
# 南京大学

# 本 科 毕 业 设 计

院　　系　计算机科学与技术系

专　　业　计算机科学与技术

题　　目　基于机器学习和选择取样的循环不变式自动生成

年　　级　2016 级　学 号 161220092

学生姓名　孟 华

指导教师　卜磊　职 称　教授

提交日期

# 南京大学本科生毕业论文（设计、作品）中文摘要

题目：基于机器学习和选择取样的循环不变式自动生成

院系：计算机科学与技术系

专业：计算机科学与技术

本科生姓名：孟华

指导教师（姓名、职称）：卜磊 教授

摘要：

当前软件产业快速繁荣发展，人们越来越关注软件安全。传统的软件安全检测是通过大量软件测试来进行错误定位，但是成本较高，且不能完全保证软件可靠性。而基于严密的数学和逻辑基础的形式化证明可以大大提高软件可靠性。发现充分的循环不变式信息，是其中的关键问题之一。但循环不变式的人工查找成本较高且精确度难以得到保证，因此循环不变式的自动生成成为一个重要课题。

传统的循环不变式自动生成大多是应用约束求解，但是不能解决复杂程序。因此研究人员提出“先猜后证”方法，主要思路是在初始样本的基础上生成候选循环不变式，再验证是否满足霍尔三元组的限制。但是此方法性能和效率依赖于初始样本且需要大量迭代，因此研究人员提出了基于机器学习和选择取样的循环不变式自动生成方法，通过选择取样加速循环不变式收敛从而减少迭代次数和对初始样本的依赖。

我们的研究目标，就是基于机器学习和选择取样自动生成循环不变式这一方法，实现一个高效精确的循环不变式自动生成系统 MIG，并且在这个方法基础上，增加“人机交互”的方式，进一步减少整个系统的迭代次数。

MIG 是一个面向 c 和 c++程序的循环不变式自动生成系统，支持对于 bool、int 和 double 型变量的简单和复杂程序生成线性或非线性循环不变式。为了测试 MIG 的实际效果，本文实验设置了 20 个实际程序作为测试用例。实验数据表明，增加选择取样和人机交互模块可以显著减少算法总体迭代次数，并且可以解决单纯机器学习算法无法收敛的问题；同时通过实验发现 MIG 和目前其他循环不变式生成系统相比在性能上具有较好的表现，达到了预期效果。

关键词：循环不变式；机器学习；选择取样；人机交互

# 南京大学本科生毕业论文（设计、作品）英文摘要

THESIS：Loop Invariant Generation by Machine Learning and Selective Sampling

DEPARTMENT: Computer Science and Technology Department

SPECIALIZATION: Computer Science and Technology

UNDERGRADUATE: MENG Hua

MENTOR: BU Lei

ABSTRACT：

With the prosperity of the software industry, people are paying more and more attention to software security. The traditional software safety detection is to locate errors through a large number of software tests, but the cost is high, and the software reliability cannot be fully guaranteed. Formal verification based on mathematical and logical foundations can greatly improve software reliability. And finding sufficient information of the loop invariant is one of the key problems. The manual search for loop invariant is expensive and difficult to guarantee accuracy, so the automatic generation of loop invariant becomes an important issue.

The traditional loop invariant automatic generation is mostly solved by constraints solving, but it cannot solve complex programs. Therefore, the researchers proposed a "guess and check" method. Firstly generate candidate loop invariant based on the initial samples, and then verify whether can prove the Hoare triple. However, the performance and efficiency of this method depend on the initial samples and require a large number of iterations, so the researchers proposed a method to generate loop invariant based on machine learning and selective sampling. By selecting sampling, the convergence of loop invariants is accelerated.

Our research goal is to implement an efficient and accurate loop invariant generation system, MIG, based on above method. And add human-machine interaction Way to further reduce the number of iterations of the entire system.

MIG is a loop invariant automatic generation system for c ++ programs, which supports the generation of linear or nonlinear loop invariants for simple and complex programs of bool, int and double type variables. In order to test the effect of MIG, we

set up 20 actual programs as test cases. Experimental data shows that adding selective sampling and human-machine interaction can reduce the overall number of algorithm iterations, and can solve the problem that pure machine learning algorithms cannot converge; at the same time, through experiments, it is found that MIG has better performance than current loop invariant generation systems. We achieved the desired effect.

KEY WORDS: Loop Invariant; Machine Learning; Selective Sampling; Human-Machine Interaction

# 目 录

# 第一章 绪论

当今社会已经进入信息化时代，繁荣发展的计算机软件为人类的生活带来了无数便利，计算机技术已经渗透进了我们生活的各个方面，因此保证软件正确性也就显得尤为重要。相比于成本较高且并不能完全保证正确性的软件测试，基于数学和逻辑基础的形式化证明则更能保障软件的正确。而发现充分的循环不变式信息是形式化证明中的重要环节。传统的约束求解获得循环不变式信息会受到程序复杂度的限制，因此本文在基于机器学习和选择取样的循环不变式自动生成方法的基础上，添加人机交互方式进一步加快算法收敛，设计实现了面向 c 和 c++ 文件的循环不变式自动生成工具。

## 1.1 本课题的目的、意义

现如今，软件产业快速繁荣发展，大量软件不断问世，与此同时，人们也越来越关注软件安全。2004 年，EDS 为英国儿童支持机构提供了一款高复杂度系统，但是由于软件适配等问题，导致这个系统当天多付了 190 万人的福利金，造成了超过 1 亿美元的损失。2012 年 8 月 1 日，在 Knight 交易软件中的一个缺陷，使得在 Knight 在当天纽约证券交易所营业的第一个小时内疯狂买入了总成本约为 70 亿美元的股票，造成了重大损失。由此可见，软件安全在实际的生产生活中十分重要。

传统的软件安全检测方法是通过大量的软件测试来进行错误定位与分析，但是这样对测试人员的要求较高，需要耗费大量人力和成本；同时由于测试的局限性，并不能完全保证软件的可靠。因此，研究人员将焦点转移到形式化验证上，形式化验证是基于严密的数学和逻辑基础，可以大大提高软件可靠性。但是形式化验证中的模型检验在遇到循环时如果展开验证，往往会遇到状态爆炸的问题，因此，发现充分的循环不变式信息，是形式化证明方法中的关键问题之一。如果对于循环不变式进行人工查找，同样成本较高并且精确度难以得到保证，因此循环不变式的自动生成成为一个重要的课题。传统的循环不变式自动生成大多是应用约束求解，但是一旦程序复杂度升高则难以求解，并且不能很好的扩展。

最近，研究人员提出了“先猜后证”的循环不变式自动生成方法，主要思路是在初始样本的基础上生成候选循环不变式，再验证是否满足所要求的霍尔三元组的限制，如果满足要求则得到最终的循环不变式，反之，则可以将不满足的边界条件作为初始样本的补充添加进去，重新循环直到找到正确结果。但是这个过程的精确度及迭代次数严重依赖于初始样本的质量，对于初始样本的数量有比较高的要求，并且往往需要多次迭代。为了解决这一问题，研究人员提出在利用机器学习对于初始样本进行分类的时候，结合选择取样这一主动学习方式，自动优化生成的候选循环不变式，在减少迭代次数的同时提升精度。

因此，我们的研究目标，就是基于机器学习和选择取样自动生成循环不变式这一方法，实现一个高效精确的循环不变式自动生成系统，并且在这个方法基础上，增加“人机交互”的方式，进一步减少整个系统的迭代次数。

## 1.2 相关工作

循环不变式的生成是形式化验证中的经典问题，有大量的关于循环不变式的自动生成的研究工作，我们可以将现有的工作大致分为静态生成和动态生成两种。

### 1.2.1 循环不变式的静态生成

循环不变式的静态生成主要分为抽象解释和约束求解两大框架。抽象解释是指让程序在抽象域上进行符号执行，在这一过程中它会不断趋近于真正循环的语义。它过度逼近循环迭代的语义。更新后的该技术支持现代编程语言，包括堆管理和面向对象过程。但是抽象解释工具仅可以验证不存在简单和常见的编程错误，例如被零除或无效解引用，对于复杂程序的支持性不好。

约束求解技术则非常依赖于数学方面的求解性能，比如在多项式或凸多面体上进行的复杂决策。约束求解技术通过将程序中的特征以数学方式提取出来，并且通过一系列计算化简等，生成循环代码片段的语义。和抽象解释技术类似，约束求解技术对于复杂性程序的支持性不好，并且往往需要大量的计算资源。

静态方法是有效且合理的，而且通过静态方法推导得到的循环不变式往往是正确且完整的。但是这些技术由于其自身限制，往往对于程序的复杂度有一定要

求，并不能满足日常应用的需要。

### 1.2.2 循环不变式的动态生成

近些年来，循环不变式的动态生成越来越受到研究人员的关注，动态生成的可行性也得到了证明。动态方法与 1.1 节中提到的“先猜后证”方法十分类似，首先需要发掘出程序中的大量候选属性集合，然后在程序运行的过程中，所有没有被违反的性质都被保存到最终生成的循环不变式当中。也就是说，我们得到的循环不变式只能说明在运行过的程序中正确，这是“可能”的不变式，或者可以说是有一定依据的猜测，因此如何提高这个猜测的正确率成为了研究人员关注的热点。

在提高正确率和运行效率的诸多尝试中出现了很多效果不错的方法：Gupta 和 Heidepriem 通过使用不同代码覆盖率的测试套件，提高了生成的循环不变式的精度。 Fraser 和 Zeller 通过发现代码库中存在大量重复片段，通过应用这些重复改进了测试用例，使得生成的测试用例更易理解且提高了运行的效率。但是这些动态生成技术往往需要提供足够多的测试用例，才能保证比较高的精度，

但是对于处理复杂程序的循环来说，生成大量测试用例是非常耗时且不现实的，因此研究人员提出了启发式动态生成方法。如文献[]中利用随机搜索加验证工具来生成循环不变式，但是单纯使用随机搜索会使得整体算法收敛时间过长，而且生成的循环不变式精度也并不能得到保证。所以在此基础之上，研究人员提出了基于机器学习的循环不变式自动生成，即首先生成一系列初始样本及相应标签，然后对于初始样本进行分类学习“猜”得循环不变式，最后验证生成的循环不变式是否满足给定霍尔三元组。现有的基于机器学习的循环不变式自动生成技术有许多不同的初始样本和循环不变式生成方法，但是他们共有的问题为系统的效率和性能依赖于第一次生成的样本的质量，并且通常需要大量迭代次数和大量样本数量，因此我们的工具采用的基于机器学习和选择取样的循环不变式自动生成可以大大缓解这一问题，通过在学习候选循环不变式阶段的主动学习迭代加速整体算法收敛速度。

## 1.3 研究内容

基于上文所述，我们需要设计实现一个基于机器学习和选择取样的循环不变式自动生成系统，并且支持人机交互功能，这一系统被命名为 MIG (Machine Learning based Invariant Generation)，下面将介绍具体的模块划分与相关工作：

（1）基于约束求解和随机取样的初始样本生成模块：利用约束求解和随机取样，生成给定霍尔三元组的初始样本及对应标签。

（2）生成候选循环不变式模块：这一模块主要分为两个部分，第一部分为利用 SVM 算法对于模块一中生成的带标签的初始样本进行分类学习，得到分类超平面即为候选循环不变式；第二部分为选择取样，根据候选循环不变式生成边界样本，添加标签后加入到原始样本集中重复模块二，直到收敛。

（3）验证候选循环不变式是否满足霍尔三元组模块：这一模块主要分为两个部分：第一部分为采用约束求解和符号执行结合的方式，验证是否存在不满足霍尔三元组的样本点，如果不存在，则证明生成的循环不变式正确；如果存在，则可以将生成的样本点作为边界点加入到初始样本集中，重复整个算法进行迭代；第二部分为人机交互模块，如果候选循环不变式不能满足霍尔三元组，且用户配置参数中开启了人机交互选项，则会进入人机交互部分，按照提示输入对应变量的值作为样本点加入初始样本集重复算法迭代。

通过 MIG 与 Interproc 等传统循环不变式生成工具，以及 MIG 自身是否开启选择取样和人机交互模块在 20 个实际程序的测试分析，MIG 系统在效率和性能上都有较好的表现，符合预期效果，具体实验设置与实验数据将在第五章进行详细说明。

## 1.4 论文结构

论文第一章介绍了此研究的目的和意义，与此相关的现有工作以及概述了此研究的主要内容。第二章介绍包括形式化验证、循环不变式、霍尔三元组、支持向量机监督学习算法等此研究用到的相关背景知识，并且介绍了基于机器学习的循环不变式自动生成的相关概念和算法。第三章介绍基于机器学习和选择取样工具的模块划分、具体设计和相关算法。第四章详细介绍 MIG 系统的具体实现细节。第五章介绍了 MIG 系统的运行说明和实验设置，对比了是否开启选择取样和人机交互模块的性能和效率，并且通过与其他循环不变式生成工具的实验比

较，分析 MIG 系统表现。第六章为论文的总结，概述了本课题的主要贡献及相应创新点，并且提出了改进的方向。

# 第二章 背景知识

## 2.1 形式化证明

现如今，软件产业快速繁荣发展，大量软件不断问世，与此同时，人们也越来越关注软件安全。传统的软件安全检测方法是通过大量的软件测试来进行错误定位与分析，即使对于很简单的程序，所有可能的测试用例也基本上是无穷的，需要耗费大量成本；同时由于测试的局限性，并不能完全保证软件的可靠性。因此，研究人员将焦点转移到基于严密的数学和逻辑基础的形式化证明上，大大提高软件可靠性。

