进程对系统环境稳定性影响的评估算法

专业：计算机软件与理论

学位申请人：王斌强

指导老师：周青副教授

摘　要

在计算机科学中多年的研究与实践后，我们发现依据稳定性的传统定义是很难使得计算机系统获得稳定性的，因为单一的一个计算机系统显然是不可能对在其中运行的所有进程都保持稳定。我们能够预期的只能是系统对其中的某一些进程是稳定的，所以这个时候就需要找到一个方法来确定哪些进程在这个系统中运行是不会破坏系统稳定性的,是适合这个系统的。基于这些观察，在这篇文章中，我们提出了一个基于进程的系统稳定性定义，并且设计了一个基于知识的评估算法，该算法能够被用来评估进程对系统环境稳定性的影响。这个算法是在一阶谓词形式化系统中建立的,这个形式化系统包括了计算机系统的各种属性，关于该系统的知识，以及该系统中所有可能的动作。最后,本文证明了该评估算法的评估结论与我们的稳定性定义的一致性。

关键词：稳定性、系统环境、进程、一阶谓词逻辑、评估算法

An algorithm for evaluating effects of processes on stabilization of system environments

Major: Computer Software and Theory

Name: Wang BinQiang

Supervisor: Asso. Prof. Zhou Qing

Abstract

After decades of practice and research in computer science, we found that it is very difficult to obtain stabilization of computer systems from traditional definitions of stabilization. It is obvious that a single computer system is not impossibly stable for all the processes running on it. All we can expect is that a computer system can be stable for a part of these processes. Then we need a way to determine which processes are suitable on the system without breaking the stabilization of the system. Based on this observation, we propose a new definition of stabilization from which both computer systems and processes are considered. Then we propose a knowledge based algorithm which evaluates effects of processes on stabilization of system environments. The algorithm is based on a first order predicate formal system, which contains the properties of a computer system and our knowledge on the computer system and all the possible actions in the system. The paper concludes with a comprehensive discussion on the consistence of our definition of stabilization and the corresponding algorithm proposed in this paper.

Key words: stabilization、system environment、process、predicate logic of first order、evaluation algorithm

第 1 章 前言

1.1 选题的背景与意义

自从 1973 年 Dijkstra[1]提出自稳定性概念以来，系统稳定性的研究已经得到了计算机科学家们越来越多的重视。

直观的描述，自稳定性[1]指的是计算机系统能够从任意的系统状态通过有限的动作自动的调整到一个它当前要执行动作所需要的系统状态的能力，而稳定性[2]指的是计算机系统能够从某一些满足特定要求的系统状态出发通过有限的动作自动的调整到一个它当前要执行动作所需要的系统状态的能力。从应用范围来说，稳定性要比自稳定性更大。

设计稳定性的目的与动力[3]大部分是为了去提供一种容错，通过这种容错，系统能够自发的从发生错误的情况下恢复过来，并且最终继续执行它所需要的动作。尤其是，稳定性经常的被用于解释计算机系统中的错误的发生[4]，错误的恢复[4][5]，以及可容的错误类型[6]等等。

对于一个一般的计算机用户来说,拥有一个具备稳定性的计算机系统是非常重要的,因为当计算机出现错误时,系统如果能够自动的恢复到正常的状态,而不需要用户去进行一些外部的操作,这样不仅带来巨大的便利也可以避免用户由于系统知识的缺乏或处理失误而带来额外的损失。同时,对一些要求非常高的大型系统,如银行系统,电信的收费系统等,稳定性就显得更加重要。这些系统如果不能够在出现错误的情况下,自动的进行恢复，而仅仅是简单的将系统重设，必然会导致数据丢失或其他各种可能的损坏，给企业与国家带来巨大的经济损失。所以研究稳定性对于设计一个计算机系统来说无疑是非常重要，也是非常有意义的。

1.2 本文研究的重点

在对系统稳定性传统定义研究的基础上，我们希望使用一阶谓词演算系统来形式化的描述计算机系统，并建立起相应的基于进程的稳定性定义。同时，在这个形式化系统之上，希望能够找到一个基于知识的进程评估算法，它可以评估每个在系统环境中运行的进程对于该系统的稳定性的影响。

