插值

解释完**二元插值**后，我们介绍**归纳序列插值**（Inductive Sequences of Interpolants）。归纳序列插值指，给定一个谓词公式序列 F_1, F_2, \dots, F_n ，使得 $F_1 \wedge F_2 \wedge \dots \wedge F_n$ **不可满足**，我们可以算得一系列的谓词公式 I_1, I_2, \dots, I_{n-1} ，满足：

- $F_1 \Rightarrow I_1$
- $I_k \wedge F_{k+1} \Rightarrow I_{k+1}$
- $I_{n-1} \wedge F_n \Rightarrow \text{false}$ ，即不可满足。
- I_k 仅含有 F_1, F_2, \dots, F_{k-1} 和 F_1, F_k, \dots, F_n 的公共符号，以及理论本身的符号。

实际上，从以上的几点要求中，我们可以推导出 $F_1 \wedge \dots \wedge F_{n-1} \Rightarrow I_{n-1}$ 。所以 I_{n-1} 可以看作是 $F_1 \wedge \dots \wedge F_{n-1}$ 与 F_n 进行二元插值的结果。而**更一般的**，我们可以推导出 $F_1 \wedge \dots \wedge F_k \Rightarrow I_k$ ，且 $I_k \wedge F_{k+1} \wedge \dots \wedge F_n \Rightarrow \text{false}$ 。所以，所以 I_k 可以看作是 $F_1 \wedge \dots \wedge F_k$ 与 $F_{k+1} \wedge \dots \wedge F_n$ 进行二元插值的结果。

我们可以通过命令 `(get-interpolants F1 F2 ... Fn)` 来获取词公式序列 F_1, F_2, \dots, F_n 的插值。以如下的公式为例，

```
(set-option :produce-interpolants true)
(declare-const a Int) (declare-const b Int)
(declare-const c Int) (declare-const d Int)
(declare-const e Int)
(assert (! (and (= a b) (= a c)) :named f1))
(assert (! (= c d) :named f2))
(assert (! (and (= b e) (not (= d e))) :named f3))
(check-sat)
(get-interpolant f1 f2 f3)
```

将其交给在线版iZ3求解后，返回：

```
unsat
(= b c)
(= b d)
```

其中，对应 I_1 为 $b = c$ ， I_2 为 $b = d$ 。在接下来**谓词抽象**的例子中，我们会使用到**归纳序列插值**。

谓词抽象

我们首先需要定义**抽象域**，例如符号抽象的 $\{+, -, 0, \top, \perp\}$ ，区间抽象的 $[a, b]$ 。然后在程序的控制流节点上，打上对应**抽象域中的元素的标记**（Label），以刻画程序执行过程中在对应**位置**（Location）上的**信息**（Data）。符号抽象的标记的是当前位置程序变量值的符号，而区间抽象标记的是当前位置程序变量值所在的区间。我们使用标记来**抽象**程序的实际运行时在每个位置上的**状态空间**，将每个位置的标记称为对应位置的**抽象状态**（Abstraction State）。我们沿着程序控制流图，仅在**当前抽象状态**的基础上，计算执行后继程序代码后，对应的**下一位置的抽象状态**，即**抽象后继**（Abstraction Successor）。整个计算过程都在抽象域所定义的抽象空间中进行，而不涉及程序的实际运行状态空间，从而能极大地提高计算效率。

谓词抽象也可以看作是**抽象解释**框架下的一种分析方法。它的抽象域是一个**谓词集合**（Predicate Set）。我们会在程序的每个控制流节点上，标记上在对应位置**有效的**谓词组合。例如谓词集合是 $\{a > 0, b > 0\}$ ，而在某个程序位置，我们能得出 $a = 1, b = -1$ 。那么，我们会在该位置标记上 $\langle a > 0, \neg b > 0 \rangle$ 。

在程序验证领域，我们说的**谓词抽象**一般指带有**反例制导的抽象精化**（Counter-example Guided Abstraction and Refinement, CEGAR）的谓词抽象。它是在抽象解释框架下的谓词抽象算法后，加入了使用**伪反例**（Suspicious Counter-example）制导的**精化**（Refine）步骤。通过**抽象-精化**两个步骤的不断迭代，逐步**增加**谓词，**细化抽象粒度**，直到在当前的抽象域下，能证明对应的属性，或找到一条**真反例**路径（错误路径）。

