摘要

类似C语言等拥有控制语句的高级语言，在自动修复的过程中推理其复杂的数据依赖和控制关系往往是比较困难的。通过将C语言程序转换为布尔程序，本文得以完整模拟布尔程序，收集所有使程序出错的错误路径并推导出正确的布尔修复，并最终转换为修复后的C语言程序，使得整个C语言程序的修复过程完全自动化。文末，我们给出了根据这些方法实现的C语言程序修复工具的相关设计，并采用了TCAS测试集来验证工具的正确性及有效性。

**关键词**：代码修复、布尔程序、逆向转换

第一章 前言

调试是发现和减少计算机程序或电子硬件中的错误与缺陷的有序过程，以使其按照预期的行为执行。计算机程序调试的过程包括错误定位和代码修复两个阶段。近年来的研究主要提出了基于形式化规范与基于测试用例两种代码修复方式，但在实际应用中，对于使用像C语言等具有控制结构的程序，对其复杂的数据依赖和控制关系进行推理往往是比较困难的。

布尔程序是Thomas Ball和Sriram K. Rajamani在2000年提出的一种软件分析模型[1]，它有着类似于C语言的控制结构和命令语句：有函数和递归、全局变量和局部变量。主要的区别在于，布尔程序的所有变量都是布尔类型的，同时它还支持断言、并行赋值、布尔变量不确定性等。布尔程序的这些特性使得它适合用于作为代码修复过程的中间语言。过去针对代码修复的研究中就有学者利用了布尔程序的特性，而且从实验结果上看，这种基于控制结构的代码修复十分有效。遗憾的是，其所提出的修复方法尚未能完全自动化。

本文在布尔程序修复方式上根据前人的工作做了进一步的研究和改进。本文根据控制结构对布尔程序代码进行模拟，收集所有会使程序进入错误状态的错误路径，并推导出正确的布尔修复。同时，本文提出了布尔程序修复的逆向转换方法，将得到的布尔修复逆向转换为C语言的表达式，从而使得整个C语言程序的修复过程完全自动化。

本文还给出了根据该修复方法和逆向转换方法设计的一个布尔程序修复逆向转换工具：通过给定待修复的C语言程序以及出错的具体位置，该工具能将其转换为布尔程序并进行修复，再把布尔程序的修复结果逆向转换为C语言的表达式，最终得出修复后的C语言程序。最终，我们采用TCAS测试集来验证工具的正确性及有效性，实验结果也显示该工具能够找到正确的修复结果，经过转换化简的修复表达式也更贴近原程序的语义。

本文的结构如下。第二章讲解布尔程序的语义语法及C语言程序到布尔程序的转换规则。第三章给出推导正确的布尔修复并将其转换为C语言表达式的方法。第四章将展示布尔程序修复工具的基本设计，并在第五章给出具体的测试结果。

第二章 布尔程序基础

本章主要介绍布尔程序的语法语义，以及针对布尔程序的模型检测方法，同时还阐述了C语言程序转换为布尔程序的具体过程，为后文布尔程序的应用提供理论基础。

## 2.1 语法和语义

布尔程序是对一般的程序应用谓词抽象之后产生的结果。布尔程序的变量既可以是全局变量，也可以是局部变量。在布尔程序中，所有的变量均是布尔类型的，因此在声明变量时无须再给出变量的类型。

布尔程序中只包含两个常量：0（假）和1（真），同时布尔变量的取值还可以是“不确定”。布尔程序中的语句有着与C语言语句类似的结构，同样允许为某个语句赋予标签。同时，布尔程序还有着与Python类似的并行赋值的特性，允许同时将一组值赋给一组变量。例如简单的一句即可互换变量a和b的取值。在布尔程序中有三种语句会影响到程序的控制流程：if、while和assert。这三种语句的谓词是决策性的，可被用来进行非确定性建模。