1.3 本文的结构

这篇论文主要分为以下几个部分：

第 1 章：选题的背景与意义，以及本文的研究重点，结构

第 2 章：介绍稳定性，稳定性的定义的研究现状，稳定性与容错，以及本文对稳定性研究的思路与创新点

第 3 章：介绍一阶谓词演算系统

第 4 章：用一阶谓词演算系统来形式化的描述计算机系统，并描述相应的基于进程的系统稳定性定义

第 5 章：阐述了一个基于知识的进程评估算法，并证明其评估结论相对于我们的稳定性定义的一致性

第 6 章：列举了一个评估过程示例

第 7 章：总结与展望

第 2 章 稳定性

2.1 相关稳定性的研究情况

2.1.1 自稳定性概念的提出

Edsger W.Dijkstra[1]在 1973 年第一次将分布式控制当中的自稳定性概念介绍到计算机科学中，具体的描述如下：

他所描述的系统是一个有限状态进程的连接图，其中直接连接的进程被称为邻居。定义一个关于进程与它的邻居的状态的布尔函数为权（privilege）,表示是否有一个转换操作对于该进程是有效的。他的自稳定性的定义被分为两个阶段。在第一个阶段中它首先定义什么是合法的状态：

1）在每一个合法的状态中，存在一个或多个权是处于活动状态的（present）。
2）在每一个合法的状态中，每一次转换操作都将使得系统进入下一个合法的状态。
3）每一个权至少在一个合法状态中是活动的。
4）对于任何两个合法的状态，至少存在一个转换操作序列可以使得其中一个转换到另一个。

在第二个阶段中，他使用下面的定义来描述自稳定性：

我们称一个系统是“自稳定的”的当且仅当无论初始的状态是什么以及无论下一步操作哪个权被选择为活动的，这个系统在当前情况下至少有一个权是活动的,并且系统将在有限的转换操作之后使自己处于合法的状态。

但这实际上，也仅仅是自稳定性概念的一个简单描述，在下面将介绍自稳定性的更加严格的定义。

2.1.2 自稳定性定义

Marco Schneider[7]由 Dijkstra 所描述的概念给出了自稳定性的详细定义：

一个系统是由两部分组成的：进程与进程间的联系（如共享内存，信息管道等），系统的拓扑图是一个进程作为结点以及联系作为边的有向图。系统中每一个组件都有一个局部状态（local state）,而系统的全局状态（global state）为局部状态的合集。一个系统的行为包括了三个集合:状态集合，状态集合中的状态之间的转换关系以及在转换关系上的公平（fairness）规则集合。

接下来就定义了在全局状态集合中，系统关于一个谓词 P 的自稳定性，谓词 P 描述了状态集合中部分正确的运行状态。当系统 S 满足下述两个条件时，它关于谓词 P 是自稳定的：

1）封闭（Closure）:系统运行下，P 是封闭的。也就是说，一旦 P 在 S 中成立，它将不会被改变；
2）收敛（Convergence）:从任意一个全局状态出发，S 都保证可以在有穷的步骤内达到一个全局状态满足 P。

以上就是自稳定性详细的定义，但这样的定义存在一个问题，就是在一些的情况下,要求系统在任意的初始状态下都收敛也许太严格了。例如，某些错误的出现仅仅扰乱了某一个状态集合，而并不是所有的系统状态。为了解决这个问题, Arora and Gouda and Arora[2] 提出了自稳定性的一个衍生属性---稳定性。

2.1.3 稳定性定义

Q 与 P 都是关于全局状态的谓词，一个系统对于 P 称为 Q-稳定的当且仅当它满足下述两个条件：

1）封闭（Closure）:系统运行下，P 是封闭的。也就是说，一旦 P 在 S 中成立，它将不会被改变；
2）收敛（Convergence）:从任意一个满足 Q 的全局状态出发，S 都保证可以在有穷的步骤内达到一个满足 P 的全局状态。

注意到，假如 S 关于 P 是自稳定的，那么也可以说 S 对于 P 是 true-稳定的，所以稳定性定义在很多时候要比自稳定性应用的范围要广。

但上述基于稳定性的研究始终都存在着两个问题:

1. 稳定的系统的定义太严格了。一些感觉上是稳定的系统事实上依据定义并不稳定；
2. 由于定义的过于严格,使得去设计一个稳定的系统或验证它们的正确性变得非常困难。

针对上面的两个问题, Mohamed Gouda 提出了一个非常有趣的概念,多级稳定性（multiphase stabilization）[8]，它把一个系统定义为一个变量集合与一个已分级的动作集合，动作被系统用来改变变量的赋值。定义可简单描述如下：

一个系统 S 关于一个状态谓词 Q 被称为 n-稳定的当且仅当系统有状态谓词 P.0,…,P.n, 其中 P.0 = true, P.n = Q，它们同时满足下述的两个条件：