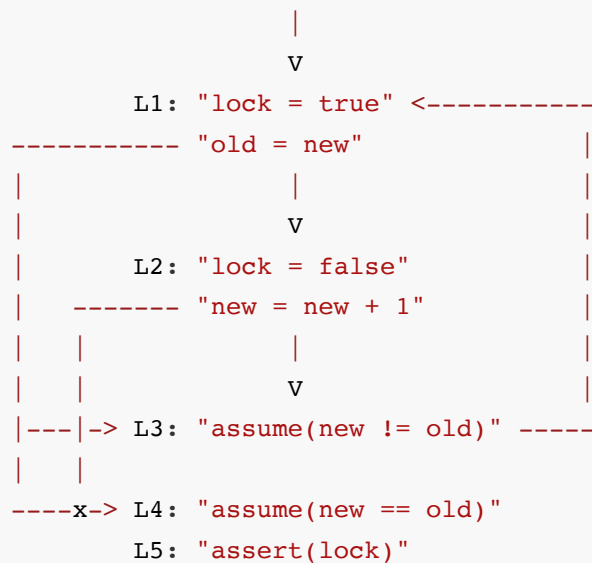
谓词抽象综合案例

我们使用以下的例子，来详细讲解谓词抽象的流程。在如下的C语言程序中，有一个 `do-while` 循环，在循环体内，包含对 `Bool` 类型变量 `lock` 的操作，并使用整形变量 `new` 和 `old` 来维护一些操作的信息。其中的 `if(*)` 表示非确定的条件语句，`*` 可以理解成一个随机的 `Bool` 变量，每次程序运行到这一行时，它的值可能为 `true`，也可能为 `false`。我们需要验证，在退出循环后，`lock` 变量的值为 `true`。

```
do {
    lock = true;
    old = new;
    if(*) {
        lock = false;
        new = new + 1;
    }
} while(new != old)
assert(lock);
```

首先，我们将以上源代码转为如下的控制流图，并在每个**基本代码块**（Basic Block，也称**基本块**）的头部区域，标记上位置编号，如 `L1`、`L2` 等等。`L5` 位置对应属性的**断言**，我们也给出一个单独的标记 `L5`。

Control-flow Graph:



事实上，我们可以每条语句前都给一个位置编号，也可以每隔 `N` 条语句给出一个位置编号。不同的编号策略，会对抽象后继的计算**次数**以及**效率**都会产生影响。为了便于讲解，我们仅在**控制流结点**处给出位置编号。

第一轮分析

接下来我们尝试模拟执行谓词抽象算法。一般而言，在没有额外的信息的情况下，谓词抽象会以空集 \emptyset 作为**初始的谓词集合**来进行分析。但为了使我们的分析过程更加自然易懂，我们选择将属性断言中的谓词 $lock$ ，加入初始的谓词集合。所以，我们从 $\{lock\}$ 开始，进行分析。

如下所示，我们从 L1 位置开始进行分析。由于变量的初始值一般假定为未知，所以在 L1 位置，无法判定谓词 $lock$ 的真假，所以我们**使用谓词 $true$ 表示 $lock$ 真假都有可能**。接下来，经过 $lock = true$ 和 $old = new$ 两条赋值语句，计算抽象后继状态。我们可以很自然地看出，后继抽象状态应该是 $\{lock\}$ 。因为经过 $lock = true$ ，谓词 $lock$ 成立，而谓词 $\neg lock$ 不成立。但是，我们需要全自动的计算方式。

```
Initial Predicate Set: {lock}

      L1: <true>
      "lock = true" |
      "old = new"  |
      _ _         |
      / | \ _ _    (L3... todo)
(todo ...L4) _ _ / |
      L2: <lock>
      "lock = false" |
      "new = new + 1" | _ _
      | \ _ _        (L3... todo)
      |
      L4: <! $lock$ >
      "assume(new == old)" |
      L5: <! $lock$ >
      "assert(lock)" |
      X <--- "error!"
```