在此，我们给出布尔程序的形式化表示方法。

首先，我们定义布尔程序的变量的集合为，其中包括了布尔程序的全局变量及局部变量。我们将估值 定义为程序在某一时刻所能访问到的值为1（真）的变量，并可得出的幂集。

由此，我们便可将布尔程序*B*的控制流图视为一个确定型有穷自动机，它包括了：

* 一个有穷的语句集合，其中语句的下标对应着语句在布尔程序中的行号。
* 一个有穷的状态集合，其中每个状态由一条语句和一个估值组成，即，代表着程序当前即将执行语句，且所有变量的当前取值由给出。
* 一个转移函数，以一个状态作为变量，返回一个状态。我们将转移函数记作，如果我们有状态，那么是这样一个状态，使得程序以估值执行语句后将进入语句，且估值为，即在非形式化表示自动机的图中，存在从到的箭弧。
* 一个初始状态，是中的状态之一。

上述的定义给出了一般布尔程序的形式化定义，其中包括了语句执行所代表的状态转移。但若所模拟的语句是对某个函数的调用，情况则会相对复杂一些。如果进入函数的话，我们可以认为函数调用语句的下一个语句是函数体的第一条语句；若不进入函数，直接假设函数执行完成，我们也可以认为函数调用语句的下一个语句即为其下一条语句。

对此，我们使用另一个自动机来模拟被调用的函数，且该自动机的初始状态将由调用语句的状态所决定。在非形式化的图像表示中，我们用虚线箭头表示自动机间的转移，即调用边。形式化地说，假设调用的自动机为，被调用的自动机为，则调用边。该映射将实际参数赋值给形式参数，全局变量保持不变，除形式参数外的被调用函数的局部参数将被赋值为“不确定”。此外，我们用函数来计算调用返回后的估值。它从被调用函数的估值里拷贝全局变量的值，从被调用前的函数里拷贝局部变量的值，并将的返回值赋予由赋值语句指定的变量。

在图2-1中，我们给出了一个相关的例子。对于图(a)中的C语言程序，通过简单的推导即可得知第5行的断言将无法满足。对该C语言程序应用图(b)中的谓词抽象后，我们很容易得出图(c)中的布尔程序，并由此得出图2-2中的状态机，其中由于本身所指代的谓词，这五个变量可能的取值只有10000、01000、00100、00110、00111五种。

## 2.2 布尔程序的构建

本文中用于代码修复的布尔程序是通过SATABS[2]对C程序转换得来的。SATBS是一个模型检测的工具，实现了谓词抽象细化循环[3]，它能够将C/C++的程序转换成布尔程序。给定的程序将会被划分为由赋值语句组成的初始块和控制语句（例如, 和语句等）。

给定了输入程序之后，SATABS会自动计算出它的抽象表示。为了达到程序的状态空间缩减的目的，抽象是一个很有效的方法。谓词抽象[4]是其中一种广泛应用的方法，它通过只保持跟踪一定的谓词来抽象数据。在抽象模型中，消去了原程序的变量，以布尔变量来表示根据原程序抽象出来的谓词。抽象程序是通过存在抽象[5]创建的，存在抽象对于可达性属性来说是保守的抽象，也即对于在抽象模型中存在的属性，也一定能在原始程序中找到同样的属性。

## 2.2.1 赋值语句转换

SATABS根据具体程序的状态迁移关系计算基本块的抽象迁移关系，得出抽象谓词集合，每一个布尔变量对应集合中的一个谓词。在得出谓词集合后进行转换的过程中，若原始程序在位置处有赋值语句，则布尔程序在出将包含对布尔变量的赋值语句[6]。布尔变量的真假值根据计算出的抽象状态迁移关系的谓词的真假值进行复制。如果在处不能确定谓词的真假值，则在中的布尔变量只表示为：[6]。