1.假如系统的初始状态满足 P_{j-1} 且只要 S 中被允许的动作的级数小于或等于 j，那么系统 S 就可以达到一个满足 P_j 的状态；

2.满足 P_j 的状态集合对于级数小于 j 的动作是封闭的。

这个定义使得验证和设计稳定的系统变得更加的简单，而且原来某些依据一般定义不稳定的系统在多级稳定性的定义中则是稳定的。但其中也存在着缺陷，就是对系统的动作进行分级这本身就是一个比较难以处理的问题，它增加了对程序员的要求，因为几乎每个程序员都必须非常详细的了解整个系统是如何进行动作分级的。

2.1.4 其他稳定性定义

1981 年 Lehman 和 Rabin 提出一个特殊的自稳定性定义，概率（probabilistic）自稳定性[9]。一个系统 S 被称为关于谓词 P 满足 概率（probabilistic） 自稳定性当且仅当：

1） 封闭：系统运行下，P 是封闭的。也就是说，一旦 P 在 S 中成立，它将不会被改变；

2） 收敛：存在一个满足 $\lim_{k\to\infty} f(k)=0$ 的自然数到闭区间[0，1]的函数以至于从任意全局状态开始，S 在 k 步内将达到一个全局状态满足 P 的概率至少是 1-$f(k)$.

1990 年,Burns eta 介绍了一个比自稳定性要弱一些的概念---伪稳定性（pseudo-stabilization）[10]。用一般的语言来说，一个系统对一个谓词 P 是伪稳定的，当且仅当在所有的计算下，从任意的状态出发，P 永远成立（hold)。换句话说，无论 P 是否在很多时候是成立或者不成立的，P 最终都将成立。实际上可以认为伪稳定性是放弃了自稳定性中封闭的条件。

2001 年，Mohamed Gouda 提出了有关计算系统的一个新的属性，弱稳定性（weak stabilization）[11]。所谓的弱稳定性指的是在稳定性的收敛条件中，系统 S 只需要存在一个计算，它使系统最终达到所需要的状态。同时一个系统满足稳定性就一定满足弱稳定性。而且他也验证了一个重要的结果,即满足弱稳定性的系统如果执行是强平均(strongly fair)的，那么这个系统也满足稳定性。弱稳定性实

际上可以说是稳定性的一个好的近似值。

目前为了去获得系统稳定性的精髓，还有许多其它的不同的定义被提出，而以上仅仅是一些我们认为最具代表性的定义。

2.1.5 稳定性与容错性

瞬间的失败（transient failure）[12]：指的是系统在一段有限的时间内出现了一些错误的状态，在此之后系统仍能够正常的执行动作。

自稳定性[7]介于前期的避免错误方式与容错性之间，它实际上提供了一个统一的方法去处理瞬间的失败,它的方式是把这些瞬间的失败形式化的合并到设计模型当中，它是对失败的反思。容错性与可靠性通常是一个整体而不是系统设计中固定的一部分，也不是对失败的反思。从容错性与可靠性这方面考虑可能我们很难解决瞬间失败的问题，而一个自稳定的系统则符合一个更强的，更令人满意的纠错性的概念。换句话说，如果一个瞬间的失败发生了并且造成了矛盾的系统状态，自稳定的系统将在不需要外力的情况下，最终调整自己达到需要的状态。

我们可以注意到自稳定性与传统的处理状态中错误的方法（如 replication 与 error correction）是非常不同的：传统的处理方法通常是去掩盖错误以达到避免失败的发生，而自稳定的系统采用的是恢复的方式。从这个方面来说，自稳定性是容错性中其他方法的一个很好的补充。

但是，自稳定性一直存在着一个缺点，那就是作为一个设计目标来说，自稳定性太强了，很难去获取这样一个目标。但是,由于瞬间的失败一般通常都不是任意的，所以如果能从一些指定的瞬间失败中恢复也许是更加有意义的。这样，我们就可以由稳定性来建立一个系统：与从任意的状态恢复不同，这个系统只要状态集合满足某些特定要求，那么它就可以恢复到所需要的状态，这个特定的状态集合由谓词 Q 来描述（这个集合中的状态也许是由一些失败造成的）。这样，

一个系统获得稳定性要比获得自稳定性更加的实用。

自稳定性与稳定性都能够被表示为一个系统容纳所有错误的能力。我们可以把错误当作一种系统行为，在这种系统行为中状态空间与转换关系都被扩大了。这样在这个扩大了的系统中的稳定性与自稳定性就能够被用来说明原来系统的容错性。