形式化证明是针对软件或硬件系统，提供一种形式化方法或数学证明，来证明或证伪某个系统的某种属性预期算法的正确性。形式化证明可以证明许多系统的正确性，例如组合电路、加密协议、数字电路或软件系统等。这些系统的形式化证明是通过对系统抽象出数学模型，然后对该模型进行形式化证明来完成的。经常用于对系统进行抽象建模的数学方法有：有限状态机、Petri 网络、编程语言的形式语义，比如公理语义、操作语义和霍尔逻辑等。

形式化证明主要包含两种主流方法，演绎验证(Deductive Verification)和模型检查(model checking)。演绎验证主要使用已有的公理定理体系，证明从系统中抽象出的数学逻辑的正确性。但是这种方法需要用户熟悉系统如何正常工作，并且要把相关信息以定理序列的形式传给验证工具，并不方便用户实际操作。

模型检查主要通过从系统中抽象出有限状态集来表示系统运行中的路径和过程，通过对于有限状态机的分析进行正确性的验证。模型检查的优点是它是全自动系统，避免了演绎验证那样对于用户输入的高要求。但是它也有一个显著缺点就是通常无法应用到大型系统，因为可能存在的循环会使得整个状态机存在状态爆炸问题。

因此，如何发现准确的循环不变式信息，并用循环不变式代替循环在模型检查中的状态，可以显著提高模型检查工具的应用范围和使用场景。除此之外，循环不变式除了在验证中具有重要作用之外，对于循环不变式还可以增强对于算法

本身的了解，从而有助于理解整个程序。但是在实践中，如何自动生成合理准确的循环不变式仍然是一个挑战。

## 2.2 循环不变式

循环不变式是指在循环代码中恒为真的断言，即在循环的每轮迭代开始之前和之后都满足的性质。在形式化证明中，如果可以生成准确的循环不变式，则可以将循环代码替换为相应断言，从而简化整体的证明经过。在给出循环不变式的定义之前，为了方便后续说明，我们定义一个小型的编程语言。

### 2.2.1 小型编程语言定义

1. 变量：使用$V$, $V_1$, ... , $V_n$ 来表示任意变量，如具体变量 X, R, Q 等。
2. 表达式：使用$E$, $E_1$, ... , $E_n$ 来表示任意表达式，如表达式 X + 1, $\sqrt{2}$ 等。
3. 语句：使用$S$, $S_1$, ... , $S_n$ 来表示任意语句，如 X < Y, X = 1 等等。
4. 程序命令：使用$C$, $C_1$, ... , $C_n$ 来表示任意程序命令，具体包括：
   - 赋值语句：$V := E$，如 $X := X + 1$
   - 语句序列：$C_1; ... ; C_n$，语句按顺序执行，如 $R := X; X := Y; Y := R$
   - 条件语句：***IF*** $S$ ***THEN*** $C_1$ ***ELSE*** $C_2$，如果 S 为 true，则执行$C_1$，反之执行$C_2$，如 ***IF*** $X < Y$ ***THEN*** $Max := Y$ ***ELSE*** $Max := X$
   - 循环语句：***WHILE*** $S$ ***DO*** $C$，当 S 为 true 时一直执行 C，如 ***WHILE*** $(X = 0)$ ***DO*** $X := X - 2$

### 2.2.2 循环不变式定义

一个循环不变式包括：

1. 由 2.2.1 中程序语言组成的循环程序片段，用 C 表示
2. 一个表达式，用$\Phi$表示。
3. 循环不变式要求，表达式$\Phi$在程序片段执行 0 或多次之后恒成立。

根据定义，循环不变式是对于一段循环代码片段的断言，表示在这一循环代

码片段中恒成立的表达式。值得注意的是，循环不变式的结果并不唯一，但通常我们更关心可读性好且范围适当的循环不变式，如对于以下程序来说：

```
1: int x = 0;
2: while(x < 10)
3:     x++;
```

满足的循环不变式有 `x != 100, x > -1000, x >= 0, x>=0 && x <= 10`，可以看出虽然都符合循环不变式的定义，但是显然最后一个最为精确，也更加符合我们对于循环不变式的期望。

## 2.3 霍尔三元组

霍尔逻辑（也被称为弗洛伊德-霍尔逻辑或者霍尔规则）是一组逻辑规则系统，可以用来严密证明和推理程序的正确性。而霍尔逻辑的中心特征是霍尔三元组，它说明了代码片段在运行过程中如何更改自身状态。以 2.2.1 中程序语言为例定义霍尔三元组如下：

一个霍尔三组包括：

1. 由逻辑表达式组成的前置条件，用 P 表示。
2. 由 2.2.1 中程序语言组成的程序片段，用 C 表示。
3. 由逻辑表达式组成的后置条件，用 Q 表示。
4. 一个逻辑断言，如果满足 P，则执行 C 0 或多次之后，满足 Q。

霍尔三元组通常记为$\{P\}C\{Q\}$，其中，P 和 Q 为断言而 C 为指令集，一般将 P 命名为 precondition，而 Q 命名为 postcondition，当满足 precondition 时，执行指令集 C，而执行后结果一定满足 postcondition。为了更方便地说明霍尔三元组在循环不变式生成中的作用，我们将霍尔三元组的表示方式更新如下：

$\{Pre\}$ /* Assumption*/
$while(Cond)\{Body\}$ /* Loop Body*/
$\{Post\}$ /* Assertion*/

从霍尔三元组的定义和性质可以看出其和循环不变式具有相关性，循环不变式用 Inv 表示，假设$V=\{x_1,x_2,...,x_n\}$是和循环体 Body 相关的有限变量集，也是

生成循环不变式所关心的变量集合，令 $s=\{x_1 \mapsto v_1,...,x_n \mapsto v_n\}$ 表示对于变量集 V 的一组取值，用 *Body*(*s*) 表示对于给定 s 执行完 Body 之后的取值。

则二者之间满足以下三个条件：

$$Pre \subseteq Inv \quad (1)$$

$$\forall s.\ s \in Inv \wedge Cond \Rightarrow Body(s) \in Inv \quad (2)$$

$$Inv \wedge \neg Cond \subseteq Post \quad (3)$$

对于给定循环代码片段和霍尔三元组，霍尔三元组并不一定是正确的，如果存在 $s \in Pre$ 且执行完循环跳出时候的值 $s', s' \notin Post$，则该霍尔三元组错误。

## 2.4 基于机器学习的循环不变式自动生成

传统的基于约束求解的循环不变式自动生成会有程序复杂度的限制，因此研究人员提出了基于机器学习的循环不变式自动生成，运用“先猜后证”的方法生成复杂程序的循环不变式信息。为了更好的说明该算法的流程，我们将结合文献[]中的示例程序进行介绍，示例程序如图 2-1 所示：

```
1  assume(x<y);
2  while(x<y){
3    if(x<0) x:=x+7;
4    else x:=x+10;
5    if(y<0) y:=y-10;
6    else y:=y+3;
7  }
8  assert(y ≤ x ≤ y+16)
```

(a) Invariant: x ≤ y+16

```
1  assume(x ≥0);
2  while(x ≥0 && x < 10){
3    x++;
4  }
5  assert(x ≥10)
```

(b) Invariant: x ≥0 && x ≤10

**图 2-1 示例测试程序及对应期望循环不变式**

在示例程序中，*assume* 表示 *precondition*，*assert* 表示 *postcondition*，对于程序 2-1(a)来说，变量集 *V* 包含 *x, y* 两个变量，程序 2-1(b)包含 *x* 一个变量。在每个示例程序底部我们给出了可以证明该霍尔三元组的循环不变式。

总体算法如图 2-2 所示，首先我们随机生成变量集 *V* 的一组初始样本点，用 *SP* 表示，这样我们可以获得用于分类学习的一组初始样本。

**Algorithm 1** Algorithm *generateLoopInvariant()*

```
Output:
   Φ : 候选循环不变式
Procedure:
  1: SP = initGen(); // randomly generate samples for value set V
  2: While not time out do
  3:     call activeLearn(SP) to generate a candidate invariant Φ ;
  4:     if the program is verified with Φ  then
  5:         return "proved"
  5:     else add the counterexample into SP;
  6:     endif
  6: done
```

**图 2-2 基于机器学习的循环不变式自动生成算法**

生成第一组初始样本之后，对于任意一组 *SP* 中的取值 *s*，我们从 *s* 开始执行程序并且记录每一轮循环结束之后的变量值 *s'*，我们用 $s \Rightarrow s'$ 来表示存在 $i \geq 0$，使得 $\exists i \geq 0,\ s' = Body^i(s)\ and\ Body^k(s) \in Cond\ for\ all\ k \in [0, i)$，也就是说，对于从取值 *s* 执行之后每轮循环生成的值 *s'*都添加到 *SP* 当中。

将 SP 生成完成之后，我们可以将 *SP* 划分为四个不相交的集合：*CE, Positive, Negative* 和 *NP*。直观介绍，*CE* 包含可以证明霍尔三元组错误的边界样本；*Positve* 包含满足任何可以证明霍尔三元组的循环不变式要求的变量取值；*Negative* 包含不满足任何可以证明霍尔三元组的循环不变式的变量取值；*NP* 包含剩下的所有取值。下面我们给出正式的定义：

$$\begin{aligned} CE(SP) = \{ & s \in SP | \exists s_0, s'. \\ & s_0 \in Pre \wedge s_0 \Rightarrow s \Rightarrow s' \\ & \wedge s' \notin Cond \wedge s' \notin Post \} \end{aligned}$$

*CE(SP)*中的取值 *s*，是从满足 *Pre* 的取值 $s_0$ 开始并且在终止循环之后的值 $s'$ 不满足 *Post*，这一过程中的所有取值。如果 *CE(SP)*不为空，则霍尔三元组被证明是错误的。

$$\begin{aligned} Positive(SP) = \{ & s \in SP | \exists s_0, s'. \\ & s_0 \in Pre \wedge s_0 \Rightarrow s \Rightarrow s' \\ & \wedge s' \notin Cond \wedge s' \in Post \} \end{aligned}$$

*Positve(SP)*中的取值 $s$，是从满足 *Pre* 的取值 $s_0$ 开始，经过 0 次或多次迭代成为 $s$，并且最终在终止循环是迭代为 $s'$，值 $s'$ 满足 *Post*。让 *Inv* 表示任意满足条件的循环不变式，因为 $s_0 \in Pre$，并且 Inv 满足条件(1)，所以 $s_0 \in Inv$；因为 *Inv* 满足条件(2)，所以如果 $Body(s_0) \in Cond$，则 $Body(s_0) \in Inv$；因此我们证明了 $s \in Inv$。

$$\begin{aligned} Negative(SP) = \{ & s \in SP | \exists s_0, s'. \\ & s_0 \notin Pre \wedge s_0 \Rightarrow s \Rightarrow s' \\ & \wedge s' \notin Cond \wedge s' \notin Post \} \end{aligned}$$

*Negative(SP)*中的取值 $s$，是从不满足 *Pre* 的取值 $s_0$ 开始，经过 0 次或多次迭代成为 $s$，并且最终在终止循环是迭代为 $s'$，值 $s'$ 不满足 *Post*。我们将证明对于所有满足条件(1), (2), (3)的循环不变式 *Inv*，都存在 $s \notin Inv$。假设 $s \in Inv$，则根据条件(2)，必然存在 $s' \in Inv$，根据条件(3)，如果 $s' \in Inv$，则 $s' \in Post$，矛盾，故证明。

$$NP(SP) = SP - CE(SP) - Positive(SP) - Negative(SP)$$

*NP(SP)*包含了除了 *CE(SP)*, *Positve(SP)*和 *Negative(SP)*之外的所有取值，对于 *NP(SP)*中的取值，有可能满足也有可能不满足期望循环不变式，因此并不能直接应用到分类程序当中。

我们以图 2-1(a)为例说明 *SP* 的四个分类，假设我们初始随机生成了以下三个样本点: (1, 2), (10, 1)和(100, 0)，执行循环之后我们得到了三组样本点，分别是{(1,2), (11, 5)}, {(10, 1)}和{(100,0)}，需要注意的是对于后两个初始样例，循环一次都没有执行。分类之后，*CE(SP)*为空，*Positive(SP)*包含{(1,2), (11, 5)}，*Negative(SP)*包含{(100,0)}，*NP(SP)*中包含{(10, 1)}。

在生成了初始样本集并且分类之后，相当于给予对应集合相应的标签，则可以利用这组带标签的样本训练分类模型进行机器学习分类，得到结果为：

$$-0.0224x + 0.00126y + 1.240 \geq 0 \qquad (2.4.1)$$

得到候选循环不变式之后，开始验证其是否满足霍尔三元组，即是否满足条件(1), (2), (3)，也即利用约束求解和符号执行求解是否存在满足以下三个约束条

件的值：

$$Pre \wedge \neg\phi \tag{4}$$
$$sp(\phi \wedge Cond, Body) \wedge \neg\phi \tag{5}$$
$$\phi \wedge \neg Cond \wedge \neg Post \tag{6}$$

条件(5)中的 $sp(\phi \wedge Cond, Body)$ 表示将满足循环不变式和循环条件作为前置条件，执行循环程序之后的最强后置条件。如果以上三个条件不存在满足的样本点，则我们证明了候选循环不变式可以证明霍尔三元组，算法结束。如果存在满足条件的点，则将该点作为边界样本添加到初始样本集中重复整个算法。

根据式子(2.4.1)和 *Pre*，我们可以得到满足条件(4)的样本点(0,0)，因此生成的循环不变式不满足条件，重复迭代。

## 2.5 支持向量机监督学习算法

在 2.4 的算法介绍中，应用到了机器学习中的分类算法，由于我们需要得到分类模型并且需要严格分类，因此采用支持向量机监督学习算法(SVM)最为合适。支持向量机(Support Vector Machine)是一种二分类问题的监督学习算法，一般用于分类问题和回归分析问题。给定一组带有正负标签的数据，SVM 可以训练出一个分类模型将这组数据一分为二，然后对于新输入的数据 SVM 就可以利用学习到的分类模型对于数据正确归类。在支持向量机中，每一个样本点可以被看作一个 $p$ 维向量，线性分类即找到一个 $(p-1)$ 维的超平面来正确划分数据集。

