Melton设计手册

## 简介

## 内存泄漏是常见的一种程序错误，通常出现在需要手工做内存管理的程序中，如 C 或 C++程序。内存泄漏是指程序中存在动态分配的内存，没有被任何指针继续引用并且没有得到释放。持续的内存泄漏会消耗系统的内存，降低系统的运行性能。最终，当所有的可用内存被消耗光时，整个或部分系统将无法正常工作。

## 自动化的检测程序中可能存在的内存泄漏，主要的方法分成两类：动态分析和静态分析。

## 动态分析是通过执行插装过后的程序来检测已存在的内存泄漏。为了能够有效的检测到内存泄漏，目标程序必须用足够的测试输入（如根据软件测试中的代码覆盖准则来保证）来运行。其优点为：精确性强，其缺点为：性能、覆盖率、应用等方面的局限性。

## 静态分析是通过静态的扫描程序源码，能够在程序运行前对程序进行全面的分析。其优点为：尽早检测出错误、覆盖率高、适用性强，代价低，其缺点为：分析代价、精度不够。

## 本工具基于静态分析方法，使用了一种基于内存状态转移图（Memory State Transition Graph, MSTG）的过程间内存泄漏检测算法。一个内存状态转移图定义了每个内存对象的一系列状态，并且在分析的过程中跟踪这些状态之间的转换，用于判断是否能到达内存泄漏的状态。该算法采用了符号执行方法，在分析完一个函数时，每个函数都会被抽象成一个内存状态转移图，如果转移图中存在内存泄漏的节点，则能够判断出该节点相关的代码存在内存泄漏。为了支持过程间分析，本算法定义了针对内存泄漏的函数摘要形式，在每个函数分析完后根据产生的内存状态转移图生成相应的函数摘要，最后将函数摘要应用于函数的被调用点。

## 为了获得精确的分析结果，本工具的算法是路径敏感的，在有效的消除误报的同时能够为生成的内存泄漏报告提供丰富的诊断信息，如通过哪些路径片断能够导致报告中的内存泄漏的发生。与采用符号分析的方法相比，本工具的方法提供了更细粒度的对函数行为的捕捉，能处理全局的指针别名和进行过程间分析。

### 工具特点

## 本工具主要有如下的特点：

## 精确的内存建模：该模型将所有与内存泄漏相关的内存对象建模成两种类型：堆对象（ Heap Objects）和外部对象（ External Objects）。其中堆对象指的是分配在堆空间里的内存对象，用来检测内存泄漏是否存在；外部对象指的通函数参数或全局指针变量引入的内存对象，在当前的函数体里无法知道该内存对象的来源。外部对象能够用来提供过程间分析的别名。如无特别说明，本章中的“内存对象”指的是堆对象或者外部对象。

## 新的分析模型：如前所述，本章提出的内存状态转移图能够描述内存的状态随着程序的分析进行转移，可以精确的检测具有泄漏状态的内存对象。

## 精确的函数摘要：由内存状态转移图生成的函数摘要包括内存行为（Memory Actions）和路径约束，在应用函数摘要的时候，内存行为用来捕捉函数调用点产生的副作用，路径约束则用来消除该调用点不可行的内存行为。

## 易于诊断的错误报告：错误报告的确认是静态分析完成后需要进行人工确认的一个阶段，通常也是一个较为耗时的阶段，特别对于内存泄漏这种错误，发生时并没有很明显的现象，而是缓慢的增加程序的内存，因此非常难以定位。而大部分的工具的错误报告中只是将内存分配的程序点指出来，本章方法生成的错误报告在指出内存分配点的同时能够显示出分配函数的调用链和导致该内存泄漏的路径片断。

## 算法框架

算法框架 LeakDetect如图 1所示，算法介绍如下：

1. 从整个程序 C中建立函数调用图，根据调用图自底向上分析每个函数
2. 使用分析函数 Visit 来分析调用图中的函数
   1. 生成一个对应的内存状态转移图（GenMSTG ）
   2. 在上述生成过程，对泄漏状态进行检测
   3. 从内存状态转移图中生成函数摘要（GenSum ）并将该函数置为已访问