回顾在符号抽象和区间抽象中，我们对程序中的每种操作，都定义了对应的抽象后继计算方式。在谓词抽象中，也需要有这样的定义。我们使用霍尔逻辑来实现抽象后继自动计算。对于谓词集合中的每个谓词，我们需要分别判断两种情况。例如，在 L1 的后继位置，我们需要判断是谓词 $lock$ 成立，对应霍尔三元组 $\{true\}lock = true; old = new\{lock\}$ 成立；抑或谓词 $\neg lock$ 成立，对应霍尔三元组 $\{true\}lock = true; old = new\{\neg lock\}$ 成立。我们通过计算**最弱前置条件**的方式，来判断对应的霍尔三元组成立与否。

$$\begin{aligned} & wlp(lock = true; old = new, lock) \\ &= wlp(lock = true, wlp(old = new, lock)) \\ &= wlp(lock = true, lock) \\ &= true \end{aligned}$$

由于前置条件 $true \Rightarrow true$ （最弱前置条件），所以霍尔三元组 $\{true\}lock = true; old = new\{lock\}$ 成立。抽象后继状态下谓词 $lock$ 成立。对于另一种情况，

$$\begin{aligned}
& wlp(\text{lock} = \text{true}; \text{old} = \text{new}, \neg \text{lock}) \\
&= wlp(\text{lock} = \text{true}, wlp(\text{old} = \text{new}, \neg \text{lock})) \\
&= wlp(\text{lock} = \text{true}, \neg \text{lock}) \\
&= \neg \text{true} = \text{false}
\end{aligned}$$

前置条件 true 无法蕴含对应的最弱前置条件 false ，所以抽象后继状态 $\neg \text{lock}$ 不成立。

综上，我们算得 L1 位置的抽象后继状态为 lock 。

从控制流图中，我们可以发现，L1 的后继位置有 L2、L3 和 L4，我们先看 L2。在对应的后继位置 L2 上标记后继状态 $\langle \text{lock} \rangle$ ，再重复执行以上的抽象后继计算，得到 L2 的抽象后继状态为 $\neg \text{lock}$ 。从控制流图有，L2 的后继位置有 L3 和 L4，假设我们先看 L4。在 L4 上标记 $\langle \neg \text{lock} \rangle$ ($\langle !\text{lock} \rangle$)，继续计算抽象后继仍得 $\neg \text{lock}$ 。L4 的后继位置为 L5，在 L5 上标记 $\langle \neg \text{lock} \rangle$ ($\langle !\text{lock} \rangle$)。由于 L5 是属性断言所在的位置，所以我们需要特殊分析。我们发现 L5 位置的抽象状态 $\langle \neg \text{lock} \rangle$ 不能使断言 $\text{assert}(\text{lock})$ 成立，即 $\neg \text{lock} \Rightarrow \text{lock}$ 不成立。所以，我们至此所探索的路径，L1 \rightarrow L2 \rightarrow L4 \rightarrow L5，是一条抽象状态空间下的**错误路径**（Error Path），也称**反例**（Counter-example）。

我们将以上从程序入口开始，沿着程序控制流计算抽象后继状态所形成的树状结构称为**抽象可达树**（Abstract Reachable Tree, ART）。倘若某些后继位置形成了环，我们则将其称为**抽象可达图**（Abstract Reachable Graph, ARG）。

反例分析

注意到上近似只能用于证明正确性，而下近似只能用于找错。谓词抽象状态空间是上近似空间，所以在抽象空间找出的错误路径**不一定准确**，可能并不属于程序的实际行为。因此，我们需要进行**反例分析**（Suspicious Counter-example Analysis），以判断我们是否找到了一条**真实的**错误路径。