例如，对于图2-1(a)中的C语句程序，利用SATABS计算得出的布尔变量的谓词集合为，则C语言程序的赋值语句和对应的布尔程序语句为和。

## 2.2.2 控制流语句转换

除了由赋值语句组成的初始块，程序中还包括了、和等控制语句。这些语句包含作为参数的条件语句，但它们不会改变变量的取值而只会决定控制流的方向[7]。表2-1中给出了C语言程序中比较常见的控制语句转换的例子。从表中的3个例子可以总结出C语言程序的控制结构转换为布尔程序的规则。

|  |  |  |  |
| --- | --- | --- | --- |
| 语句类型 | C语言程序 | 谓词抽象 | 布尔程序 |
|  | if (x > 1) {  f();  } | b1 : x > 1 | if !b1 then goto l1; fi;  f();  l1: |
|  | while (x > 1) {  x = x – 1;  } | b0 : x == 0  b1 : x ≥ 1  b2 : x == 1  b3 : x ≥ 2 | l1:  if !b2 then goto l2; fi;  b0,b1,b2,b3 := b2,b3,\*,\*;  goto l1;  l2; |
|  | a = 2  for (i = 0; i < a; i++) {  x = x – 1;  } | b0 : x == 0  b1 : i ≥ a  b2 : 1 + i ≥ a  b3 : x == 1  b4 : a ≤ 0  b5 : a ≤ 1 | b1,b2,b4,b5 := \*,\*,0,0;  b1,b2 := b4,b5;  l1:  if b1 then goto l2; fi;  b0,b3 := b3, \*;  b1, b2 := b2, \*;  goto l1;  l2: |

由此，我们不难得出结论，C语言程序的条件控制和循环控制语句在转换时均被转换为布尔程序

对于条件选择，我们假设原始C语言程序在位置对一个控制条件语句进行判断：如果该控制条件成立，那么程序将会跳转到位置继续执行条件成立时的语句，否则会跳转到位置执行该条件不成立时的语句。转换时，我们需要对控制条件语句进行抽象，首先就需要遍历该语句的语法结构，检查该控制条件语句中的子表达式是否是谓词集合中的谓词：如果是的话，用对应的布尔变量取代控制条件语句中的表达式。SATABS所采用的策略是对转换后的控制条件语句进行取非，真正在布尔程序中用于判断的条件语句为：当成立时，程序将会跳转到位置，即例子(1)中的，并结束表达式；否则，程序将进入位置，即例子中语句的后续语句，继续执行。

条件循环语句在布尔程序中则被转换成了条件选择。假设原式C语言程序在位置有一个对条件语句进行判断的条件循环语句，转换后的表达式遵循如下规则：若不成立，程序将跳转到位置；否则，程序跳转至位置，执行完循环体对应的所有语句后重新跳转到位置。

条件循环语句在布尔程序中同样被转换成了条件选择。本质上，循环与循环是相同的，差别仅在于循环能够同时为循环内部声明局部变量，因此也需要对其所声明的局部变量进行抽象。转换时，循环的初始化语句被转换为对应的赋值语句后，放在了转换后的语句的上方。同时，循环的增量表达式也被转换为了赋值语句，并被置于标志循环体结束的语句上方。

## 2.2.3 函数调用的转换

布尔程序对于原始C语言程序中的函数调用将进行保留，因此布尔程序可以很好地展现具体的C语言程序里的函数调用关系。函数的调用分为有参数与无参数函数调用，以及有返回值与无返回值函数调用，其中有参数和无参数的函数调用在转换为布尔程序后均被转化为了无参数函数调用。

对于无返回值函数的调用，一般情况下被调用函数都是用于对全局变量进行操作，因此函数调用语句本身可以直接转换，即无论函数调用语句是还是，转换成布尔程序后均为。

对于有返回值函数的调用则稍微复杂一些。在转换时，我们需要额外声明一个布尔变量：该布尔变量不表示任何从C语言程序中抽象出来的具体谓词，而是用于表示被调用函数的返回值。如图2-3中所示的布尔程序，变量代表的便是被调用函数的返回值。进行函数调用的时候，在调用语句之前，程序会对被调用函数里的局部变量赋初始值，在被调用函数结尾会将返回值赋予该布尔变量，并在调用语句结束后将该布尔变量的值赋予赋值语句所指定的布尔变量。