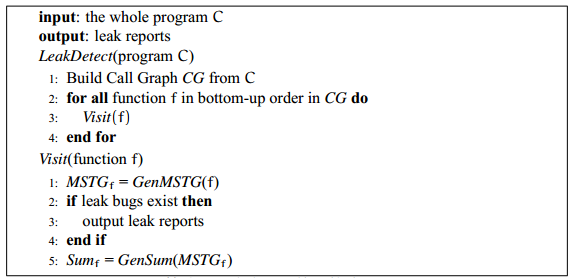


图 1 算法框架

## 过程内分析

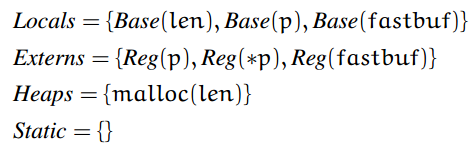
过程内分析指在一个函数内部的分析，不考虑函数间的相互调用的分析方式。本节主要描述了过程内分析的算法，包括内存建模，内存状态转移图的定义以及利用内存状态转移图的分析过程。

### 内存建模

**内存区域** 为了能够对内存类型进行更好的区分，将内存区域分成如下四种类型：

1. Local：表示分配在栈里的内存区域。
2. *Static*：表示分配在静态空间的变量，如全局变量和静态变量。
3. Heaps：表示分配在堆空间的内存区域。
4. Externs：表示在当前函数的范围内未知的内存对象，如指针类型的函数参数或全局变量所指向的内存区域。

为了更清晰的描述内存区域，使用 Reg(p) 表示指针p 指向的内存对象，使用Base(p) 来表示变量p 自身对应的内存对象。图[3.8](#_bookmark52)是从图[3.7](#_bookmark47)抽取出来的bufSelect 函数，其相应的内存区域分类如下：



需要注意的是，这里的内存建模是限制当前函数范围内的，因此上述的四个集合是互不相交的。例如，给定一个全局变量char \*g，Base(g) 属于 Statics 集合，而Reg(g)(等价于 Base(∗g)) 属于 Externs 集合（假定g 指向的内存对象未知）。

**内存行为抽象** 根据上述的内存区域分类，定义堆对象和外部对象的行为为如下六种类型：

1. ToLocal：表示将内存对象赋给属于 Locals 集合的指针。

void f(char \*p){ char \*q=p; };

1. ToExtern 将内存对象赋给属于 Externs 或 Statics 集合的指针。

\*p = fastbuf;

1. ToReturn：表示返回了一个堆对象或外部对象。

void\* f(void \*p) { return p; };

1. ToAlloc：将内存对象赋给堆内存上的指针。

list \*l = (list \*) malloc(...); l->next = p;

1. ToFree ：表示释放一个内存对象。
2. ToUnknown：表示将内存对象分配给未知的指针。（用于消除误报）

**状态抽象** 通过已定义的内存行为，能够引发堆对象和外部对象内存状态的改变。分别为堆对象和外部对象定义内存状态。

为每个堆对象，定义如下 5 种内存状态：

1. Allocated 状态：堆对象被分配并且只有本地的或临时的指针变量指向它。
2. Escaped 状态：堆对象通过返回值返回、赋给参数指针或全局变量指针（包括 ToReturn 和 ToExtern 行为）
3. Freed 状态：堆对象已经被释放（包括 ToFree 行为）。
4. Relinquished 状态：堆对象被赋给一些无法处理的复杂的 C 表达式（包括ToUnknown 行为）。
5. Leaked 状态：堆对象在函数的最后不是Escaped、Freed 或 Relinquished 状态。

为每个外部对象，定义如下 4 种内存状态：

1. Accessed 状态：初始时的外部对象，并且在当前函数的范围内只能被本地或临时指针变量指向。
2. Escaped：堆对象通过返回值返回、赋给参数指针或全局变量指针。（包括 ToReturn 和 ToExtern 行为）
3. Freed：堆对象已经被释放。（包括 ToFree 行为）
4. Relinquished：堆对象被赋给一些无法处理的复杂的 C 表达式。（包括ToUnknown 行为）。

### 内存状态转移图

为了能够表示多个内存对象的状态转移，提出了一个表示能力很强的模型：内存状态转移图。

**定义** 内存状态转移图（Memory State Transition Graph, MSTG），具体定义为一个四元组 ⟨S, M, L, ->⟩，其中：

* S：内存状态的集合，包括上述堆对象和外部对象的状态。其中 Allocated和 Accessed 是两个初始的状态，其他的状态均可作为终止状态。
* M：内存对象的集合，可以包括多个堆对象和外部对象。
* L：是一系列标签（Label）的集合。一个标签可以是在符号执行过程中收集的符号约束或者一个内存行为。
* —>⊂ 2M×S × L × 2M×S：带标签的多个状态对象的状态的转移。