抽象空间中的“错误路径”L1 \rightarrow L2 \rightarrow L4 \rightarrow L5 对应的实际路径为 $\text{lock} = \text{true}; \text{old} = \text{new}; \text{lock} = \text{false}; \text{new} = \text{new} + 1; \text{assume}(\text{new} == \text{old}); \text{assert}(\text{lock});$ 。我们需要判断**违背属性断言**的错误路径是否**可达**（Feasible），所以需要将最后的**属性断言** $\text{assert}(\text{lock})$ 转为**违背断言的假设** $\text{assume}(!\text{lock})$ 。如果对应的路径公式可满足，则表示路径可达。否则，路径不可达，我们所找到的是一条**伪反例**。我们编码**路径公式**（Path Formula），得到 $\text{lock} = \text{true} \wedge \text{old} = \text{new} \wedge \text{lock}_1 = \text{false} \wedge \text{new}_1 = \text{new} + 1 \wedge \text{new}_1 = \text{old} \wedge \neg \text{lock}_1$ 。将其交给SMT求解器，返回**不可满足**（UNSAT），即实际路径**不可达**（Infeasible）。因此，我们找到的 L1 \rightarrow L2 \rightarrow L4 \rightarrow L5 实际上是一条抽象空间里的**伪反例**，不是程序的实际执行路径。

由于伪反例的存在，导致我们**无法使用当前的抽象空间，来证明程序的正确性**。这代表我们**抽象得太过了**，我们需要**依据得到的伪反例**，对当前抽象空间进行**修正**，使其抽象粒度变小，从而**排除伪反例**。我们称这一修正过程称为**精化**（Refined），更进一步而言，**反例制导的精化**（Counter-example Guided Refined, CEGAR）。对于谓词抽象而言，由于其抽象空间是由**谓词集合**生成的，所以其精化的方式就是在**谓词集合**中加入**新谓词**。而我们本篇开头所介绍的Craig插值，恰好可以从**不可满足**的公式组合中，得到新的谓词。

将以上路径公式编码为以上SMT-LIB表达式。我们将不同位置的语句对应的公式分组到不同的公式块里。最后进行一次 L1 \rightarrow L2 \rightarrow L4 \rightarrow L5 的序列插值。

```
(set-option :produce-interpolants true)
(declare-const lock Bool) (declare-const lock1 Bool)
(declare-const old Int)
(declare-const new Int) (declare-const new1 Int)
(assert (! (and (= lock true) (= old new)) :named L1))
(assert (! (and (= lock1 false) (= new1 (+ new 1))) :named L2))
(assert (! (= new1 old) :named L4))
(assert (! (not lock1) :named L5))
(check-sat)
(get-interpolant L1 L2 L4 L5)
```

iZ3在线版给出的插值结果如下：

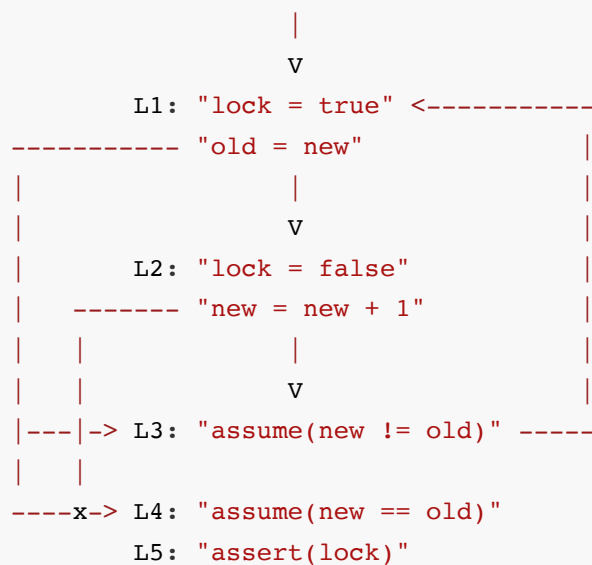
```
unsat
(= old new)
(not (= old new1))
false
```

插值给出了新的谓词 $old = new$ 。注意， $false$ 实际上是 $\neg true$ ，而谓词 $true$ 我们已经使用了。

第二轮分析

依据插值的结果，我们可以在谓词集合中添加新谓词 $old = new$ ，新的谓词集合变为 $\{lock, new = old\}$ 。

Control-flow Graph:



我们将同样的控制流图展示在上方，便于阅读。我们在精化后的抽象空间中再次进行**抽象可达树**的计算。同样，**L1**位置对应的抽象状态为 $\langle true, true \rangle$ 。我们使用之前相同的方式来进行抽象后继状态的计算，可得**L1**的后继抽象状态为 $\langle lock, new = old \rangle$ 。**L1**的后继位置有**L2**、**L3**和**L4**，我们先看**L2**，在**L2**位置标记上算得的**L1**的抽象后继状态 $\langle lock, new = old \rangle$ 。接着，我们计算**L2**的抽象后

继状态。按照之前的计算结果，后继状态中谓词 $lock$ 应该取 $\neg lock$ 。所以我们接着计算谓词 $new = old$ 在后继状态上的取值，先看霍尔三元组 $\{new = old\}lock = false; new = new + 1\{new = old\}$ 是否成立。

$$\begin{aligned} & wlp(lock = false; new1 = new + 1, new1 = old) \\ &= wlp(lock = false, wlp(new1 = new + 1, new1 = old)) \\ &= wlp(lock = true, new + 1 = old) \\ &= (new + 1 = old) \end{aligned}$$

由于蕴含式 $new = old \Rightarrow new + 1 = old$ 不成立，所以后继状态上 $new = old$ 不成立。再看 $\neg new = old$ 。

$$\begin{aligned} & wlp(lock = false; new1 = new + 1, \neg new1 = old) \\ &= wlp(lock = false, wlp(new1 = new + 1, \neg new1 = old)) \\ &= wlp(lock = true, \neg new + 1 = old) \\ &= (new + 1 \neq old) \end{aligned}$$

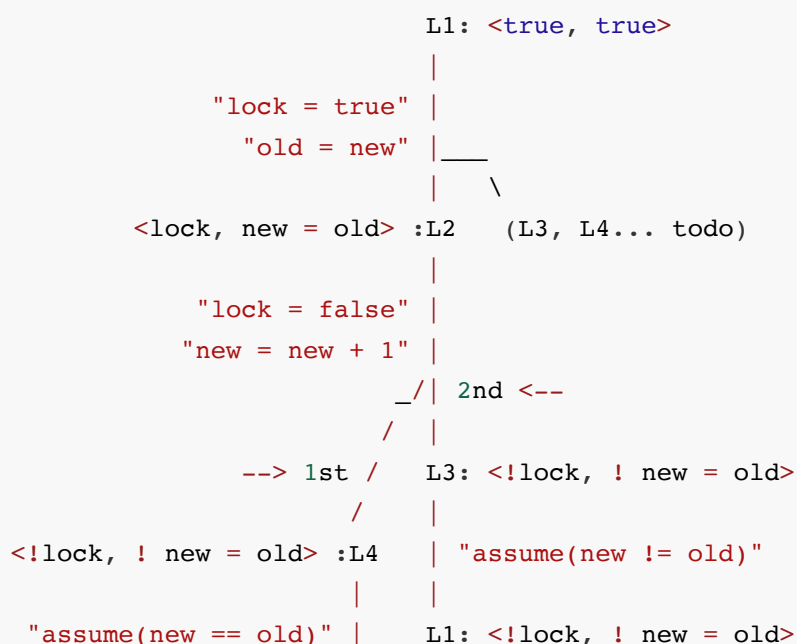
而蕴含式 $new = old \Rightarrow new + 1 \neq old$ 成立，所以我们算得 L2 的抽象后继状态上 $\neg new = old$ 成立。故 L2 的抽象后继状态为 $\langle \neg lock, \neg new = old \rangle$ 。

接着，我们先看 L2 的后继位置 L4，再看后继位置 L3。在 L4 位置打标 $\langle \neg lock, \neg new = old \rangle$ 。我们发现 L4 的后继状态为**不可达** ($(\neg new = old \wedge new = old) = false$)，故程序执行路径在此**阻塞** (Blocked)，故我们无需再往后计算。

再看后继位置 L3，算得其后继状态仍为 $\langle \neg lock, \neg new = old \rangle$ 。而其后继位置为 L1，其原本的抽象状态 $\langle true, true \rangle$ 已经**包含** (Covered, 也称**覆盖**) 了新算得的抽象状态 $\langle \neg lock, \neg new = old \rangle$ 。所以这条环路的计算已经收敛，达到不动点，我们也无需继续计算。

倘若当前计算得的抽象后继状态**未被**之前的 L1 原本的抽象状态**覆盖**，我们则生成一个新的 L1' 结点，继续沿着控制流图往后进行计算，并将计算结果以树状结构展开，直到计算路径被**阻塞**，或者被之前的同名结点**覆盖**。

Refined Predicate Set: {lock, new = old}




```

      |      |
blocked! --> o      o <-- covered!