第三章 布尔程序修复

## 3.1 错误路径修复

为了找到错误，首先要建立合适的错误模型。在本文中，我们假设程序只包含一个错误，也就是说在C 语言程序中只包含一个错误语句。

## 3.1.1 错误模型建立

本文的错误模型只针对C语言程序的一个语句，更确切的说是针对C语言程序里的一行语句。在转换为布尔程序的过程中，原C语言程序的一个变量可能会对应多个布尔变量，C语言程序的一行也可能对应布尔程序的多行，而本文在对转换后的程序进行修复也只是针对布尔程序的一行错误进行修复。因此本文的错误模型在转换为布尔程序后对应的错误模型也必须只体现在一行语句上。形式化来讲，我们的错误模型有定义如下：

**定义3.1** 当C语言程序的出错语句为控制语句的控制条件时，则对应的布尔程序中出错的语句为控制语句；当C语言程序出错的语句为赋值语句时，则对应的布尔程序中出错的语句为赋值语句。即C语言程序的单行错误语句映射到布尔程序的单行错误语句。

我们将图2-1(a)作为图3-1。在这个错误的C语言程序中，变量在程序启动时被赋予了初值，并且程序在完成对函数的调用后，使用语句对程序的当前状态进行了断言。实际上，这两个操作组成了单元测试用例的两个主要元素，即“将程序设置到某个特定的状态”以及“断言程序的最终状态”。

因此我们给出错误程序的定义：

**定义3.2** 对于布尔程序，若给定的初始状态使得在某个语句上断言失败，那么我们就说没能通过由该初始状态和语句所断言的最终状态组成的单元测试，即布尔程序是错误的。

在2.1节中，我们将布尔程序的运行图视为了一个有穷自动机。在有穷自动机中，“路径”的概念自然是不陌生的。那么，对于这样的有穷自动机，我们不妨得出其路径的定义如下：

**定义3.3** 布尔程序的一条路径是一个状态的有序序列，其中对于任意使的元素对与，有。在非形式化的图形表示中，一条路径由一系列首尾相连的边组成。

结合定义3.2和定义3.3，我们给出错误路径的定义：

**定义3.4** 若存在状态，其中是一条断言语句，而当前估值不满足该断言，那么我们就将路径称为错误路径。

## 3.1.2 收集错误路径

对于已知的错误语句，我们需要计算出正确的修复语句来进行替换。我们假设是任意与变量相关的函数，并将作为替换错误语句的表达式，即修复语句。替换后，我们便需要去考虑在替换出所延伸出的每种可能路径。

正确的程序在运行的过程中总能从初始状态走到程序的正确终止状态且不会经过会使断言失败的错误状态。根据程序的运行图，我们可以构造出替换后的程序状态转换图，并能构造出所有从初始状态可能迁移到错误终止状态的错误路径。

假设布尔程序中只出现了一个。为了计算的恰当实现，我们需要构造该程序的状态转移图。在出现的地方，可以根据它的一个实现从当前状态转移到下一个状态，而它不同的实现则会使当前状态迁移到不同的状态。

假设有估值集合，满足：

**定义3.5** 对于任意的，有当且仅当。为所有可能的实现的集合。

如图3-2，其中图(a)即为原图2-1(b)中的程序运行图。我们将原布尔程序中的错误语句替换为后，考虑由不同实现所可能带来的所有状态转移，便能得出图3-2(c)中的状态转移图。我们设定了命题，它们分别代表着中的每个的元素，也就能够组合成不同的实现。以状态为例，程序在此行对函数返回的结果进行判断：若此时函数返回1，即命题成立，程序进入状态并再次调用函数；反之，若此时函数返回0，即命题不成立，程序进入状态并结束函数。假设我们进入了状态并随之来到了状态，程序再次对函数返回的结果进行判断：若此时函数返回0，即命题不成立，程序进入状态并结束函数。依此类推，我们就可以计算出函数从状态分别进入到第三行五个不同状态所需满足的条件。其中，状态、、、均无法通过断言，属于错误状态，其所属的路径便是错误路径。