图2给出了内存状态转移图，其中每个节点包含一个内存对象的集合，每个内存对象都有一个状态，即 2M×S；图中的每条边带有一个标签，表示两个节点间的状态转移的条件。一个状态转移可能改变一个节点内多个内存对象的状态，每个内存对象的状态转移需遵循基本内存转移内的转移规则。

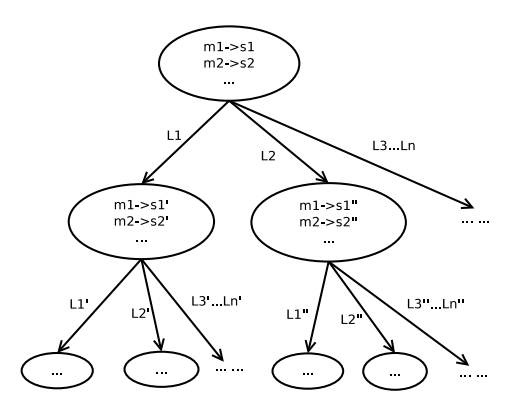


图 2 内存状态转移图模型

**构建** 内存状态转移图的构建是由路径敏感的符号执行来完成的。图3给出了内存状态转移图的生成算法。

1. 首先对函数 f 建立了一个控制流图，然后采用宽度优先的策略对控制流图进行遍历。
2. 在分析每个基本块时，算法中主要关注赋值语句、释放语句、返回语句、函数调用语句和分支语句。
3. 若遇到前三种类型的语句，即赋值语句、释放语句、返回语句，则可能会导致状态的转移。因此第一步是先获取到可能会发生状态转移内存对象。例如给定一个赋值语句q = p，其中p 和q 都是指针变量，其目标内存对象是 Reg(p)，其行为取决于 Base(q) 所属的内存类型。 UpdateState 则从当前的节点 N 根据相应的内存行为，生成一个新的节点 NewN，可记为 N − Act ! NewN。这其间如果检测到内存泄漏的状态，那么会将该节点传递给错误报告生成器，以生成相应的内存泄漏报告。
4. 若遇到函数调用语句，则会将相应的函数摘要应用到函数的调用点。具体的函数摘要的应用算法 ApplySum 会在下面的章节给出。
5. 若遇到分支语句，则会在先判断真分支上的分支条件 PC 的可行性。如果可行的话就生成一个新的节点 LN，然后采用 BFS 的方式进左分支的基本块 LB 进行分析。之后以类似的方式对假分支进行处理。

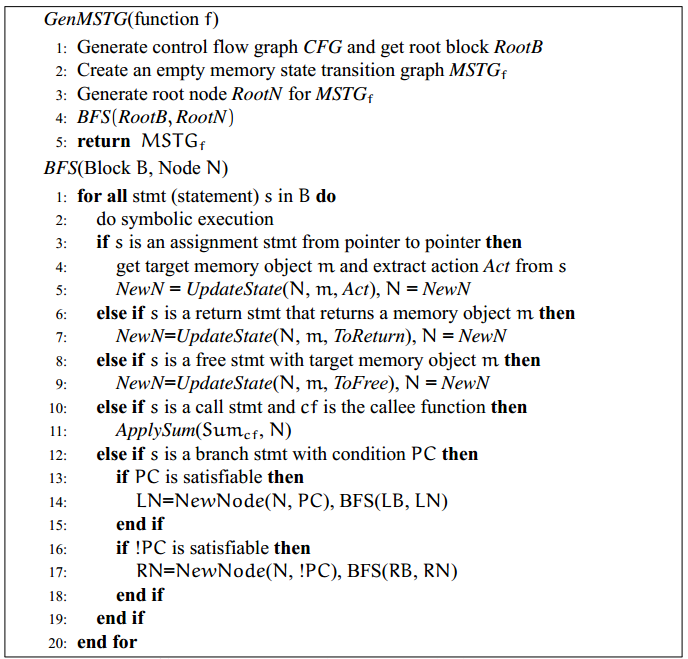


图 3 MSTG生成算法框架

**MSTG生成示例** 图4给出了一个简单的示例函数foo，用于演示 MSTG 的生成。函数foo 的控制流图如图5所示。在控制流图中，我们采用 Nop 基本块来将含有 3 个或以上分支的if-else 分支拆成只有两个分支的语句。

函数首先分配一个堆对象给指针p，并在三个不同的分支里对p 做了不同的操作：

1. 第一个分支将p 作为返回值返回，此时p 指向的内存对象为 Escaped状态；
2. 第二个分支将p 赋给一个全局变量g，此时p 指向的内存对象为 Escaped状态；
3. 第三个分支调用free 函数释放p 指向的内存对象，该内存对象的状态变为 Freed。