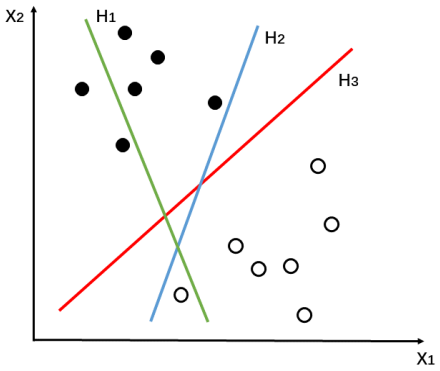



**图 2-3 分类超平面示意图**

如图 2-3 所示，黑色和白色代表数据的两组标签，从图中可以看出，H1 并

不能将数据分类，而 H2 和 H3 都可以对于数据进行划分，因此对于一组数据存在多个合理分类超平面。但是最合理的分类超平面应该可以将两类数据尽可能的区分开，换言之，超平面应该是定义在特征空间上的间隔最大的线性分类器，如图中 H3 所示。所以根据定义，SVM 学习策略即为间隔最大化，可以形式化表示为求解一个凸二次规划问题，因此 SVM 算法即为求解凸二次规划最优解问题算法。

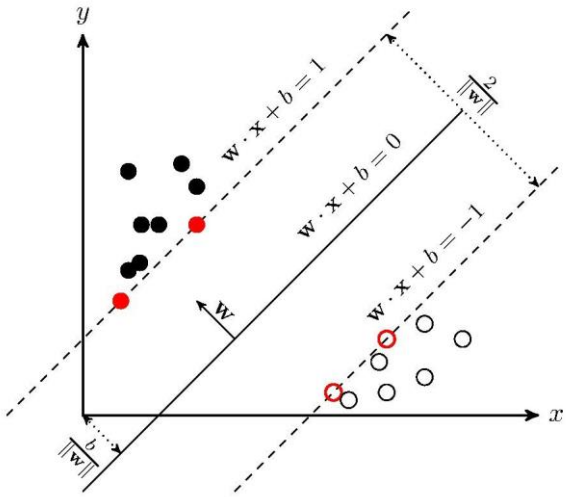



**图 2-4 SVM 算法超平面示意图**

如图 2-4 所示，$w \bullet x+b=0$ 即为几何间隔最大的唯一分类超平面，而 SVM 算法即为求解 $w \bullet x+b=0$ 中 $w,b$ 的值。在介绍具体推导求解过程之前，先进行一些基础定义说明：假设给定一组特征空间上的线性可分数据集 $T$，

$$T=\{(\boldsymbol{x}_1,y_1),(\boldsymbol{x}_2,y_2),...,(\boldsymbol{x}_N,y_N)\} \tag{2.5.1}$$

其中，$\boldsymbol{x}_i \subseteq R^n, y_i \in \{+1,-1\}, i=1,2,...,N$。

接下来定义几何间隔，对于数据集 $T$ 和待求解超平面 $w \bullet x+b=0$，定义样本点到超平面的几何间隔为：

$$\gamma_i = y_i\left(\frac{\omega}{\|\omega\|} \bullet \boldsymbol{x}_i + \frac{b}{\|\omega\|}\right) \tag{2.5.2}$$

则所有样本点到超平面的最小间隔为：

$$\gamma = \min_{i=1,2,...,N} \gamma_i \tag{2.5.3}$$

根据以上说明，则 SVM 学习算法可以表示为求解以下约束最优化问题：

$$\max_{w,b} \gamma$$
$$s.t.\ y_i\left(\frac{\omega}{\|\omega\|} \bullet \mathbf{x}_i + \frac{b}{\|\omega\|}\right) \geq \gamma, i = 1,2,...,N \quad (2.5.4)$$

对约束条件两边同除以$\gamma$，并且将标量按如下规则简化：

$$\omega = \frac{\omega}{\|\omega\|\gamma},\ b = \frac{b}{\|\omega\|\gamma}$$

又因为最大化$\gamma$，等价于最大化$\frac{1}{\|\omega\|}$，为了之后化简结果更为精简，我们将$\frac{1}{\|\omega\|}$就等价于$\frac{1}{2}\|\omega\|^2$代入（2.4.4）中得到最终的含有不等式约束的凸二次规划问题：

$$\min_{w,b} \frac{1}{2}\|\boldsymbol{\omega}\|^2$$
$$s.t.\ y_i(\boldsymbol{\omega} \bullet \mathbf{x}_i + b) \geq 1, i = 1,2,...,N \quad (2.5.5)$$

然后使用拉格朗日乘子法得到该凸二次规划问题的对偶问题并求解即可得到最终分类超平面。

如果数据集在 $p$ 维线性不可分，SVM 提供了两种方式：将数据映射到高维空间进行分类和采用非线性核函数。虽然将数据映射到高维空间会存在维度指数增长问题，但是由于在 MIG 系统中我们需要计算得到具有可读性的分类超平面，所以我们仍然采用数据映射方式解决非线性问题。举例说明：我们需要将给定数据集$\{x \mapsto 2, y \mapsto 1\}$映射到最高次为 2 的空间，则映射后的数据为$\{x \mapsto 2, y \mapsto 1, x^2 \mapsto 4, xy \mapsto 2, y^2 \mapsto 1\}$，然后在映射后的数据集上应用 SVM 算法进行分类学习，得到线性分类器。通过数学推导可得，在高维空间得到的线性分类器和原始空间的多项式分类器等价，因此完成了对于原始非线性可分数据的分类。

## 2.6 选择取样（Selective Sampling）

在现实生活中，获得带准确标签的训练样本成本很高且耗时巨大，在一些领域，比如工业建模过程，一个训练样本就需要耗费几天时间并且花费上千美元；

在另一些领域，比如邮件分类与过滤，获得样本较为轻松，但是也需要工作人员花费很多时间对邮件增添标签。因此研究人员提出了主动学习方式，在样本数据有限的前提下，可以极大的减少准确分类所需样本数量，甚至可以减少计算要求和时间复杂度。而选择取样是主动学习的经典算法之一。

选择取样主要有两种方式，第一种是贪心最优策略，可以理解为对分类后空间的每个点得到正确的概率，具体做法为将所有样本点映射到垂直于分类超平面的轴上，并对他们做逻辑回归来得到该类别的概率，最后对空间上的误差概率进行积分，并通过对一些测试样本假设一个分布加权，最终可以计算出该分类器的期望错误率。这种方法可以保证在概率上正确，但是可操作性不强，基本无法实现，因此我们一般采用第二种启发式选择取样方式。启发式选择取样如图 2-5 所示，通过在预测学习得到的分类平面上选择样本点，然后对于样本点添加标签并加入到原始样本集中重新分类学习，通过多次迭代可以越来越趋近于真实值，最终得到准确结果。

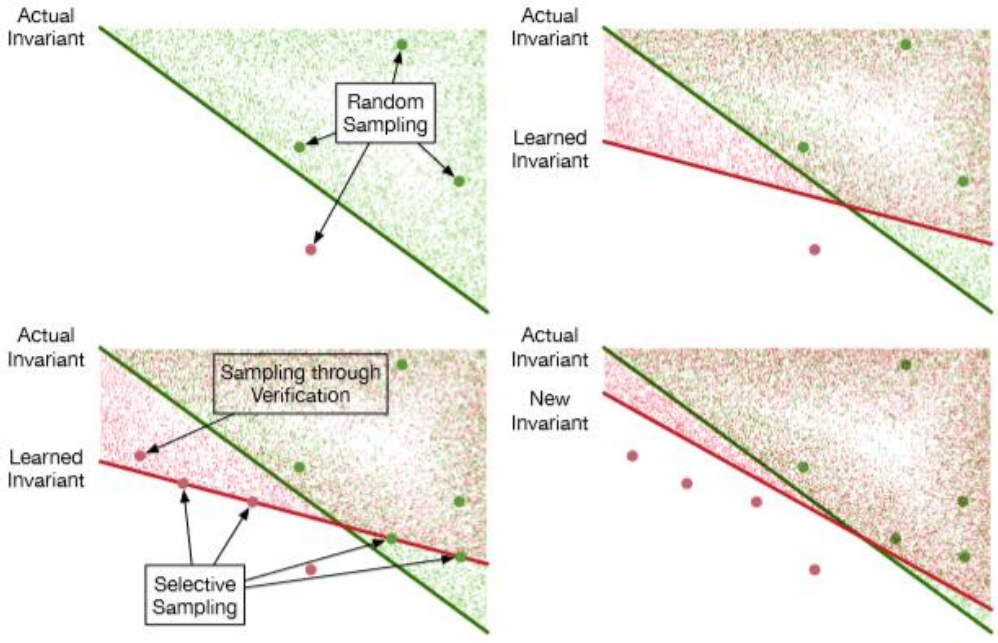



**图 2-5 启发式选择取样示例图**

将启发式选择取样应用到 2.4 中基于机器学习的循环不变式的自动生成的学习部分，可以在每一轮迭代中，通过主动学习生成边界样本点，使得候选循环不变式更快接近真实值，在加速整体算法收敛的同时可以提高生成的循环不变式精

度。

## 2.7 工具概述

MIG 工具主要约束求解工具 Z3 和符号执行工具 KLEE，下面将分别介绍它们的基本原理和使用方法：

### 2.7.1 约束求解工具 Z3

SMT(satisfiability modulo theories)求解工具通过求解满足性问题完成生成过程，可以用来解决外部静态检查、抽象预测、测试样例生成等问题。Z3 是微软研究院推出的新型 SMT 求解工具，可以解决许多软件验证和软件分析中的问题。Z3 集成了 DPLL SAT 求解器，处理等式和函数的理论求解器，用于求解算术、数组等的卫星求解器和用于求解量化器的 E 匹配抽象机，因此 Z3 的求解范围较为全面，我们使用的主要有以下几种功能：

- **简化**：将输入的式子进行尽可能的简化，包括标准代数约简规则，如 $p \wedge true \mapsto p$ ，也可以进行有限的上下文简化，如 $x = 4 \wedge q(x) \mapsto x = 4 \wedge q(4)$ 。
- **SAT Solver**：Z3 使用最新的 SAT Solver 技术解决布尔类型表达式拆分问题，集成了包括典型搜索剪枝算法、非时间线性关系回溯算法等。
- **模型生成**：Z3 可以生成约束求解模型，即给出满足约束条件的变量对应取值和为谓词和函数符号生成的部分函数图。

### 2.7.2 符号执行工具 KLEE

许多类型的软件错误如果不执行断码的话很难发现并定位，但是如果使用具体值来执行则需要大量测试用例进行定位，成本较高，因此用符号代替具体值运行程序。当程序执行到分支语句时，系统会执行分支的每一种情况，对于每一条可能的路径维护一个路径约束。当路径终止的时候或者遇到 bug 时，则可以通过求解路径约束得到一组测试用例。KLEE 是一种符号执行工具，它能够自动生成测试，并且在高复杂度程序上仍然有较高覆盖率。

```
1  int get_max(int a, int b){
2    if(a>=b) return a;
3    else return b;
4  }
```

**图 2-6 KLEE 示例程序 get_max.c**

以图 2-6 中的测试程序为例，介绍具体使用方法如下：

1. 使用 klee_make_symbolic() (在 klee/klee.h 中定义)符号化输入变量，该函数包含三个参数：变量运行时地址，变量大小以及命名。使用图 2-7 中示例 main()函数将变量 a, b 符号化，并调用图 2-6 中测试程序。

```
1  int main() {
2    int a, b;
3    klee_make_symbolic(&a, sizeof(a), "a");
4    klee_make_symbolic(&b, sizeof(b), "b");
5    return get_max(a, b);
6  }
```

**图 2-7 符号化 KLEE 测试用例变量的 main 函数示例**

2. 使用 clang 将源文件编译为 LLVM bitcode 代码

```
$ clang -emit-llvm -c get_max.c
```

3. 使用 KLEE 运行 bitcode 得到执行结果，运行命令如下所示：

结果中包含总指令数、总路径数、生成的测试用例数以及对应的测试用例，测试用例存储在 klee-last/文件夹中

```
$ klee get_max.bc
```

## 2.8 本章总结

本章主要介绍形式化验证、循环不变式、霍尔三元组、支持向量机监督学习算法等机器学习相关背景知识。首先指出软件测试在当今信息社会的重要性，重点介绍关键概念循环不变式、霍尔三元组和支持向量机、选择取样等机器学习相

关背景，然后介绍基于机器学习的循环不变式自动生成的相关概念和算法，具体实现将在第三章中详细介绍。

# 第三章 基于机器学习和选择取样的循环不变式自动生成方法设计

第三章将按照 2.4 节中对于基于机器学习的循环不变式自动生成算法的思路，结合 2.6 节选择取样算法优化学习模块，完成对于循环不变式自动生成工具 MIG 的整体设计，包括模块划分、模块间交互、具体算法等。按照 2.4 中 Algorithm1 的算法流程，将工具分为初始化、学习、验证和人机交互四个功能模块。初始化模块由测试文件和配置文件出发，通过约束求解和随机取样相结合的方式，生成初始测试样本及标签；学习模块采用 SVM 算法，根据初始样本结合选择取样的主动学习方式多次迭代生成候选循环不变式；验证模块需要验证生成的候选循环不变式是否存在满足 2.4 中 (4) (5) (6) 条件并判断结束或继续迭代。并且为了进一步加速算法收敛速度，我们提出引入人机交互模块，优化整体算法性能。下面将进行具体说明：