## 3.1.3 修复结果

前文的过程可以找到状态转移图中所有连接初始状态和错误终止状态的错误路径。这些错误路径的的实现导致程序进入了错误的终止状态，那么不难得出，的正确实现则是这些错误实现所组成的集合的补集。

**定义3.6** 假设对于一个固定的实现有固定的程序执行路径，则正确的实现，其中为程序所有错误状态所组成的集合。

由此，在收集错误路径的过程中，我们也集了所有错误的实现。在上述例子中，我们便收集到了状态、、、所对应的四个错误的实现：、、e、，即当的实现满足时，程序将从初始状态进入错误的终止状态。根据上述定理，我们可以得出正确的实现，即唯一一条走向正确终止状态的路径。将转换为为此表示并化简后，可得结果。

## 3.2 布尔修复的可满足性问题

在得出修复公式后，我们还需要考虑该公式是否是可满足的。

布尔程序是对C程序抽象而来，得到布尔程序的修复之后，需要将其重新转换为C语言的修复语句。布尔程序的修复公式，简称布尔修复，是由一系列布尔变量和操作符构成的布尔表达式，其中的布尔变量由C语言程序转换为布尔程序中对原有的变量进行谓词抽象所得，因此布尔修复的每个布尔变量都有着对应的谓词。我们应用上一节所述方法得出的布尔修复将会是一个一阶逻辑公式。由于公式中的每个变量都着其背景理论，公式本身是否可满足是暂不确定的。例如图3-2中，谓词的背景理论为，谓词的背景理论为。考虑其背景理论后，实际上公式是不可满足的。由此，我们必须在定义3.6的基础上细化对布尔修复的定义：

**定义3.7** 设程序的所有错误路径集合为，且是可满足的，则存在布尔修复使得程序正确，该布尔修复为，其中为程序所有错误状态所组成的集合。

由此，在布尔程序中找到修复公式并不代表能为原本的C语言程序找到修复表达式。而判断一阶逻辑公式在给定背景理论的情况下是否可满足的问题可以规约到可满足性模理论问题（SMT）[8]。

### 3.2.1 可满足性模理论规约

可满足性模理论是一个扩展的SAT问题，通常应用于求解非量化的一阶逻辑公式的可满足性问题。形式化来说，对于一阶逻辑公式，其中一些函数和谓词符号有着额外的解释，即带有背景理论，而SMT问题则是决定这样一个公式是不是可以满足的。

假设我们有可数的变量集合、函数符号集合和谓词集合。A first-order signature is a partial map from to the natural numbers corresponding to the *arity* of the symbol. A -term has the form

（3.1）

Where and . For example, if and , then is a -term. A -formula has the form

（3.2）

Where and , and each is a -term. For example, if for a predicate symbol , then is a -formula.

A -structure *M* consists of a nonempty domain , for each such that , is a subset of , and for each ,\p . The interpretation of a term in is given by and . For a -formula and a -structure , satisfaction can be defined as

A first-order -formula is satisfiable if there is a -structure such that , and it is valid if in all -structures , .[9]

我们不妨以析取范式来表示通过本文方法得出的布尔修复，即对于布尔修复，有：

（3.3）

其中是由谓词变元组成的公式，它的形式如下：

（3.4）

其中项则如公式（3.1）所示，可以是个体变元或常元，也可以是函数常元或变元。

不难看出，由公式（3.3）得到的程序修复公式形如吻合非量化一节逻辑公式的定义公式（3.2），可见是一个非量化的一节逻辑公式。

此外，给定修复公式有背景理论。布尔程序的背景理论均由原C语言程序变量进行谓词抽象得出，因此背景理论对应修复中的谓词变元，背景理论的赋值模型与谓词变元的赋值模型一致。