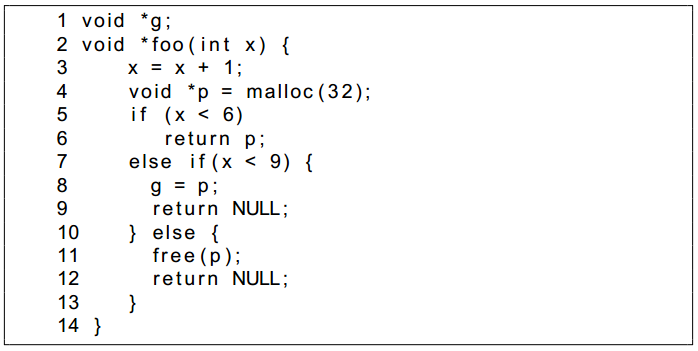


图 4 待分析代码

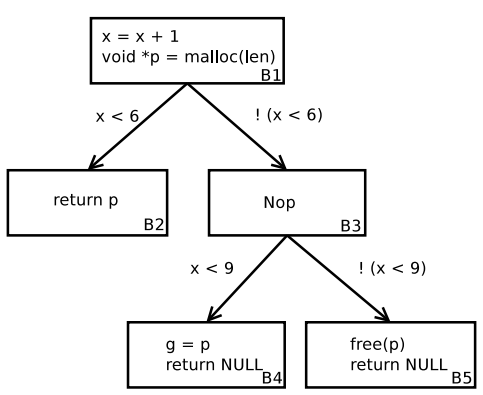


图 5 函数foo的控制流图（CFG）

整个内存转移图的生成实际上是一个过程内的分析。图5给出了相应的内存状态转移图。

分析过程：

1. 符号替换：为了生成内存状态转移图，算法首先对利用符号执行对程序表达进行符号替换。在函数开始时，对函数的每个参数生成一个新的符号变量，此处假设生成一个符号变量 $x 给参数x。在执行完第 3 行语句后，对变量x 的值做替换，即为 $x + 1。此时在控制流图中的第一个分支，会生成相应的路径约束 ($x + 1) < 6 和 !(($x + 1) < 6)，并把这些约束标记在内存状态转移图的边 S1 -> S2 和 S1 -> S4 上。
2. 对象状态分析：在节点 S1 上， Reg(g) 被初始化为 Accessed 状态，其中 HeapObj1 表示由代码示例中第 4 行的函数调用分配的堆对象。在边 S2 -> S3 上的内存行为 HeapObj1 -> return 会导致 HeapObj1 的状态转移为 Escaped 状态。节点 S6 中，由于没有指针指向 Reg(g) 了，该对象被删除。
3. 叶节点泄露检查：假设上述代码中没有第 11 行代码，那么在第 3 个分支处会发生内存泄漏。本算法会在每条路径的最后对所有的堆对象做一个状态检测，如果存在状态为 Allocated 的堆对象，那么则视为存在内存泄漏。

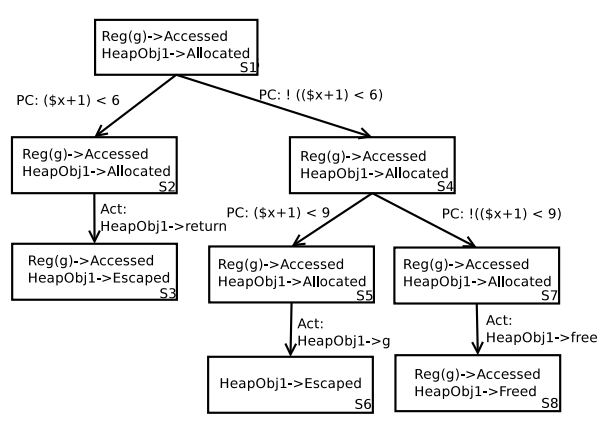


图 6 函数foo的内存状态转移图（MSTG）

## 过程间分析

函数对应的内存状态转移图（MSTG）生成完成，标志着该函数的过程内分析完成了。为了避免对相同的函数进行多次的分析，从每个函数的内存状态转移图生成函数摘要。对单个函数生成函数摘要之后，在具体的控制流图（CFG）分析中再应用函数摘要。