```

接着回溯到 L1 位置的其他后继位置 L3 和 L4。我们将同样的控制流图展示在下方，便于阅读。先看 L3，在 L3 位置打标上 L1 的抽象后继状态 $\langle lock, new = old \rangle$ 。同样，我们发现 L3 的后继状态为不可达 $(new = old \wedge \neg new = old) = false$ ，故无需继续往后计算。

Control-flow Graph:

```

      |
      v
L1: "lock = true" <-----
----- "old = new" |
|           |
|           v
|       L2: "lock = false" |
|       ----- "new = new + 1" |
|       |           |
|       |           v
|---|> L3: "assume(new != old)" -----
|       |
|---x-> L4: "assume(new == old)"
|       |
|       L5: "assert(lock)"

```

再看后继位置 L4，算得其后继状态仍为 $\langle lock, new = old \rangle$ ，标在其后继位置 L5。我们发现 L5 位置的抽象状态 $\langle lock, new = old \rangle$ 满足断言 `assert(lock)`。所以这条路径是安全的 (Safe)。

Refined Predicate Set: {lock, new = old}

```

      L1: <true, true>
      "lock = true" |
      "old = new" |
      | _
      (done ...L2) ___/ | \
      --> 2nd | | \
      | | \ 1st <--
      | | \
      <lock, new = old> :L4 | \
      | | |
      "assume(new == old)" | L3: <lock, new = old>
      | | |
      <lock, new = old> :L5 | "assmue(new != old)"
      | | |
      "assert(lock)" | o <-- blocked!
      |
      safe! --> o

```

至此，我们在抽象空间上，分析完了控制流图中所有的路径。由于没有找到错误路径，这说明在当前的抽象状态空间中，我们**证明了断言属性成立**。

以上便是**基于反例制导抽象精化的谓词抽象算法**的全部基本流程。我们从谓词集合 $\{lock\}$ 开始，在抽象空间找到一条错误路径。接着，我们进行了**错误路径分析**，发现其为一条**伪反例**。我们基于这条伪反例进行**精化**，使用**插值**算得新的谓词，并加入谓词集合，得到 $\{lock, new = old\}$ 。在精化后的抽象空间上，我们**未找到**任何错误路径。所以，我们**证明了程序的正确性**。

通过第二轮谓词分析，完成验证后，我们可以发现，在第二轮分析所得到的**抽象可达图**上，每个位置上打标的**抽象状态**，实际上也可以作为**当前位置的不变式**，它近似了程序执行到当前位置的实际状态空间。

基于插值的精化解释

其实，至此我们还留有一个问题未解决。我们之前讲过，**基于反例制导抽象精化**，在精化完成后，需要能够在精化后抽象空间中**消除对应的伪反例路径**。通过向谓词集合中加入**插值生成的新谓词**，我们能够达到这一目标么？