综上，根据先前提到的对可满足性模理论问题的定义，可以证明修复公式的可满足性问题都可以规约到可满足性模理论问题。

将布尔程序修复结果转换为C语言表达式，实际上等同于转换成了带背景理论的非量化一阶逻辑公式。为布尔程序找到的修复公式一定是布尔程序的正确修复，而转换后的是否能使C语言程序正确则仍需要进行可满足性判断。由上述证明，我们可将判断修复公式的可满足性问题归约为可满足性模理论问题，利用对可满足性问题求解的方法便很容易求解出修复公式的可满足性。

## 3.3 修复公式化简

给定析取范式，假设存在模型使得，即公式是可满足的。实际上，由于是一个析取范式，它的可满足性只能推出它包含可满足的合取子句，但并不能说明它的所有合取子句都是可满足的。

形式化地说，对于给定的修复公式，它的可满足并不代表每个都是可满足的。例如，对于修复公式，其中，，。公式本身是可满足的，因为子句可满足，但子句是不可满足的。

若将上例给出的修复公式转换为C语言表达式，我们可以得到C语言修复表达式，其中部分是没有意义的，该表达式可直接化简为。由此可见，我们在将布尔修复转换为C语言表达式前，还需要通过验证其各子句的可满足性来对其进行化简。

接下来我们将通过三种不同的方式对公式进行化简。

## 3.3.1 移除不可满足子句

给定可满足的修复公式，公式子句外的化简是通过判断公式中每个修复子句的可满足性进行的。如果修复子句是可满足的，则保留该修复子句；否则，则将其从修复公式中移除。由3.2节可知，判断子句的可满足性同样可以规约到可满足性模理论问题。我们将移除不满足修复子句后产生的新的修复公式称为。

可以证明，通过子句外化简得到的和原修复公式是等价的，即。

假设由移除不满足子句得出，即

由基本的推理公式可知

故

已知子句是不可满足的，即恒为假，恒为真。同样，根据基本的推理公式可有

即

由于恒为真，固可有

由此可得

至此，我们便完整证明了，即化简后的修复公式与原修复公式是等价的。

## 3.3.2 析取范式化简

经过上节介绍的方法，我们成功消除了修复公式中不可满足的子句。在本节，我们将讨论修复子句之间可能的化简形式，即对析取范式本身进行化简。

**定理3.1** 给定可满足公式，当或恒为真时，可以化简为或。

对公式使用分配律可以得到，由此不难证得上述定理。

**定理3.2** 给定可满足公式，当的每个谓词都是的谓词时，可以化简为的形式。

**证明**

由于的每个谓词都是的谓词，可有，那么

应用分配律可化为

其中，右侧的可应用分配律化为

故

其中，右侧的可应用交换律化为

同时，同为右侧的也可得到

综上可得

故可有

对上一节所得的修复公式，我们不断地尝试应用上述两个化简规则，最终将获得一个最简的可满足修复公式。

## 3.3.3 结合背景理论化简

假设前文得到的最简可满足公式为，其中，，则在转换为C语言表达式之后，，仍可化简为。因此，根据背景理论对谓词进行化简能够得到一个更为直观的C语言表达式，而本节将讲述这种化简方式。

**定理3.3** 给定可满足最简修复公式和对应的背景理论，如果存在一个二元谓词，则修复公式可化简为。

**定理3.4**给定可满足最简修复公式和对应的背景理论，如果存在一个二元谓词，则修复公式可化简为。

定理3.3和定理3.4所进行的化简与实际背景理论有关，而其中出现的新谓词有可能就是或，也有可能是其它的谓词。

通过应用上述两种化简方式，最终我们能够获得一个相对直观的最简C语言修复表达式。

## 3.4 本章小结

本章3.1节介绍了如何通过收集布尔程序的错误路径来获得布尔程序的修复表达式，3.2节描述了如何将该修复结果的可满足性问题规约到可满足性模理论问题。最后，我们在3.3节给出了对可满足修复公式的各种化简方式，从而完善了将C语言程序转换为布尔程序进行修复并将修复结果映射回C语言表达式的过程，最终得到了正确有效且相对直观的C语言程序修复表达式。