本节主要描述了基于函数摘要的过程间内存泄漏检测，包括：函数摘要定义、函数摘要生成、如何在函数的调用点应用函数接要。

### 函数摘要定义

一个函数摘要由一系列的 ⟨P, UActs⟩ 对来组成，形式定义为：

⟨P1, UActs1⟩ ∪ ⟨P2, UActs2⟩ ∪ ... ∪ ⟨Pn, UActsn⟩

* P 表示从内存状态转移图中起始节点到叶子节点的路径约束。
* UActs 是从起始到叶子节点中抽取出来的内存对象行为集合。

### 函数摘要生成

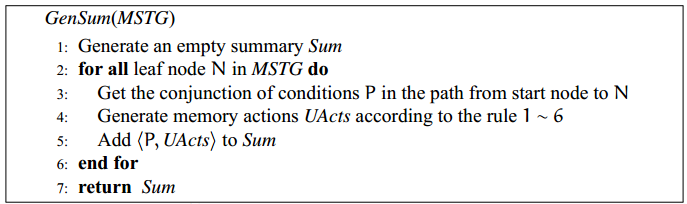


图 7 函数摘要的生成算法

抽取规则如下：

1. 若堆对象的终止状态为 Freed 或 Relinquished，不收集与这个对象相关的内存行为。
2. 若堆对象的终止状态为 Leaked，不收集与该对象相关的内存行为。产生一个内存泄漏报告。
3. 若堆对象的终止状态为 Escaped，所有的 ToExtern 和 ToReturn 的行为会被收集保存下来。
4. 若外部对象的终止状态为 Freed，将 ToFree 行为保存下来。
5. 若外部对象的终止状态为 Relinquished，将 ToUnknown 行为保存下来。
6. 若外部对象的终止状态为 Escaped，所有 ToExtern 和 ToReturn 行为会被收集保存下来。

### 函数摘要应用

具体的算法描述如下：

1. 采用 Access Paths 将实参与形参进行绑定。（Acess Paths 最早由 Landi 和 Ryder 的工作 [52] 中引入，用于通过内存位置来表示的符号名。）
2. 判断 P 的可满足性。用于有效减少不可行的 UActs 的应用。
3. 在函数调用点应用 UActs。在 UActs，每个元素都可以用 m -> r 来表示。（其中左边的 m 表示一个堆对象或外部对象。在应用时，首先会将堆对象当成函数调用者新分配了一个堆对象，而每个外部对象会被 Access Paths 映射成函数调用者上下文中实际的内存对象。 m -> r 右边的 r 可以用一个返回值、free符号、unknown符号或者外部指针来表示。）