## 3.1 整体框架

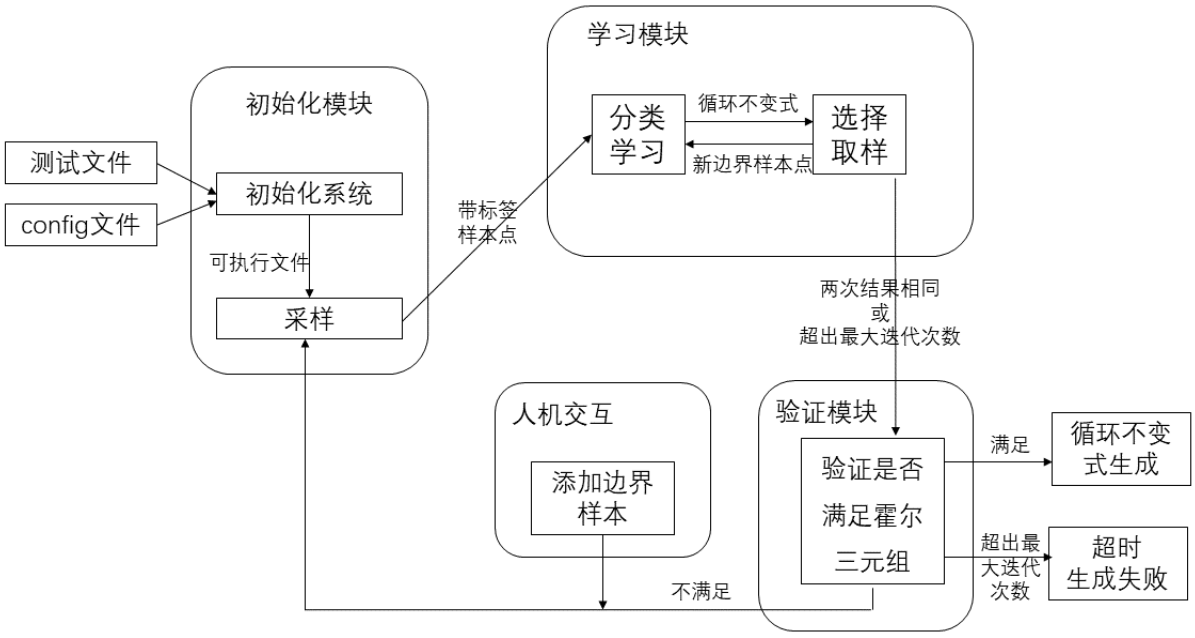



**图 3-1 基于机器学习和选择取样循环不变式自动生成工具整体框架**

MIG 工具主要模块划分与交互如图 3-1 所示，首先初始化模块利用输入的测试文件和 config 文件初始化系统，完善与测试文件相关的配置；系统初始化结束后则可以利用生成的可执行文件进行数据采样，生成带标签的样本供学习模块使用。学习模块通过 SVM 算法对初始样本进行分类学习，然后选择取样在生成的候选循环不变式基础上生成边界样本重复迭代，直到两次生成的循环不变式相同或超过最大迭代次数，将最后一次生成的循环不变式传给验证模块进行验证。如果验证满足霍尔三元组，则完成最终循环不变式的生成；若不满足，如果超出整体算法最大迭代次数则输出超时，反之则将边界样本加入到初始样本中重复算法过程。人机交互模块在验证模块之后，如果验证结果不满足，则询问用户是否进入人机交互模块，由用户输入边界样本，与验证模块生成的样本一起加入到初始样本集，加速算法收敛。

下面将对各个模块进行具体阐述。为了更加清楚的介绍不同模块的具体设计与预期结果，我们以图 2-1 中的样例程序进行说明：

## 3.2 初始样本及标签生成

初始样本及标签生成同时也是 MIG 工具初始化模块，因此首先需要抽取测试文件和 Config 文件中的相关参数和配置，生成之后所需的一系列 CPP 文件和可执行文件，具体生成过程将在 4.2 节进行详细介绍。

完成相关初始化工作之后，我们开始取样生成初始样本并添加标签，具体算法流程如图 3-2 所示。

**Algorithm 2** Algorithm *initGen*()

```
Output:
    f : 初始样本文件
Procedure:
  1: s = solve(precondition)
  2: outputPositive(f, s)
  3: s = solve(not precondition)
  4: outputNegative(f, s)
  5: while positive_num < maxNum or negative_num < maxNum do
  6:     s = randomGen()
  7:     if s can satisfy precondition then
  8:         outputPositive(f, s)
  9:     else
 10:         outputNegative(f, s)
 11:     endif
 12: done
```

**图 3-2 初始样本生成算法流程**

算法 Algorithm2 的目标是生成 *Positive(SP)*和 *Negative(SP)*。由 2.4 节对 *Positive(SP)*和 *Negative(SP)*的定义可知，我们需要找出满足 *Pre* 且在循环结束之后满足 *Post* 作为 *Positiv(SP)*的值，找出不满足 *Pre* 且在循环之后不满足 *Post* 作为 *Negative(SP)*的值。

由算法可以看出，我们采用约束求解与随机取样相结合的方式生成初始样本。因为一方面，在 *Pre* 的限制条件非常严格的情况下，如$(x==0\&\&y==0)$，单纯使用随机生成有可能不能找出符合条件的样本点或生成时间较长；另一方面，如果单纯使用约束求解，会对 *Pre* 的复杂度存在要求，降低 MIG 系统的适用范围。因此约束求解与随机生成相结合可以互为补充，相辅相成地完成初始样本的生成过程。下面将对算法细节进行说明：

在算法中，*solve*(precondition)是指利用约束求解工具解出满足 *Pre* 的变量取值，找出起始值之后在 *outputPositive*(f, s)中执行循环并写入文件。对于 *Negative(SP)*同理。约束求解方式执行之后，判断已经生成的 *Positive(SP)*和 *Negative(SP)*的数量是否达到预设最小值，如果没有达到，则开始随机生成过程。首先利用随机数为每一个变量生成一个相应的值，然后验证是否满足 *Pre*，如果

满足，则调用 *outputPositive*(f, s)完成后续生成工作；反之，则调用 *outputNegative*(f, s)。*outputPositive*(f, s)和 *outputNegative*(f, s)的算法如图 3-3，3-4 所示。

**Algorithm 3** Algorithm *outputPositive*(f, s)

```
Input:
    f : 初始样本文件
    s : 起始样本点集
Procedure:
  1: for i in s do
  2:     add all valuation i' such that i⇒i' by execute loop in f
         with label "+1"
  3:     let q be i ⇒ q and q can't satisfy loop condition
  4:     if q can't satisfy postcondition then
  5:         report exist CE(SP) and end process
  6:     endif
  7: done
```

**图 3-3 outputPositive(f, s)算法**

**Algorithm 4** Algorithm *outputNegative*(f, s)

```
Input:
    f : 初始样本文件
    s : 起始样本点集
Procedure:
  1: for i in s do
  2:     add all valuation i' such that i⇒i' by execute loop in f
         with label "-1"
  3:     let q be i ⇒ q and q can't satisfy loop condition
  4:     if q can satisfy postcondition then
  5:         remove i and all value generated by i in f
  6:     endif
  7: done
```

**图 3-4 outputNegative(f, s)算法**

*outputPositive*(f, s)算法将每一轮迭代生成的结果加入到 *Positive(SP)*中，当循环结束时，验证结束的变量值是否满足 *Post*，如果不满足，则存在 *CE(SP)*，霍尔三元组错误，整体算法结束；如果满足，则将新生成的所有变量值写入到初始样本文件中，并且标签设置为"+1"。 *outputNegative*(f, s)算法与之类似，

如果循环结束时的变量值满足 *Post*，则该变量为 *NP(SP)*，将相关变量从 *Negative(SP)*中删除。

我们使用图 2.n(a)的测试程序为例对算法进行详细说明。该测试霍尔三元组 *Pre* 为 $x < y$，*Post* 为 $y \le x \le y+16$。首先对于 *Pre* 进行约束求解，生成满足 *Pre* 的一组变量 $\{x \mapsto 0, y \mapsto 1\}$，代入 *outputPositive*(f, s)执行。每一轮循环之后的变量值为 $\{\{x \mapsto 10, y \mapsto 4\}\}$，因为只有一轮迭代故只生成了一组解。循环结束时的变量取值为 $\{\{x \mapsto 10, y \mapsto 4\}\}$，符合 *Post*，因此生成的变量属于 *Positive(SP)* 的范畴，可以写入到文件中。则此时 *Positive(SP)* 中的值为 $\{\{+1\ x \mapsto 0, y \mapsto 1\}, \{+1\ x \mapsto 10, y \mapsto 4\}\}$。利用约束求解生成 *Negative(SP)*的过程类似，在此不再赘述。

假设 *Positive(SP)*数量的预设最小值为 3，则我们将进入随机生成方法增加 *Positive(SP)*的取值。首先通过随机数生成一组取值 $\{\{x \mapsto -5, y \mapsto 10\}\}$，符合 *Pre*，因此代入 *outputPositive*(f, s)执行。每一轮循环之后的变量值为 $\{\{x \mapsto 2, y \mapsto 13\}, \{x \mapsto 12, y \mapsto 16\}, \{x \mapsto 22, y \mapsto 19\}\}$。循环结束时的变量取值为 $\{\{x \mapsto 22, y \mapsto 19\}\}$，符合 *Post*，因此生成的变量属于 *Positive(SP)*的范畴，可以写入到文件中。则此时 *Positive(SP)*中的值为：

$$\{\{+1\ x \mapsto 0, y \mapsto 1\}, \{+1\ x \mapsto 10, y \mapsto 4\}, \{+1\ x \mapsto -5, y \mapsto 10\},$$
$$\{+1\ x \mapsto 2, y \mapsto 13\}, \{+1\ x \mapsto 12, y \mapsto 16\}, \{+1\ x \mapsto 22, y \mapsto 19\}\}$$

*Positive(SP)*中值的数量满足最小值的要求，*Positive(SP)*的生成过程完成。当 *Negative(SP)*的数量也满足最小值要求的时候，初始化样本及标签生成模块结束。

## 3.3 候选循环不变式生成

候选循环不变式生成模块是利用支持向量机分类学习算法结合选择取样这一主动学习方式，生成循环不变式并提供给验证模块检验，对应 2.4 节算法 Algorithm1 中的 *activeLearn*(SP)，具体算法流程如图 3-5 所示。

**Algorithm 5** Algorithm *activeLearn*(SP, d)

```
Input:
   SP: 初始样本集
    d:  循环不变式最高次幂
Output:
   Φ: 候选循环不变式
Procedure:
  1: While not exceed maxIterationNum do
  2:     let SP′ = reflecting(SP, d);
  4:     let Φ be a set of candidates generated by classify(SP′);
  5:     if( Φ is the same as last iteration) return Φ;
  6:     add selectiveSampling(Φ, num) into SP;
  7:     add all valuations s′ such that s⇒s′
         for some s∈SP into SP;
  8: done
```

**图 3-5 候选循环不变式生成算法**

算法 Algorithm5 首先将数据和变量映射到指定维度空间，用以完成非线性的循环不变式的准备工作，扩展 MIG 系统的应用场景，如果需要生成线性循环不变式，则设置映射到想同维度空间即可，我们先以线性循环不变式的生成为例讲解算法流程。接下来对于映射后的数据集采用支持向量机监督学习算法进行分类学习，计算得到分类超平面，从而得到候选循环不变式，判断循环不变式与上一次迭代生成的是否相等，如果相等则说明迭代收敛，返回；反之，运用选择取样的方式对于新生成的循环不变式添加边界样本，并将边界样本添加到初始样本集中，重复迭代过程。选择取样函数如图 3-6 所示。

**Algorithm 7** Algorithm *selectiveSampling*(Φ, num)

```
Input:
   Φ: 候选循环不变式
  num:  新生成节点数
Output:
  SP′: 新生成counterexamples
Procedure:
  1: for i = 1 to num do
  2:     if Φ only contains one variable
  3:         return solvingEquation( Φ );
  3:     else for the first n-1 variables in random generate value
  4:         add solvingEquation( Φ ) to SP′ based on generated values;
  5: done
  6: return SP′;
```

**图 3-6 选择取样算法**

通过选择取样算法，我们将在候选循环不变式中选取样本点加入到初始样本集中，重复分类学习算法直至收敛，用以减少整体迭代次数，并且提高生成的循环不变式精度。在候选循环不变式上选取样本时，如果循环不变式只含有一个变量，因为我们需要是的循环不变式等于 0 然后求解，则该等式有确定解，因此我们使用约束求解工具直接解出结果，然后将结果按照初始化样本集标签模块中由生成的初始样本点添加其余循环过程中点的方式补充进初始样本集中。如果循环不变式含有多于一个变量，则变量数量多于等式数量，没有确定解，因此我们对于前 n-1 个变量采用随机生成的方式确定它们的值，然后用取值替代它们在循环不变式中的位置进行约束求解，解出第 n 个变量的值。这样重复 num 次（num 可在 Config 文件中配置），最终可以添加至少 num 组初始样本加速循环不变式收敛。

以图 2.n(a)的测试程序为例对于循环不变式生成算法进行详细介绍，在 3.2 中我们生成了它的 *Positive(SP)*和 *Negative(SP)*样本，通过支持向量机分类算法得到循环不变式为 $-0.12x+0.12y+0.7\geq 0$，则通过随机生成 x 取值为 0，通过求解得到 y 的取值为 6，则我们生成了一组起始样本 $\{x\mapsto 0, y\mapsto 6\}$，可以验证这组取值满足 Pre，按照 3.2 中的方法将 $\{\{+1\ \ x\mapsto 0, y\mapsto 6\},\{+1\ \ x\mapsto 10, y\mapsto 9\}\}$ 最终添加到初始样本集中。在此应用支持向量机分类算法得到 $-0.2x+0.2y+1.7\geq 0$。重复这一过程直至两次生成的循环不变式相等。

之前的介绍是基于生成线性循环不变式的算法，如果需要生成非线性循环不变式，则需要首先将数据映射到高维空间，然后进行分类学习和选择取样的过程。映射数据算法如图 3-7 所示。