我们注意到，在第一轮的分析中，我们从 L4 位置的抽象状态 $\langle \neg lock \rangle$ ，计算其后继抽象状态 $\langle \neg lock \rangle$ ，并标在 L5 位置上。从而发现了一条到达 L5 位置，且违背断言的错误路径。

```

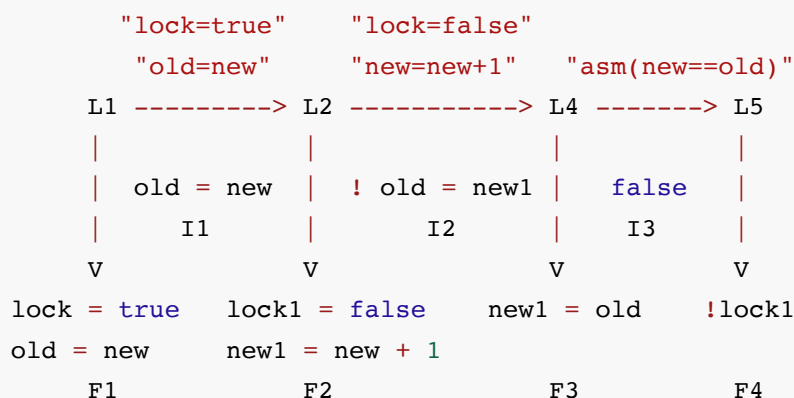
                                L1: <true>
                                |
    "lock = true"               |
                                |
    "old = new"                 |
                                |
                                L2: <lock>
                                |
    "lock = false"              |
                                |
    "new = new + 1"             |
                                |
                                L4: <!\lock>
    "assume(new == old)"        |
                                |
                                L5: <!\lock>
    "assert(lock)"              |
                                |
                                X <--- "error!"
```

通过分析程序实际的路径 L1 -> L2 -> L4 -> L5，我们发现违背断言的**路径公式**

$lock = true \wedge old = new \wedge lock_1 = false \wedge new_1 = new + 1 \wedge new_1 = old \wedge \neg lock_1$ 不可满足，这说明这条路径**不可达**。

我们发现，在**抽象空间**里，L5 可达且违背断言，是因为我们仅从 L4 位置的抽象状态 $\langle \neg lock \rangle$ 出发进行计算。而**实际空间**里，路径 L1 -> L2 -> L4 -> L5 不可达，是因为我们计算了完整的路径公式（约束）。

为了解释添加插值产生的新谓词的作用，我们将插值计算的结果展示如下。我们注意到横向上方的代码，与箭头下方的路径公式一一对应，如 `lock=true; old=new` 对应 $F_1 \triangleq lock = true \wedge old = new$ 。



回顾一下归纳序列插值所需要满足的几个特点：

- $F_1 \Rightarrow I_1$
- $I_k \wedge F_{k+1} \Rightarrow I_{k+1}$
- $I_{n-1} \wedge F_n \Rightarrow false$

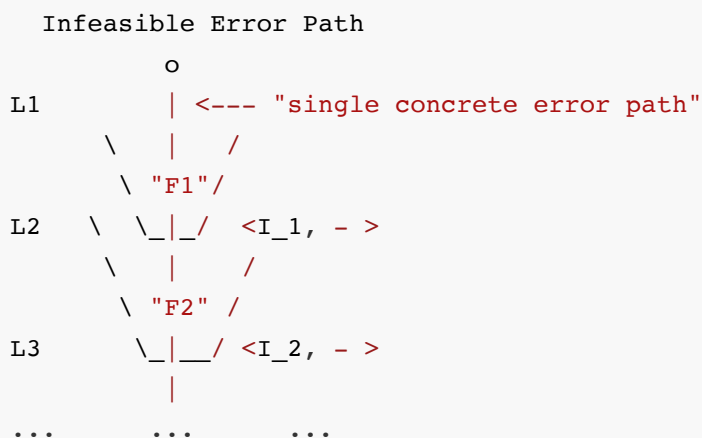
我们来模拟一下，添加了新谓词后，在该路径上的抽象后继的计算过程。为了方便起见，我们不再使用霍尔三元组以及最弱前置条件计算来求抽象后继状态，而是使用直观解释法。

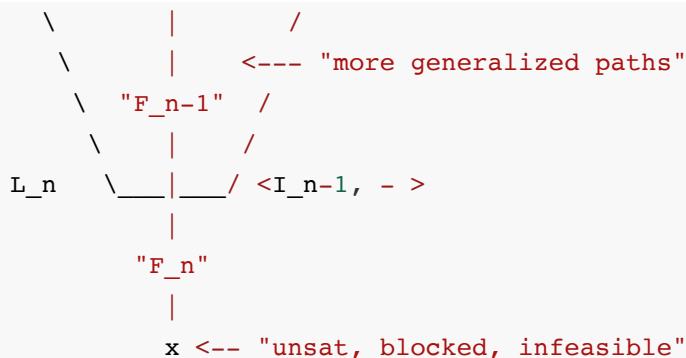
首先有 $F_1 \Rightarrow I_1$ ，也就是说，从 L1（抽象状态为 $\langle true, true \rangle$ ）位置出发，经过代码 `lock=true; old=new`（路径约束为 $F_1 \triangleq lock = true \wedge old = new$ ）后，可以得到 I_1 （ $old = new$ ）成立。这也就是说，L1 位置的后继位置 L2 的抽象状态一定是 $\langle -, old = new \rangle$ ，其中 $-$ 为通配符，表示未确定。