第四章 布尔程序修复工具的设计与实现

布尔程序修复系统主要分为四个模块：C语言程序转换模块、布尔程序预处理模块、布尔程序修复模块、布尔修复逆向转换模块。整体架构如图4-1所示。

## 4.1 布尔程序修复工具模块介绍

本节首先介绍系统各个模块的作用、输入输出以及该模块的主要实现方式。

* C语言程序转换模块：将C语言程序转换为布尔程序，同时对程序的变量和表达式进行谓词抽象。

本模块主要调用了基于可满足性的程序抽象工具SATABS[2]来对原C语言程序进行转换。SATABS能处理C语言程序的赋值语句、控制流语句和函数调用语句等的转换，可以完成C语言程序到布尔程序的初步转换过程。

* + 输入：包含测试用例和预期输出的错误程序文件
  + 输出：错误程序对应的布尔程序文件
* 布尔程序预处理模块：布尔程序预处理模块在一定程度上对C语言程序转换模块的完善。该模块对SATABS转换得出的布尔程序进行进一步的处理，去除文件中的无用信息，同时将布尔变量和其所代表的谓词以及C语言程序与布尔程序对应的位置关系提取出来。我们还在这里输入了原C语言程序出错语句的位置，使后续修复模块能够针对错误的位置进行修复。本模块主要根据布尔程序的语法语义以及SATABS中C语言程序到布尔程序的转换规则对上一模块输出的文件进行分析和处理。
  + 输入：SATABS输出的布尔程序文件和C语言文件错误位置行号
  + 输出：处理后的布尔程序文件和关系映射
* 布尔程序修复模块：对已知错误位置的布尔程序构建错误路径并计算得出正确的布尔修复。本模块可以细分为错误路径构建和修复求解两部分。错误路径构建主要是在已知错误位置的情况下，将错误语句用未知修复函数替代，并构造出所有的错误路径（见3.1.2节）。修复求解则是在构建完错误路径后根据错误路径计算出修复函数的实现，即布尔修复（见3.1.3节）。
  + 输入：处理后的布尔程序文件和关系映射
  + 输出：布尔修复语句
* 布尔修复逆向转换模块：本模块首先将布尔修复的布尔变量用预处理模块给出的关系映射里对应的谓词进行替换，在根据上文的SMT问题规约（见3.2节），判断转换后的修复表达式的可满足性，同时根据已知的化简规则（见3.3节）将转换后的修复表达式化至最简，从而得到一个正确有效的修复。
  + 输入：布尔修复语句和关系映射
  + 输出：修复后的C语言程序

## 4.2 布尔程序修复模块的设计与实现

我们首先需要利用布尔程序修复模块找到布尔程序的正确修复。在对布尔程序进行修复的时候，我们通过引入多个测试案例来确保修复结果的准确性。该模块又可以细分为错误路径构建和修复求解两部分。

对于一个给定的错误程序，我们需要构建出它所有可能到达错误终止状态的路径。本文以测试用例作为判断程序正确性的标准，通过语句来给出程序需要满足的规范，也即测试用例对应的正确输出结果。为错误程序构建错误路径时，我们会将错误语句所在的位置作为参数传入，将对应位置的语句替换为未知的修复函数，再将所有可能到达的错误状态的路径记录下来。

在求出了给定测试用例的所有错误路径之后，我们将对错误路径的并集取反来得到一条正确的执行路径，再进而计算该路径的具体实现。

## 4.2.1 多测试用例修复方法

为了确保修复结果的正确性，我们决定为一个错误程序引入多个不同的测试用例进行多次错误路径构建，从而使得到的错误路径能够完全覆盖。

3.1.2节中描述的代码修复方法能够根据给出的输入和正确的输出找出所有可能到达错误终止状态的错误路径。当只给出一个测试用例时，程序的初始状态是确定的，因此在构建错误路径时所能得到的错误路径也是确定的。不同的测试用例往往会有不同的初始状态，而在构建错误路径时，不同的初始状态到达错误状态的错误路径也是不相同的。因此，在修复的过程中只引入一个测试用例来进行修复是不完备的。