**Algorithm 6** Algorithm *reflecting*(SP, d)

```
Input:
   SP: 样本点, SP = {x, y, z}
    d: 最高次次数
Output:
    polydata: 映射到高维空间的样本点
Procedure:
  1: function CalcValue(var, degree)
  2:     for i = 1 to degree do
  3:        result = result * var
  4:     end for
  5:     return result
  6:  function GeneratePolyData(degree, SP, polydata)
  7:     res[][]
  8:     for i in SP do
  9:        dataTemp[]
 10:        dataTemp.add(CalcValue(i, degree))
 11:        for j = degree - 1 to 1 do
 12:           prefix = CalcValue(i, j)
 13:           remain = degree - j
 14:           for m = i + 1 to SP.size do
 15:              for n = 0 to polydata[remain-1][m].size do
 16:                 temp = prefix
 17:                 temp *= polydata[remain - 1][m][n]
 18:                 dataTemp.add(temp)
 19:              end for
 20:           end for
 21:         res.add(dataTemp)
 22:     end for
 23:  return polydata
 24:  function OutputPolyData(degree, SP)
 25:     polydata[][][]
 26:     for i in SP do
 27:        temp[]
 28:        temp[0] = i
 29:        polydata[0].add(temp)
 30:     end for
 31:     for i = 2 to degree do
 32:        GeleratePolyData(i, SP, polydata)
 33:     end for
 34:  return polydata
```

**图 3-7 映射数据到高维空间算法**

映射算法总体原理为：因为将数据映射到高维空间，则在这个空间生成的循环不变式中的变量，可能从 1 次到 n 次都有存在，所以算法从低次向高次依次生成数据。以 x, y, z 生成最高次为 3 次数据为例，生成过程中数据存储结构如图 3-8 所示：

元素及其顺序：x, y, z

| 开始元素 | x | y | z |
|---|---|---|---|
| 最高次为1 | {x} | {y} | {z} |
| 最高次为2 | {x*x, x*y, x*z} | {y*y, y*z} | {z*z} |
| 最高次为3 | {x*x*x, x*x*y, x*x*z, x*y*y, x*y*z, x*z*z} | {y*y*y, y*y*z, y*z*z} | {z*z*z} |

**图 3-8 映射变量{x,y,z}到 3 维空间样例**

首先生成一次数据存储在 polydata[0]中，即{{x}，{y}，{z}}；然后二次数据对变量依次遍历，即从 x 开始到 z 结束，首先对于当前变量，不失一般性假设为 x，生成 x 的当前次幂，即 x*x，然后 x 的当前次幂递减，因为次数总和要等于 degree，因此剩余次数为 degree – x_degree，对于排在 x 之后的变量的对应剩余次数位置存储的数据，依次与 x 相乘，即 x*y, x*z，结果存在 polydata[1][0]中（即次数为 2 的第一个变量可以组成的结果）；同理，对于三次，首先生成 x*x*x，然后 x 的最高次递减为 2，则在 polydata[1]大于 x 的位置查找，即 polydata[1][1] = {y}，polydata[1][2] = {z}，得到结果 x*x*y, x*x*z，最后 x 最高次为 1，在 polydata[2]中查找，polydata[2][1] = {y*y, y*z}, polydata[2][2] = {z*z}，得到 x*y*y, x*y*z, x*z*z。其余变量同理，这样即可把变量和数据映射到高维空间。

## 3.4 候选循环不变式验证

具体算法如图 2.n 算法中验证部分所示，主要验证是否存在满足条件(4), (5), (6)的样本点，如果存在，则说明循环不变式不能验证霍尔三元组，将此样本作为边界样本添加到初始样本集中重复整体算法迭代。

$$Pre \wedge \neg\phi \tag{4}$$
$$sp(\phi \wedge Cond, Body) \wedge \neg\phi \tag{5}$$
$$\phi \wedge \neg Cond \wedge \neg Post \tag{6}$$

条件(5)中的 $sp(\phi \wedge Cond, Body)$ 表示将满足循环不变式和循环条件作为前置条件，执行循环程序之后的最强后置条件，我们采用符号执行的方式将循环不变式和循环条件作为条件限制，执行一轮迭代之后判断是否存在不满足循环不变式的点，从而完成条件(5)的判断。

仍然以图 2-1(a)的测试程序为例进行详细介绍，假设在 3.3 中我们最终生成的循环不变式为$-0.2x+0.2y+1.7\geq 0$，则对于该循环不变式判断是否存在满足以上三个条件的点：

- 条件(4)：通过约束求解，具体约束限制如下：

$$\begin{cases} -0.2x+0.2y+1.7\geq 0 \\ x\geq y \end{cases}$$

易得$\{x\mapsto 0, y\mapsto 0\}$满足条件。在实际算法中，对于三个条件进行短路算法，一旦有一个条件满足，则直接返回该点，进行后续操作。按照 3.2 中由起始点添加到初始样本集中的算法说明，该点不满足 Pre，在执行循环后点仍为$\{x\mapsto 0, y\mapsto 0\}$满足 Post，因此该点属于 NP(SP)，无法添加到初始样本中。

这是“先猜后证”方法中验证过程的随机性可能导致算法无法收敛或循环不变式精度不高的情况，因此我们在 3.5 节中引入人机交互模块，可以为用户提供补充数据的渠道，解决这一问题。

正常算法中如果存在满足条件(4)的点则终止验证过程，但这里我们为了介绍的完整性，继续以 2-1(a)为例说明条件(5)(6)的验证过程

- 条件(5)我们采用符号执行的方式，符号执行伪代码如下：

```
1  assume(-0.2*x+0.2*y+1.7 ≥ 0)
2  assume(x<y)
3  if(x<0) x:=x+7;
4  else x:=x+10;
5  if(y<0) y:=y-10;
6  else y:=y+3;
7  assert(-0.2*x+0.2*y+1.7 < 0)
```

**图 3-9 验证条件(5)符号执行伪代码**

易得$\{x\mapsto -2, y\mapsto -1\}$满足条件。按照 3.2 中由起始点添加到初始样本集中的算法说明，该点满足 Pre，在执行循环后点仍为$\{x\mapsto 5, y\mapsto -11\}$满足 Post，因此属于 Positive(SP)，将$\{\{+1\ x\mapsto -2, y\mapsto -1\}, \{+1\ x\mapsto 5, y\mapsto -11\}\}$添加到初始样本中。

- 条件(6)采用约束求解方式，具体约束限制如下：

$$\begin{cases} -0.2x+0.2y+1.7\geq 0 \\ x\geq y \\ x<y \cup x>y+16 \end{cases}$$

求解得不存在符合条件的点。

## 3.5 人机交互

正如 3.4 条件(4)的验证过程中可能出现的问题，完全依靠机器自身进行纠偏可能会存在误差；同时有时候用户可以观察到更加明显的可加快收敛的点，通过人机交互可以使得整体算法更快收敛，具体实验对比在第五章中有详细说明，这里我们通过图 3-10 给出更直观的对比解释：

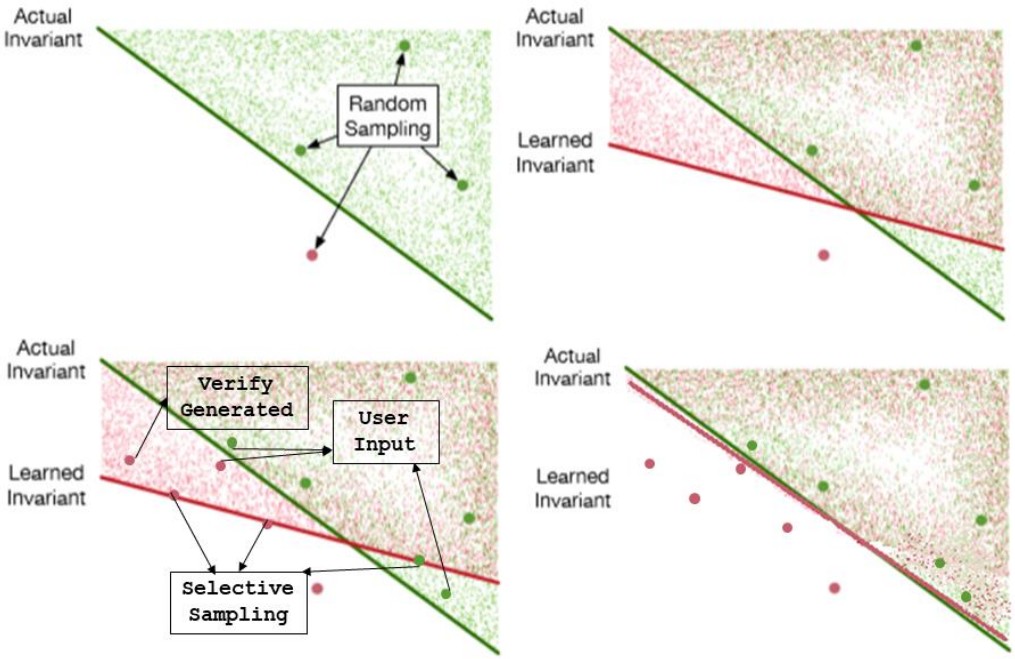



**图 3-10 人机交互加速收敛示意图**

从图片中可以直观看出人机交互可以显著加快收敛速度，但同时也看出人机交互获得良好的效果的前提是，用户需要对当前循环不变式及霍尔三元组非常熟悉，才能更为精确的生成有区分度的样本，如果大量增加无意义样本，反而会增加算法收敛时间。

## 3.6 本章总结

第三章中我们介绍了基于机器学习和选择取样的循环不变式自动生成方法的系统设计，包括模块划分、模块间交互、具体算法等。MIG 工具分为初始化、学习、验证和人机交互四个功能模块，重点介绍了初始样本生成、候选循环不变式生成、选择取样、数据映射登算法，为之后 MIG 工具的实现提供了高层框架结构。

# 第四章 基于机器学习和选择取样的循环不变式自动生成工具设计与实现

第四章我们将详细介绍基于机器学习和选择取样的循环不变式自动生成工具的具体实现。我们将这一工具命名为 MIG(Machine Learning based Invariant Generation)，该工具由 shell 脚本语言和 c++语言编写，应用约束求解工具 Z3，符号执行工具 KLEE 和 LIBSVM 支持向量机分类工具，最终实现了一个面向 c 和 c++程序的循环不变式自动生成工具。用户只需要按照 5.1.3 中的说明输入霍尔三元组测试文件，更改相关自定义配置，就可以得到满足该霍尔三元组的循环不变式。在第四章的说明过程中，我们以图 2-1(a)测试程序为例进行说明，图 2-1(a)为 MIG 工具中 Benchmark/01 文件。

## 4.1 顶层文件架构与关系

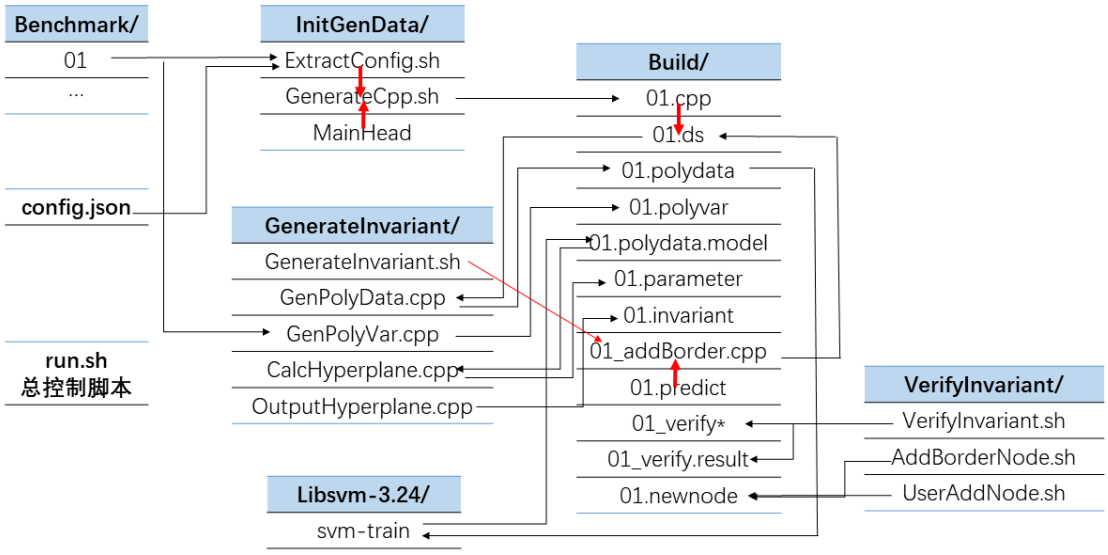



**图 4-1 MIG 工具顶层文件架构与关系**

MIG 工具顶层文件架构与关系如图 4-1 所示，运行时用户将测试文件作为参数调用 run.sh 即可生成满足条件的循环不变式。Build 文件夹存储所有运行过程

中生成的中间文件，每个测试文件以自己的文件名前缀在 Build 文件夹下命名一个独立的文件夹，比如 Benchmark/01 会在运行过程中建立 Build/01/文件夹，在 Build/01/文件夹中存储一系列如初始样本 01.ds，最终生成循环不变式 01.invariant 等等中间或最终结果。

## 4.2 结合约束求解和随机生成的初始样本生成

在开始整体算法流程之前，需要先检查 Build 文件夹中是否已经存在以当前测试文件名前缀命名的文件夹，如果存在则删除，这样可以方式由于同名测试文件或之前的生成结果干扰本次过程。同时检查输入参数是否正确，是否存在输入测试文件等。初始样本生成部分的文件及函数关系如图 4-2 所示：