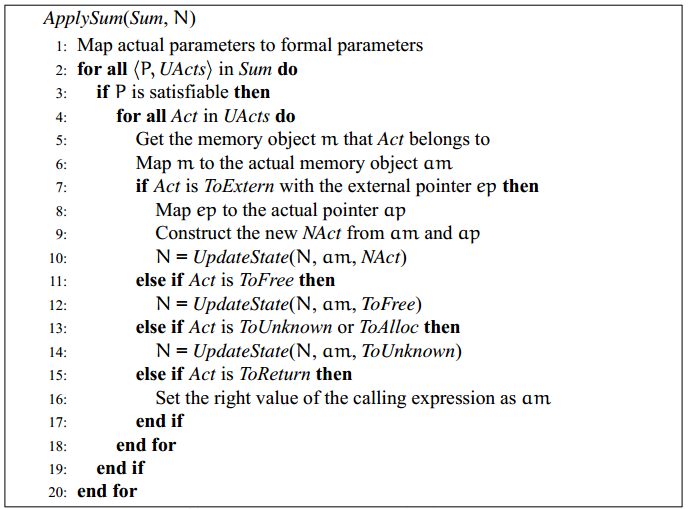


图 8 应用函数摘要的算法

**示例** 考虑如下例子：

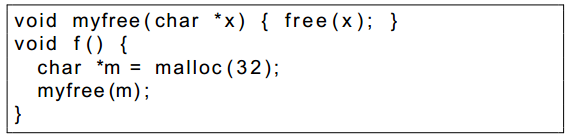


图 9 函数摘要应用示例

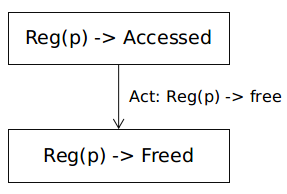


图 10 函数f的内存状态转移图MSTG

分析过程：

1. 当该函数被调用时，首先将 Reg(x) 映射到实参上（也就是由malloc分配的堆对象）
2. 然后再将 ToFree 应用到该堆对象上

生成函数摘要：

函数myfree 的摘要可以表示为 Reg(x) -> free

## 分析示例

使用以下代码演示整个内存泄漏检测的过程：

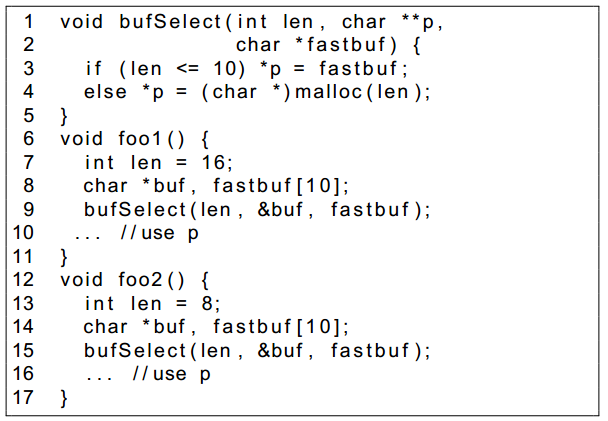


图 11 内存泄露检测示例

### 检测过程

1. 生成调用图：见图12
2. 遍历调用图：
3. 为每个函数生成内存状态转移图MSTG：如图13所示，将函数bufSelect 转换成内存状态转移图。在第一个节点，通过函数参数引入的外部对象 Reg(p)、 Reg(∗p) 和 Reg(fastbuf) 被初始化为 Accessed 状态。然后计算第一个路径条件len <= 10 的可行性：若可行则分析\*p = fastbuf 所在的分支，由于\*p 属于外部对象，因此 Reg(fastbuf) 转移到 Escaped 状态；若不可行，则分析另外一个分支，即!(len <= 10，该分支中分配了一个新的堆对象并赋给\*p，该堆对象的状态从 Allocated 转移成 Escaped 状态。
4. 为每个函数生成函数摘要：在函数bufSelect 分析完成后，从bufSelect 的内存状态转移图中生成相应的函数摘要。在内存状态转移图中的每条路径都会被抽取成函数摘要的一部分，即一个 ⟨P, UActs⟩ 对。图14给出了函数bufSelect 的函数摘要。
5. 将函数摘要应用到函数的被调用点上：在第 9 行，变量len 的值大于 10，因此函数bufSelect 分析了一个新的堆对象给指针buf。如果该堆对象在函数foo1 的最后没有释放，那么就会造成内存泄漏。在第 15 行，由于第二个分支的条件!(len <= 10) 是不可行的，所以不会产生误报。

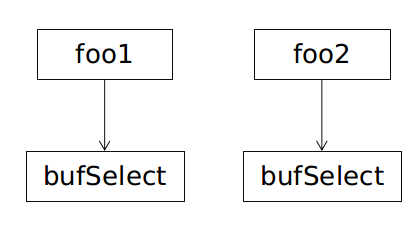


图 12 调用图

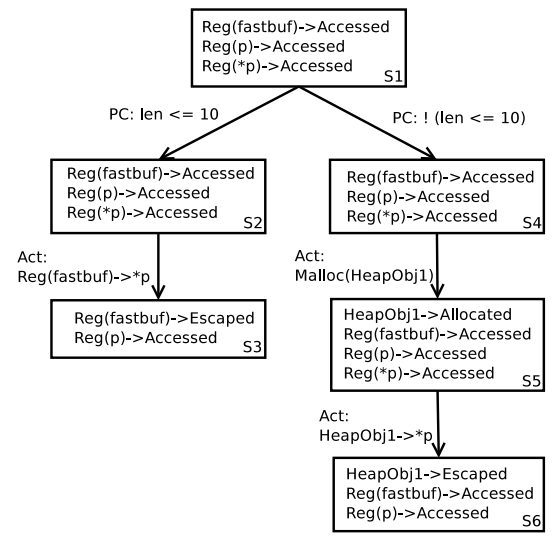


图 13 函数bufSelect的内存状态转移图MSTG

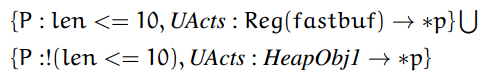


图 14 函数bufSelect的函数摘要