我们针对同一个错误程序引入多个不同的测试用例，得到多个不同版本的C语言程序文件，再分别对它们进行错误路径构建和收集，最终再对这些结果进行分析，得出正确的修复结果。尽管不同版本的C语言程序转换为布尔程序后，布尔变量所代表的的谓词可能会有所不同，但在找到错误路径后我们可以先对不同错误路径中的布尔变量进行合并，即对代表相同谓词的布尔变量统一命名。

**定理4.1** 设引入一个测试用例对程序构建得到的错误路径集合为，在此基础上增加测试用例对程序构建得到的错误路径集合为，则有。

由定义3.6可知，的正确实现由排除所有错误路径后计算得出的。在引入了多个测试用例之后，由定理4.1我们可以得知，在程序修复过程中得到的错误路径的覆盖范围将增大，也即找到了更多错误的实现。如此一来，我们便使得得出的修复结果更为精确，其所能覆盖的程序的范围也更广了。

## 4.2.2 布尔程序修复模块的伪代码

（直接复制原文的4.3.3节）

## 4.3 布尔修复逆向转换模块的设计和实现

通过布尔程序修复模块，我们得到了针对布尔程序的布尔修复。布尔修复是由布尔变量组成的布尔表达式。为了得到C语言程序的修复结果，我们还需要其转换为C语言表达式，并判断转换后的结果是否是正确且有效的。

（直接复制原文的4.4.2节）

## 4.4 算法复杂度分析

（直接复制原文的4.5节）

第五章 实验结果

第六章 结语

在本文中，我们介绍了如何将错误的C语言程序转换为布尔程序，通过收集布尔程序的错误路径得出正确的布尔修复，并介绍了该如何验证该布尔修复公式的可满足性。最终通过对布尔修复公式进行化简，我们得到了正确有效且相对直观的C语言修复表达式。我们还对实现的工具进行了大量的测试，证明了该方法对TCAS测试集中的赋值语句错误和控制语句错误都能找到正确的修复结果。

参考文献

|  |  |
| --- | --- |
| [1] | Ball T., Rajamani S. Boolean programs: A model and process for software analysis[J].  Microsoft Research, http://research.microsoft.com/slam, 2000. |
|  |  |
| [2] | Clarke E., Krowning D., Sharygina N., Yorav K. SATABS: SAT-based predicate abstraction for ANSI-C[A]. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)[C]. 2005: 570-574. |
|  |  |
| [3] | Basler G., Krowning D., Weissenbacher G. A complete bounded model checking algorithm for pushdown systems[A]. Proceeding of HVC 2007[C]. 2007: 202-217 |
|  |  |
| [4] | Colon M., Uribe T. Generating finite-state abstractions of reactive systems using decision procedures[A]. Computer Aided Verification[C]. 1998: 293-304. |
|  |  |
| [5] | Clarke E., Grumberg O., Long D. Model checking and abstraction[J].  ACM Transactions on Programming Languages and Systems (TOPLAS), 1994, vol. 16(5): 1512-1542. |
|  |  |
| [6] | 王波, 沈云付. 针对Java语句中间形式的谓词抽象算法[J].  计算机工程与设计, 2009, (013): 3185-3188. |
|  |  |
| [7] | Clarke E., Kroening D., Sharygina N., Yorav K. Predicate abstraction of ansi-c programs using sat[J].  Formal Methods in System Design, 2004, vol. 25(2): 105-127. |
|  |  |
| [8] | Barrett C., Sebastiani R., Seshia S., Tinelli C. Satisfiability modulo theories[J].  Handbook of Satisfiability, 2009, vol. 4. |
|  |  |
| [9] | De Moura L., Dutertre B., Shankar N. A tutorial on satisfiability modulo theories[A].  Computer Aided Verification[C]. 2007: 20-36 |