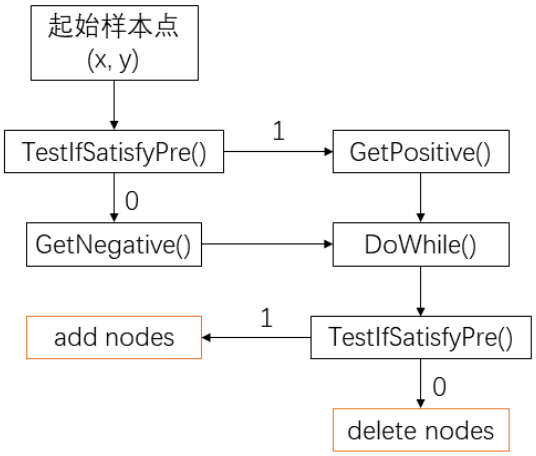
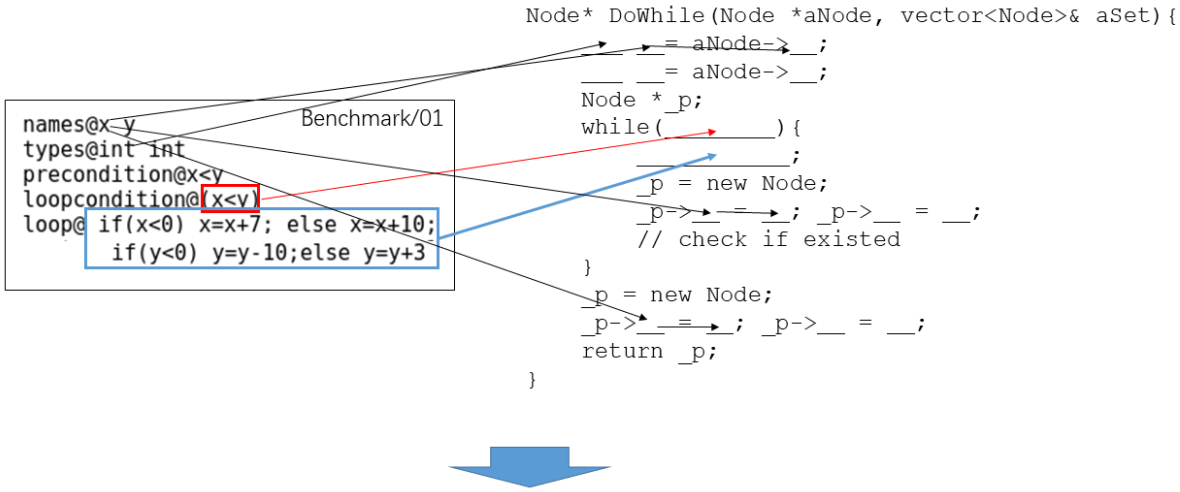



**图 4-2 初始样本生成 CPP 文件函数关系**

给定起始样本点，首先判断是否满足 *Pre* 条件，如果满足则通过 GetPositive()函数将生成的样本点加入到 PositiveSet 当中；反之，通过 GetNegative()加入到 NegativeSet 当中。在 GetPositive()和 GetNegative()当中都会调用 DoWhile()函数，用来执行循环并将每一轮迭代结果添加到对应 set 中，当循环结束后，判断是否满足 *Post*，如果满足则将 Node 添加到初始样本中；反之，删除相应节点。

初始样本最终是由如图 4-2 所示 CPP 文件编译得到的可执行文件生成的，但是我们需要通过 shell 脚本将 Config.json 以及 test 文件中的相关配置，类似于填空的方式添加到 CPP 文件中，从而完成整个过程。以 Benchmark/01 为例，图 4-3 演示了如何从 Test 文件中提取元素并生成 DoWhile 函数部分。




```
39 Node* DoWhile(Node *aNode, vector<Node>& aSet) {
40     int x = aNode->x;
41     int y = aNode->y;
42
43     Node *_p;
44     while((x<y)){
45          if(x<0) x=x+7; else x=x+10; if(y<0) y=y-10;else y=y+3;
46
47         _p = new Node;
48         _p->x = x;
49         _p->y = y;
50
51         vector<Node>::iterator it = find(aSet.begin(), aSet.end(), *_p);
52         if(it == aSet.end()) {
53             aSet.push_back(*_p);
54         }
55     }
56     _p = new Node;
57     _p->x = x;
58     _p->y = y;
59     return _p;
60 }
```

**图 4-3 DoWhile 文件生成示意图**

从图 4-3 中我们可以看出 shell 脚本提交关键元素的基本方式，并且 DoWhile 函数会返回循环结束时的样本点取值。同时注意到 DoWhile 函数的样本点参数采用的是 struct Node 形式传递，采用结构体进行样本点的存储主要是基于性能和可扩展性的考虑。由 3.2 节中对于初始样本生成的算法可以看出，在生成 *Positive(SP)*的时候，我们需要首先判断该点是否满足 *Pre*，然后在每一轮迭代结束时的样本点取值同样需要加入 *Positive(SP)*中，最后在循环结束的时候判断跳出循环的样本点是否满足 *Post*，如果满足则正常执行，但如果不满足则刚刚加入 *Positive(SP)*中的样本点都要去除。

为了实现这样的功能，主要有两种思路：第一种为每新生成一个 *Positive(SP)* 的样本点直接写入初始样本文件中，并记录起始样本点，如果发现最终不符合 *Positive(SP)*的要求，再从文件中起始样本点位置删除到文件末尾。这样在单线程系统下没有问题，但是如果需要将 MIG 工具扩展到多线程环境之下则会出现重复读写或擦除问题从而导致错误。因此我们采用第二种思路，将生成过程的 *Positive(SP)*存放在内存中，如果确定满足要求，则再将其写入文件中。因为 MIG 工具对初始样本的数量要求不高，一般数目都会控制在 1000 以内，所以并不会对内存造成很大的负担；同时这样的设计可以支持多线程环境下的运行要求。因此最终决定以结构体形式存储样本点，方便整体实现。具体结构体生成过程如图 4-4 所示。

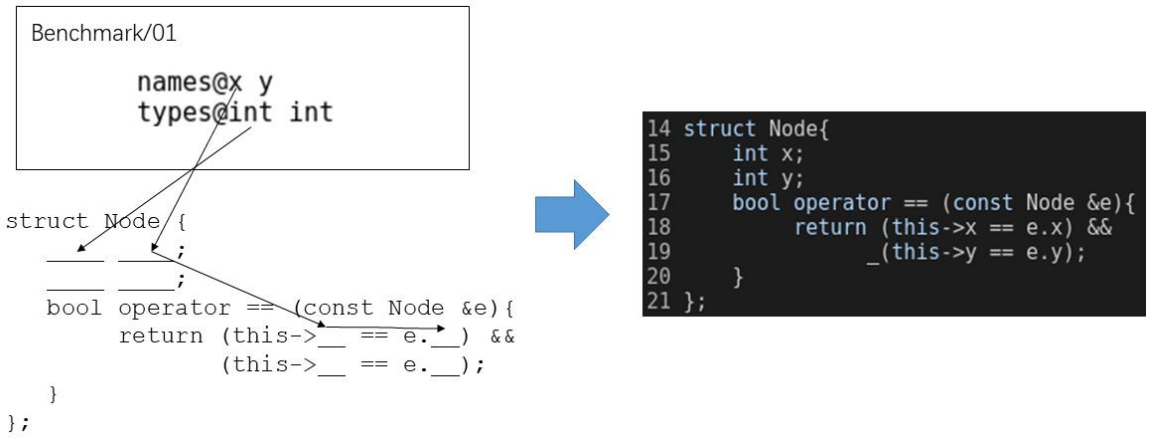



**图 4-4 MIG 工具数据结构体生成过程**

相关处理函数完成之后，开始 main 函数的编写。由 3.2 节介绍可知，我们需要先利用约束求解生成初始样本，然后再进行随机取样。

在这里我们使用约束求解工具 Z3 进行样本的生成，调用 Z3 中 solve 类进行求解，并使用 solve.add()函数添加约束条件。约束条件添加完成之后调用 solve.check()函数检查是否存在满足条件的样本点，如果存在，则相关信息存储在 model 类中，利用 solve.get_model()函数获得相关信息。这个过程中同样需要利用脚本将相关限制条件“填空”到相应位置，最终在 CPP 中利用约束求解生成 *Positive(SP)*的重点部分代码如图 4-5 所示。

```
112     // precondition
113     s.add(x<y);
114
115     switch(s.check()) {
116         case z3::unsat: return -1; break;
117         case z3::sat: {
118             z3::model m = s.get_model();
119             Node *p = new Node;
120             GiveVarValue(p, m);
121             GetPositive(p, positiveSet);
122             break;
123         }
124         case z3::unknown: return -1;break;
125     }
126
127     s.reset();
```

**图 4-5 利用约束求解生成 Positive(SP)**

从图中可以看出，我们使用 GiveVarValue(Node *p, model m)对于得到的 model 进行解析，这是因为在 model 中得到的变量取值的顺序并不唯一，我们必须通过读取该取值的对应变量名称之后，才可以为结构体对应变量赋值。同时我们得到的取值是 Z3 中 expr 类数据，如果是整型数据或 bool 型数据，则直接调用 expr.get_numeral_int()函数转为 int 值；但如果是 double 型数据，由于 expr 没有提供对应 API，所以我们需要先利用 expr.get_decimal_string()来获得 string 类型的数据再转化为 double 型。

约束求解结束之后，如果 *Positive(SP)*或 *Negative(SP)*的数量并未达到预设最大值，则进入随机生成部分。我们采用系统毫米时间初始化随机数生成器，使用 rand()函数生成随机数。对于 bool 型变量，我们将生成的随机数模 2；对于 int 型变量，我们模 201 后减去 100，将生成范围定义在[-100, 100]之间；对于 double 型变量，我们除以 RAND_MAX 然后乘以 200 再减去 100，同样将生成范围定义在[-100, 100]之间。

随机数生成之后，首先进行重复性检查，如果该样本点已经在 *Positive(SP)* 或 *Negative(SP)*中则忽略该点，重新生成新样本点。如果不存在重复，则首先调用 TestIfSatisfyPre()函数判断是否满足 *Pre*，如果满足则调用 GetPositive()函数完成后续操作；反之调用 GetNegative()函数。

当 *Positive(SP)*和 *Negative(SP)*的数量达到预设值时，我们将样本点按照

LIBSVM 要求的格式写入文件中，具体生成的初始样本文件部分截图如图 4-6 所示。

```
+1 1:0 2:1
+1 1:10 2:4
-1 1:60 2:-94
-1 1:6 2:-48
-1 1:31 2:-35
-1 1:-21 2:-61
-1 1:30 2:-36
-1 1:-41 2:-71
+1 1:-95 2:-73
```

**图 4-6 生成的初始样本部分截图**

## 4.3 基于选择取样自动迭代优化的候选循环不变式生成

初始样本生成完成，我们首先利用 LIBSVM 训练数据得到对应模型文件，然后将模型文件解析为分类超平面等式，判断符号后得到候选循环不变式。在选择取样过程中，利用约束求解器 Z3 对于循环不变式进行求解，得到边界样本值按照 4.2 节介绍的方式添加到初始样本集中，重复迭代，直到两次循环不变式相等，迭代结束。

### 4.3.1 根据初始样本生成循环不变式

如 3.3 节对于候选循环不变式生成的算法描述，我们利用首先 GenPolyData.cpp 和 GenPolyVar.cpp 将原数据和变量映射到目标维度，这里同样将变量进行映射，方便后续循环不变式输出。

以 Benchmark/01 为例，映射完成后，新数据保存在 01.polydata 文件中，新变量保存在 01.polyvar 中。对于 01.polydata 使用 LIBSVM 进行训练，具体命令如下所示：

参数 ”-t 0” 表示采用线性内核进行分类模型训练，最终得到训练模型及对应解释如图 4-7 所示。

```
svm_type c_svc
kernel_type linear
nr_class 2
total_sv 3
rho -1.645403002166782
label 1 -1
nr_sv 2 1
SV
0.0032734042554439743 1:10 2:4
0.0067552908957404321 1:-81 2:-93
-0.010028695151184415 1:-41 2:-71
```

**图 4-7 LIBSVM 训练模型结果及相关注释**

根据 2.5 节对于支持向量机分类算法的介绍，计算 $\omega$ 和 b 的值即可得到分类超平面。在 LIBSVM 生成的 model 文件中和 $\omega$、b 相关的参数为：

- *rho* ：*-b*
- *SVs* ：支持向量，格式为：

$$coefficient_1\ 1:a_{11}\ 2:a_{12}\ \ldots\ \mathrm{n}:a_{1n}$$
$$coefficient_2\ 1:a_{11}\ 2:a_{12}\ \ldots\ \mathrm{n}:a_{1n}$$
$$\ldots\quad\ \ldots\quad\ \ldots\quad\ \ldots\quad\ \ldots$$
$$coefficient_j\ 1:a_{j1}\ 2:a_{j2}\ \ldots\ \mathrm{n}:a_{jn}$$

根据定义，$b=-rho$，$\omega$ 的计算公式如下：

$$\omega_i=\sum_{k=1}^{j}coefficient_k*a_{ki}$$

得到相应参数之后，需要计算不等式符号：取一组初始样本，代入生成等式中计算，如果结果和相应的标签相符，则为≥，反之为≤，这样我们就得到了最终的候选循环不变式：

$$\omega^T x+b\geq 0 \text{ or } \omega^T x+b\leq 0$$

生成的循环不变式存储在 01.invariant 文件中，对应参数存储在 01.parameter 中，第一行为 b，其余行是对应 01.polyvar 中变量顺序的 $\omega$ 参数。