再由 $I_1 \wedge F_2 \Rightarrow I_2$ ，同理可以得到 L3 位置的抽象状态一定是 $\langle -, \neg old = new \rangle$ （ I_2 成立）。继续往后，由 $I_2 \wedge F_3 \Rightarrow I_3$ ，同理可得 L4 位置的抽象状态一定是 $\langle -, false \rangle$ （ I_3 成立）。注意到此时我们已经得到目前的路径不可达。倘若此时仍有路径可达，那么依据 $I_3 \wedge F_4 \Rightarrow false$ ，我们可以得出后续的路径一定是不可达的。

所以，再将插值得到的新的谓词公式加入谓词集合后，我们就可以保证，在伪反例路径上的抽象后继状态计算，最终会在某个位置 Li 上得到 *false*（不可达），从而在精化过后的抽象空间里，消除了对应伪反例。同时，如下所示，由于相对于具体路径形成的路径约束 $F_1 \wedge \dots \wedge F_n$ ，由插值公式组成的抽象路径约束 $\langle I_1, - \rangle, \langle I_2, - \rangle, \dots, \langle I_{n-1}, - \rangle$ 包含了更多的程序执行路径（Execution）。所以，使用插值进行精化，实际上不仅仅在新的抽象空间里消除了伪反例路径，而且还消除了和伪反例路径有着相同不可达缘由的其他路径，从而实现了泛化。

Generalization by Sequence of Interpolant:





实际上，从以上的分析过程中，我们可以发现，仅需要将插值得到的谓词序列，**一一对应依次添加**到各个位置上，就可以消除伪反例。而不需要将所有新谓词，全部加入到全局的谓词集合。也就是说，我们可以在**不同的位置使用不同的谓词集合**来形成抽象状态，而不需要一个**全局的**谓词集合。这样做可以显著减少谓词的数量，以及对于每个谓词，抽象后继的计算次数。同时，以这样的方式进行的精化，不会影响**已经计算好的**抽象可达树中到**不包含错误路径**的部分。因为这部分的结点上，谓词集合没有发生改变，从而抽象状态和可达性也都不会发生改变。这也就是说，使用这样的方法，我们可以进行**局部的**（Localized）、**增量式的**（Incremental）抽象可达树计算，从而大大提高谓词抽象算法的效率。这种优化方式，被称为**惰性抽象**（Lazy Abstraction）。

小结

在本篇中，我们较为完整细致地介绍了带有**反例制导的抽象精化的谓词抽象**算法。从算法的流程中，我们可以看出，谓词抽象也是一种基于SMT的程序验证算法。在计算抽象后继时，我们需要使用SMT表达式来判定蕴含关系，即**当前抽象状态蕴含抽象后继的最弱前置条件**。同时，在进行反例分析时，我们也需要使用SMT来对反例的路径约束进行求解。若反例为**伪反例**，还需要使用SMT的插值功能来寻找新谓词，用于**精化**。

其实，关于谓词抽象算法，还有很多进一步优化的空间。比如说，我们可以获取程序中的谓词（`if`语句和 `while` 语句的条件）来作为初始的谓词集合，以减少分析的迭代次数。我们也可以使用一些轻量级的不变式生成技术（区间分析），来生成辅助不变式作为谓词，以加快迭代。事实上，目前已有大量的研究工作来优化谓词抽象，其中最为知名的应该是**惰性抽象**（Lazy Abstraction）。此外，还有**大块编码**（Large Block Encoding）以及**可调块编码**（Adjustable Block Encoding）等等。