### 4.3.2 利用选择取样加速整体算法收敛速度

完成候选循环不变式生成后精心选择取样，选择取样相关算法和原理在 3.3 节中有详细说明，这里我们主要介绍实现细节：

首先生成 01_predict.cpp，采用约束求解工具 Z3 选取在循环不变式等式上的样本点，具体使用方法类似 4.2 中介绍，使用 solve 类进行求解，使用 solve.add() 函数添加约束。首先将循环不变式转化为等式添加到 solve 中，需要注意的是，在 Z3 中如果参数是小数，如 $-0.1*x+0.1*y=1.6$，则不能直接使用该等式作为约束条件，需要将参数用 context.real_val(“num”)的形式封装起来。同时虽然在数学意义上我们进行的是等式求解，但是在 Z3 的约束条件中，我们应该添加的是逻辑表达式，所以等号应该为程序中的”==”，故 $-0.1*x+0.1*y=1.6$ 在程序中最终应该改写为：

$$s.add(c.real\_val("-0.1")*x+c.real\_val("0.1")*y==c.real\_val(1.6);$$

然后按照算法流程，将前 n-1 个变量按照其各自的类型生成对应的随机数，并且利用 solve.add()将对应的取值作为约束条件添加进去，将第 n 个变量的类型设置为 real（Z3 系统中表示 double），然后求解。如果存在解，则解析 model 获得第 n 个变量的值，如果该变量为 bool 或 int 类型，则就近取整。最后将所有生成的样本点写入到 01.predict 文件中。

但是新生成的 01.predict 并不是最终的可以添加到初始样本集中的数据，还需要按照 4.2 中初始样本生成的方式判断是否满足 *Pre* 以及循环后是否满足 *Post*，并且循环过程中的节点都需要加入。因此我们使用类似 4.2 的方式生成 01_addBorder.cpp 文件，将相关条件解析到文件中，然后用 01_addBorder.cpp 文件读入 01.predict，分别对每一组样本点判断是否满足 Positive(SP)和 Negative(SP)的要求，并最终加入到 01.ds 中，重复迭代。

## 4.4 候选循环不变式验证

验证过程主要使用约束求解工具 Z3 和符号执行工具 KLEE，同样用脚本将相关限制条件“填空”到对应验证文件中，编译运行对应文件得到最终结果。

条件(4)和条件(6)的验证都使用的是 Z3 进行约束求解，分别用 01_verify1.cpp 和 01_verify3.cpp 及进行验证，所以他们放在一起详细说明。同样按照 4.3 节中对于 Z3 约束求解用法的说明，采用 solve 类求解，调用 solve.add()函数增加约束限制，以条件(4)为例，先添加 *Pre* 约束，再增加非循环不变式约束，然后判断是

否存在解。如果存在，则循环不变式不满足霍尔三元组要求，从 model 中解析出对应样本点，利用 01_addBorder.cpp 文件添加到 01.ds 文件中。如果不存在满足的点，则返回 0，说明循环不变式没有违反该条件。

验证条件(5)需要使用 KLEE 符号执行工具，使用 01_verify2.c 文件验证。按照 3.4 节中伪代码流程，首先使用 klee_make_symbolic()函数声明符号化执行的变量，然后使用 klee_assume()函数添加约束限制，在条件(5)中需要添加循环条件和循环不变式的限制。然后正常执行一轮循环，再用 klee_assume()添加非循环不变式的限制，如果运行时在这里报错，则说明不存在这样的点，条件(5)证实。如果没有报错，则说明存在符合条件的样本点，需要提取出该数据。KLEE 的运行结果存储在 klee-last/test000001.ktest 文件中，使用以下命令将结果输出为可读文件：

```
$ ktest-tool test000001.ktest
```

```
ktest file : 'klee-last/test000001.ktest'
args       : ['01_verify2.bc']
num objects: 2
object 0: name: 'x'
object 0: size: 4
object 0: data: b'\x00\x00\x00\x00'
object 0: hex : 0x00000000
object 0: int : 0
object 0: uint: 0
object 0: text: ....
object 1: name: 'y'
object 1: size: 4
object 1: data: b'\x00\x00\x00\x01'
object 1: hex : 0x00000001
object 1: int : 16777216
object 1: uint: 16777216
object 1: text: ....
```

**图 4-8 KLEE 结果文件部分截图**

KLEE 结果文件的形式如图 4-8 所示，可以看出我们需要的是 int 一栏的数据，因此在脚本中使用 grep 命令将所有含有”int :”的行筛选出来然后将数据分离，因为在结果中排序的顺序是我们在 01_verify2.c 中符号化变量的顺序，所以与给出的变量顺序一致，直接赋值即可。得到边界样本点之后执行 01_addBorder 添加到 01.ds 中重复整体算法迭代。

## 4.5 人机交互

人机交互的实现逻辑并不复杂，首先需要将霍尔三元组按照用户易懂的方式输出，然后输出当前循环不变式，给用户充分的信息以供发现边界样本，具体界面如图 4-9 所示。

```
xx mh% ./run.sh Benchmark/01
Generating......

            [User Input Border Node]

-------------------The hoare is----------------------
[variables]: x y
[precondition]: x<y
[The loop]
while((x<y)) {
     if(x<0) x=x+7; else x=x+10; if(y<0) y=y-10;else y=y+3;
}
[postcondition]: ((x>=y) && (x<=(y+16)))
-----------------The invariant is--------------------
-0.04 * x + 0.04 * y + 1.22 >= 0
-----------------------------------------------------
Do you want to mannually add some border node?
[input "Y" to add, input "N" or other to ignore]
```

**图 4-9 MIG 工具人机交互界面**

用户输入结束之后首先进行输入判断，判断输入是否符合要求，是否存在非法符号等等。然后运行 01_addBorder 判断新生成的样本点是否属于 Positive(SP)和 Negative(SP)，并且检查是否在 01.ds 中是否已经存在该样本点，如果不存在，则将样本点添加标签，补充到 01.ds 文件当中，加快整体算法收敛速度。

## 4.6 本章总结

本章详细介绍了 MIG 工具的具体实现以及技术细节，重点介绍了如何使用脚本语言与 C++语言结合的方式进行相关条件的设置，以及 Z3 和 KLEE 具体的使用方式，结果如何处理，LIBSVM 生成的分类模型如何计算得到超平面等等。并且以图 2.n(a)测试程序为例，说明了每一步执行后系统的期望输出，使读者对于 MIG 工具有了更加清晰的认知。

# 第五章 基于机器学习和选择取样的循环不变式自动生成工具实验与评估

第五章我们对 MIG 工具设置两组对照试验，比较 MIG 工具自身是否采用选择取样和人机交互的迭代次数、运行效率和求解精确度；同时比较 MIG 工具与最新的 Interproc, CPAChecker, BLAST 和 InvGen 循环不变式自动生成工具的性能和效率，分析 MIG 工具的表现。

## 5.1 运行说明

MIG 工具是 shell 脚本语言和 c++语言编写的，面向 c 和 c++程序的循环不变式自动生成工具。用户只需按照规定格式输入霍尔三元组便可获得满足该霍尔三元组的循环不变式。同时在实现过程中采用 Config 文件对用户配置进行管理，用户可以在 config.json 文件中进行相关参数的配置。

### 5.1.1 实验运行环境

**系统版本**：Linux debian 4.9.0 (2018-05-07) i686 GNU/Linux

**处理器**：单核 i7 处理器

**内存**：4G 内存

**软件环境**：LIBSVM (3.24)，KLEE (2.0)，Z3 (4.8.7)，同时需要将 LIBSVM 安装到项目根目录文件夹下或者添加环境变量。

### 5.1.2 配置参数

相关参数在 config.json 文件中进行设置，下面介绍参数对应含义和默认值：

- z3Build：Z3 Build 文件夹绝对路径。编译应用 Z3 相关 API 的 CPP 文件时 load Z3 Build 作为链接库。
- kleeInclude: KLEE Include 文件夹绝对路径。编译应用 KLEE 相关 API 的 CPP 文件时需要引用 KLEE 相关头文件。
- degree（默认值为 1）：设置预期循环不变式变量最高次数，如果需要线性循

环不变式设置为 1，如果需要非线性循环不变式或布尔逻辑交并集形式需要设置为对应 degree。

- selective（默认值为 1）：是否开启选择取样，0 为关闭，其余值为开启。
- interactive（默认值为 0）：是否开启人机交互，1 为开启，其余值为关闭。
- outputDetail（默认值为 0）：是否输出生成细节，1 为输出，其余值为直接输出结果。
- initialRandomNum（默认值为 0）：初始样本生成时，对于 *Positive(SP)*和 *Negative(SP)*数量的预设最大值，具体解释详见 3.2 节。
- selectiveSampleNum（默认值为 5）：选择取样时，在循环不变式等式上取点的数量。
- maxIterationTimes（默认值为 128）：整体算法和选择取样迭代的最大次数，如果超出次数仍未收敛，工具输出超时错误。

### 5.1.3 测试文件格式

参考测试用例在 Benchmark/ 文件夹下，可以根据样例编写个人测试程序，这里进行简单介绍测试程序组成和格式。

**必选项：**

- 待预测变量集：以 names@开头，以空格隔开，结尾避免多余空格。
- 待预测变量类型集：以 types@开头，以空格隔开，与变量集一一对应。
- Precondition：以 precondition@开头，跟随 bool 类型表达式。
- Loopcondition：以 loopcondition@开头，跟随循环条件。
- Loop：以 loop@开头，跟随完整程序片段，说明循环主体程序部分，需要去除换行符。
- Postcondition：以 postcondition@开头，跟随 bool 类型表达式。

**可选项：**

- 循环前程序片段：以 loopbeofer@开头，跟随完整程序片段，用以添加循环前准备工作以及不在待预测变量集中的变量定义与初始化工作。

以图 2.n(a)测试用例为例，在 MIG 工具中该用例改写为如图 5-1 所示，该文

件命名为 Benchmark/01。

```
names@x y
types@int int
precondition@x<y
loopcondition@(x<y)
loop@ if(x<0) x=x+7; else x=x+10; if(y<0) y=y-10;else y=y+3
postcondition@((x>=y) && (x<=(y+16)))
degree@1
selective@1
interactive@0
```

**图 5-1 样本测试用例在 MIG 工具中的形式**

### 5.1.4 运行说明

进入 MIG 工具根目录，运行 run.sh，输入参数为测试用例路径，以图 2.n(a)测试用例为例，具体命令如下：

```
$ ./run.sh Benchmark/01
```

- 生成结果如图 5-2 所示（未开启人机交互、不输出详细信息）：

```
 ./run.sh Benchmark/01
Generating......
-----------------------------------------------------
The invariant is : -0.10 * x + 0.10 * y + 1.60 >= 0
-----------------------------------------------------
Total run time : 5s  | Total iteration times: 1
```

**图 5-2 未开启人机交互和信息输出运行结果截图**

在生成过程中，算法总体每经过一轮迭代 Generating 后面会增加’…’，可以通过这个标识来判断系统是正在生成还是陷入死循环。生成的循环不变式高亮输出，同时输出总运行时间以及总迭代次数。

- 若输出详细信息，则生成结果如图 5-3 所示（未开启人机交互）：

```
xx mh% ./run.sh Benchmark/01
Converting the given config file to a cplusplus file...[Done]
Compile the cplusplus file and get the initial data...[Done]
#######################################################
Generating Loop Invariant...[Times 1]
------------------svm-learner 1-------------------
Reflacting data to dimention degree...[Done]
Using libsvm-3.24 to train the model...[Done]
Calculating Hyperplane of the model...[Done]
The hyperplane is : -0.01 * x + 0.01 * y + 1.03 >= 0
Generating predict cpp file and compile...[Done]
Predict border node according to the model...
[Done]
Checking convergence...[False]
Adding new border node into data file... [-1 1:68 2:-35]  [-1 1
69 2:-34]  [-1 1:-32 2:-135]  [-1 1:98 2:-5]  [-1 1:12 2:-91]
[Done]
```

• • • • • •

```
------------------svm-learner 5-------------------
Reflacting data to dimention degree...[Done]
Using libsvm-3.24 to train the model...[Done]
Calculating Hyperplane of the model...[Done]
The hyperplane is : -0.17 * x + 0.17 * y + 2.00 >= 0
Generating predict cpp file and compile...[Done]
Predict border node according to the model...[Done]
Checking convergence...[False]
Adding new border node into data file...
[Done]

------------------svm-learner 6-------------------
Reflacting data to dimention degree...[Done]
Using libsvm-3.24 to train the model...[Done]
Calculating Hyperplane of the model...[Done]
The hyperplane is : -0.17 * x + 0.17 * y + 2.00 >= 0
Generating predict cpp file and compile...[Done]
Predict border node according to the model...[Done]
Checking convergence...[True]
[Done]
Verifying Invariant...
[Done]
The generated Invariant satisfies hoare triple[Process Finished
-------------------------------------------------
The invariant is : -0.17 * x + 0.17 * y + 2.00 >= 0
-------------------------------------------------
Total run time : 8s  | Total iteration times: 1
```

图 5-3 开启信息输出运行结果截图

如果选择输出全部信息，则会将每轮迭代每一步过程输出，并且输出每一轮生成的候选循环不变式，选择取样新生成的样本点，以及如果未通过验证生成的边界样本点等，方便用户进行更具体的数据分析或错误定位。同时可以发现我们两次举例生成的结果并不相同，这是因为初始样本生成、选择取样和验证过程中都有运用随机数生成的过程，所以会造成结果的不确定性。但是循环不变式本身也并不是唯一解，可以看出两次举例生成的循环不变式都是满足霍尔三元组的要求，所以都是正确的符合用户需求的结果。

- 若开启人机交互，则会在每轮迭代之后出现相应的输入指引，具体结果如图 5-4 所示：

```
xx mh% ./run.sh Benchmark/01
Generating......

            [User Input Border Node]

-------------------The hoare is----------------------
[variables]: x y
[precondition]: x<y
[The loop]
while((x<y)) {
     if(x<0) x=x+7; else x=x+10; if(y<0) y=y-10;else y=y+3;
}
[postcondition]: ((x>=y) && (x<=(y+16)))
-----------------The invariant is--------------------
-0.04 * x + 0.04 * y + 1.22 >= 0
-----------------------------------------------------
Do you want to mannually add some border node?
[input "Y" to add, input "N" or other to ignore]
```

**图 5-4 开启人机交互运行截图**

为了方便用户进行输入和分析，我们将霍尔三元组和候选循环不变式的相关信息输出到控制台。同时用户可以输入多组数据也可以选择是否输入数据，并不是开启人机交互则每一轮必须输入。因为如 3.5 节中介绍，如果随意添加样本点反而会增加收敛时间，所以用户需要慎重添加。添加过程按照命令行指引输入对应变量取值即可。

- 若用户需要生成非线性循环不变式，则将 degree 设置为相应数值，MIG 工具

会在生成对应次数循环不变式之后，尝试因式分解，如果可以分解成功，则同时输出原结果和分解后的布尔表达式。我们以图 2-1(b)程序为例，图 2-1(b)在 MIG 工具中为 Benchmark/05 文件，具体结果如图 5-5 所示：

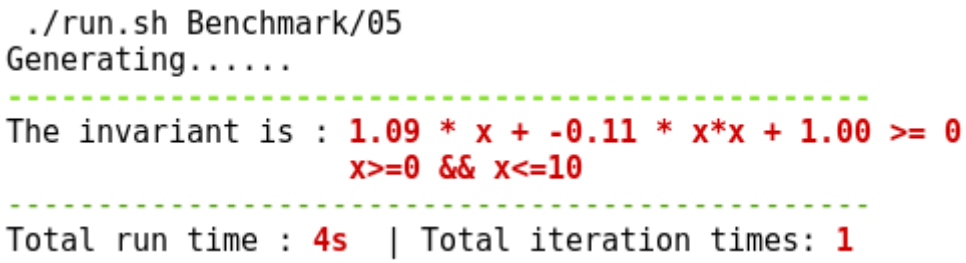



**图 5-5 degree 设置为 2 时的运行结果截图**

## 5.2 实验设置

我们针对 MIG 工具提供了 20 个样例测试程序，包含线性循环不变式预测和二次多项式循环不变式预测等多种类型测试。同时在这 20 个测试样例基础上，设置了两组对照试验，分别比较 MIG 选择取样、人机交互模块性能以及比较 MIG 与最新的循环不变式自动生成工具进行性能和效率。

### 5.2.1 程序测试集

MIG 工具的主要算法思路来源于文献[]，因此我们采用文献[]本身提供的测试集进行相关实验，因此 MIG 提供 20 个样例测试程序，包含线性循环不变式预测和二次多项式循环不变式预测，由于电脑性能限制，多变量高次数的循环不变式生成会触发超时判断，所以预测变量包含 bool, int 和 double 型的 1 – 3 个变量。具体设置如图 5-6 所示：

| benchmark | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 变量数 | 2 | 1 | 2 | 2 | 1 | 1 | 2 | 1 | 3 | 3 |
| 变量类型 | int | int | int | int | int | double | int | int | int | int |
| 最高次数 | 1 | 1 | 1 | 1 | 2 | 1 | 1 | 1 | 1 | 1 |
| benchmark | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 变量数 | 3 | 2 | 3 | 2 | 3 | 2 | 1 | 1 | 2 | 3 |
| 变量类型 | int | int | int | int | int | int | int | double | int | int |
| 最高次数 | 1 | 1 | 1 | 2 | 1 | 1 | 1 | 1 | 1 | 1 |

**图 5-6 MIG 工具测试集具体设置**

### 5.2.2 实验配置

MIG 设置两组对照实验，第一组是对比自身不采用选择取样，采用选择取样以及采用选择取样和人机交互三种方式生成循环不变式，所生成的样本数量、迭代次数和 CPU 耗费时间。为了尽量减少实验的随机性，将初始样本中的随机数样本生成数量减少为 0 或 1，即如果约束求解失败则 *Positive(SP)*和 *Negative(SP)* 各随机生成一个初始样本，如果约束求解成功则采用约束求解得到的值。因为在人机交互过程中，人力输入的时间往往显著长于 CPU 运行时间，所以我们记录时间为 CPU 消耗时间。并且因为随机取样带来的随机性会影响最终运行时间，所以记录三次运行时间的平均值。

第二组对照实验为 MIG 和现在流行的其他循环不变式生成系统 Interproc, CPACheck, BLAST 和 InvGen 的正确性进行对比，将其他系统生成的循环不变式结果代入验证程序验证是否满足霍尔三元组。Interproc 是基于抽象解释生成循环不变式的程序验证工具，因此在实验中我们采用它表达力最高的抽象域（abstract domain），即 convex polyhedral + linear congurences。CPAChecker 我们采用在 SV-COMP 2017[参考文献]中使用的版本，并且采用 SV-COMP 中相同的配置和方法。BLAST 是基于边界样本导向的抽象完善方法，InvGen 是用动态静态分析相结合的方式生成线性循环不变式。

## 5.3 实验结果

### 5.3.1 实验一：是否采用选择取样和人机交互运行对比

| | without Selective Sampling | | | with Selective Sampling | | | with Selective Sampling and Man-Machine Interaction | | |
|---|---|---|---|---|---|---|---|---|---|
| | #sample | #iter | time(s) | #sample | #iter | times(s) | #sample | #iter | times(s) |
| benchmark01 | **16** | 6 | 20.19 | 23 | **1** | **6.63** | 23 | **1** | **6.63** |
| benchmark02 | **7** | **1** | **3.94** | **7** | **1** | 4.77 | **7** | **1** | 4.77 |
| benchmark03 | **5** | 3 | **9.48** | 2185 | **1** | 10.56 | 2185 | **1** | 10.56 |
| benchmark04 | **4** | 3 | 10.65 | 41 | **1** | **8.03** | 41 | **1** | **8.03** |
| benchmark05 | **12** | **1** | **3.52** | **12** | **1** | 4.65 | **12** | **1** | 4.65 |
| benchmark06 | 14 | 5 | 15.19 | **13** | **1** | **5.12** | **13** | **1** | **5.12** |
| benchmark07 | **6** | 6 | 18.48 | 33 | **1** | **7.62** | 33 | **1** | **7.62** |
| benchmark08 | **11** | 3 | 9.46 | 97 | **1** | **7.51** | 97 | **1** | **7.51** |
| benchmark09 | **33** | 2 | 4.97 | **33** | **1** | **4.64** | **33** | **1** | **4.64** |
| benchmark10 | **7** | 5 | 15.50 | 5078 | 3 | 19.62 | 4609 | **2** | **17.34** |
| benchmark11 | to | to | to | 1035 | **2** | 9.97 | **978** | **2** | **9.78** |
| benchmark12 | 116 | 10 | 29.89 | **45** | **1** | **10.50** | **45** | **1** | **10.50** |
| benchmark13 | to | to | to | **67** | **1** | **11.60** | **67** | **1** | **11.60** |
| benchmark14 | **1598** | 3 | 9.97 | 2001 | **1** | **5.39** | 2001 | **1** | **5.39** |
| benchmark15 | to | to | to | 681 | **2** | 15.17 | **540** | **2** | **13.49** |
| benchmark16 | **3** | 2 | **6.52** | 29 | **1** | 7.56 | 32 | **1** | 7.56 |
| benchmark17 | **14** | 2 | 6.51 | 105 | **1** | **5.08** | 105 | **1** | **5.08** |
| benchmark18 | **15** | 3 | 9.44 | 18 | **1** | **7.02** | 18 | **1** | **7.02** |
| benchmark19 | to | to | to | **73** | **1** | **7.56** | **73** | **1** | **7.56** |
| benchmark20 | to | to | to | 1812 | 3 | 14.64 | **1400** | **2** | **10.50** |

**图 5-7 是否采用选择取样和人机交互运行对比结果**

在这一实验中，我们统计了不采用随机取样、采用随机取样和采用随机取样加人机交互结合三种生成方式下，生成样本数量、迭代次数以及运行时间数据，实验结果的汇总如图 5-7 所示。

从表中数据分析可知，采用随机取样方式可以显著减少迭代次数，并且可以解决非随机采样下 TIME_OUT 的测试用例，但是通常需要生成更多的样本数据；采用随机取样加人机交互的方式在 10、11、15、20 四个用例中，减少了程序运行时间或迭代次数，但是这需要测试人员对于样例及对应霍尔三元组非常熟悉，才能添加合理的加速收敛的边界点，因为 SVM 分类器生成的超平面需要使得所有点到分类器的距离最大化，所以如果随意添加样本点，可能会增加运行时间。

同时注意到 02、03、05、16 四个测试用例中，不采用随机取样方式所需时间更少，这是因为虽然采用随机取样的方式减少了总迭代次数，但是在每一轮迭代中需要多次取样使得生成循环不变式收敛，而这四个测试用例的 precondition 十分精确的给出了关键分类样本点，导致直接根据初始样本点生成循环不变式结束迭代的不采用随机取样方式时间更短。但是这样的结果并不具备通用性，因为对于给定霍尔三元组的精确度有很高的要求，当处理测试用例相对复杂的 11、13、15、19、20 测试样例时，不采用随机取样的方式会导致超时而无法得到最终结果。

### 5.3.2 实验二：MIG 和其他循环不变式生成工具的效果对比

MIG 与 Interproc, CPAChecker, BLAST 和 InvGen 的结果对比如图 4-6 所示，✓表示生成的循环不变式满足霍尔三元组，而×表示生成的结果不能验证霍尔三元组。Interproc, BLAST 和 InvGen 的效率都较高，基本都在一秒之内完成循环不变式的生成，因为我们没有记录所需时间，同时对于 CPAChecker 我们记录它成功生成循环不变式并通过验证的样例所需的时间。

| | MIG | Interproc | CPAChecker | BLAST | InvGen |
|---|---|---|---|---|---|
| | #times(s) | #result | #times(s) | #result | #result |
| benchmark01 | 6.63 | ✓ | 3.73 | ✓ | ✓ |
| benchmark02 | 3.94 | ✓ | 3.19 | ✓ | ✓ |
| benchmark03 | 9.48 | ✓ | 3.55 | ✓ | ✓ |
| benchmark04 | 8.03 | ✓ | 3.40 | ✓ | ✓ |
| benchmark05 | 3.52 | ✓ | × | × | × |
| benchmark06 | 5.12 | × | 3.16 | ✓ | ✓ |
| benchmark07 | 7.62 | ✓ | 3.71 | ✓ | ✓ |
| benchmark08 | 7.51 | × | × | × | ✓ |
| benchmark09 | 4.64 | × | 3.18 | ✓ | ✓ |
| benchmark10 | 17.34 | ✓ | 4.18 | ✓ | ✓ |
| benchmark11 | 9.78 | × | × | to | ✓ |
| benchmark12 | 10.50 | ✓ | 3.92 | ✓ | ✓ |
| benchmark13 | 11.60 | ✓ | 4.04 | ✓ | ✓ |
| benchmark14 | 5.39 | × | × | × | × |
| benchmark15 | 13.49 | × | × | ✓ | ✓ |
| benchmark16 | 6.52 | × | 3.71 | × | × |
| benchmark17 | 5.08 | ✓ | 3.05 | ✓ | ✓ |
| benchmark18 | 7.02 | ✓ | 3.28 | ✓ | ✓ |
| benchmark19 | 7.56 | ✓ | 3.39 | ✓ | ✓ |
| benchmark20 | 10.50 | ✓ | 3.75 | ✓ | ✓ |

**图 5-8 MIG 和其他循环不变式生成工具的效果对比结果**

通过图 5-8 中数据得知，MIG 可以正确生成全部 20 个测试程序的循环不变式，包括线性和非线性情况，正确率为 100%，相比之下，Interproc 正确生成 13 个测试程序循环不变式，正确率为 65%；CPAChecker 正确生成 15 个测试样例，正确率为 75%；BLAST 正确生成 15 个测试用例，1 个测试用例超时，正确率为 75%；InvGen 正确生成 17 个测试用例，正确率为 85%。当然因为测试用例为 MIG 系统提供，所以可能对其他四个系统并不特别公平，但是仍然可以反映出 MIG 系统具有较好的性能。虽然 MIG 所耗时间长于其他系统，但其他系统并没有霍尔三元组和循环不变式的验证过程；而且在 20 个测试程序中，MIG 的都可以在 18 秒内完成生成和验证过程，总时耗在接受范围之内，所以综合比较，MIG 与其他系统相比在性能上有较好的表现。

## 5.4 本章总结

本章针对实现的 MIG 系统设计了两组对比实验，实验数据表明通过选择取样和人机交互可以提升循环不变式生成系统的性能，减少迭代次数和运行时间，达到了预期研究目标。

# 第六章 总结与展望

## 6.1 本文主要贡献与创新

本文主要实现了基于机器学习和选择取样的循环不变式自动生成系统，下面介绍主要贡献及创新点：

1. MIG 系统区别于传统的约束求解方式，应用支持向量机对于初始样本进行分类学析，所以可以支持多种简单数据类型、复杂程序以及非线性循环不变式的生成，能够扩展其应用范围。

2. 通过 MIG 进行两组对比实验表明，采用选择取样以及人机交互可以提升系统性能和效率，同时相对于 Interproc 等其他现有工具有较好的表现。

## 6.2 研究展望

作为基于机器学习的循环不变式自动生成系统，MIG 同时也存在较大的提升空间：

1. MIG 不能支持字符串、结构体等复杂数据类型的循环不变式生成，所以在实际程序应用中会受到一定限制。

2. 因为在初始样本的生成过程和选择取样的过程中都存在随机数的生成过程，而样本的好坏会直接影响到迭代次数与结果，所以运行的性能与效率并不能得到完全的保证，后续可以优化随机样本生成过程减少随机性带来的影响。

# 参考文献

# 致谢