2.2 本文的思路与创新点

2.2.1 思路

依据上述的稳定性定义的描述，我们可以发现要设计一个稳定的计算机系统则必须是系统中运行的所有进程都要满足稳定性，这相对于现代的计算机系统实现起来是非常困难的，因为一个计算机系统要运行什么样的进程实际上我们是无法在设计阶段就能够确定的，也更无法确定是否在任意时刻系统中运行的每个进程都可以保证系统的稳定性。

而从进程与系统的关系这一方面,我们可以这样概括稳定性:一个系统具有稳定性就是在每个进程运行完毕后,系统能够恢复到该进程运行前的那个环境(全局状态)，或者恢复到一个能达到系统正常运行标准的环境。如果我们能够保证系统对在其中运行的每个进程都具备这样的性质,那么就可以说这个系统是具备稳定性的，是能够正常的运转的。要实现系统中运行的所有进程都具备上述性质,我们就必须找到一个方法去确定哪些进程在系统中运行是不会破坏系统稳定性的，是适合这个系统的,而哪些是不适合的。寻找这样的一个关于稳定性的进程评估方式是在这篇文章中将要探讨的问题，而在探讨这个问题之前,我们必须建立起系统关于进程的稳定性定义。

此外,我们对系统以及其他概念都将进行一个更深入的形式化描述，这将会对稳定性的研究工作带来非常多的帮助，包括设计基于知识的进程评估算法。

2.2.2 创新点

本文的创新点主要表现在以下三个方面：1)使用一阶谓词演算系统来形式化描述计算机系统，这是在以往的稳定性研究中所没有的；2)提出了基于进程的系统稳定性定义，该定义具体的描述了系统在什么情况下相对于什么样的进程是稳定的；3)得到了关于稳定性的基于知识的进程评估算法，它可以用于评估每个在系统中运行的进程对系统环境稳定性的影响。

第 3 章 一阶谓词演算系统

一阶谓词演算系统是最为经典的符号逻辑系统，其它的逻辑系统都可以看作谓词演算系统的扩充、推广和归约，而且一阶推理系统拥有强大的推理能力，如果可以用一阶谓词演算系统来形式化描述我们的计算机系统，对稳定性的研究将带来很多有利的条件。下面我们将简单介绍一下一阶谓词演算系统[15]：

3.1 谓词及谓词公式

在谓词逻辑中，命题是用谓词表示的，一个谓词可分为谓词名和个体两个部分。

个体表示某个独立存在的事物或者某个抽象的概念:谓词名用于刻画个体的性质、状态或个体间的关系。例如，对于“小王和小李是朋友”这个命题可用谓词表示成 Friend (wang, li)，其中“Friend”是谓词名，"Wang”和“Li”是个体，"Friend"刻画了“Wang”和“Li”的关系特征。

谓词的一般形式是

$$P(x_1, x_2, x_3, \cdots, x_n,)$$

其中 P 是谓词名，通常用大写英文字母表示;$x_1, x_2 \ldots\ldots x_n$ 是个体，通常用小写英文字母表示。

在谓词中，个体可以是常量，也可以是变元，还可以是一个函数。例如，对于“小张的妈妈是医生”这一命题可以表示为:Doctor (Mother (zhang))，其中"Mother(zhang)”是一个函数。个体常量、个体变无、函数统称为“项”。在用谓词表示客观事物时，谓词的语义是由使用者根据需要主观定义的。

谓词中包含的个体数目称为谓词的元数。例如 $P(x_1, x_2, x_3, \cdots, x_n)$就是 n 元谓词。如果每个 x_i 都是个体常量或变元，则这个谓词就是一阶谓词;如果 x_i 本身又是一个一阶谓词，则 $P(x_1, x_2, \cdots, x_n)$为二阶谓词，依次类推。我们所研究的谓词都是指的一阶谓词。谓词中个体变元的取值范围称为个体域，当谓词中的变元都用个体域中的个体取代时，谓词就具有一个确定的真值:T 或 F。

一阶语言 L 的合式公式是由命题逻辑中的五个真值联结词:“$\neg$”、“$\wedge$”、“$\vee$”、“$\rightarrow$”、“$\leftrightarrow$”和引入的两个刻画谓词与个体关系的量词“$\forall$”和“$\exists$”按照一定的规则构成的。其中“$\forall$”表示对个体域中的所有个体;“$\exists$”表示对个体域中的至少一个个体。谓词演算的合式公式可按下述规则得到:

(1)单个谓词是合式公式，称为原子谓词公式;

(2)若 A 是合式公式，则$\neg$A 也是合式公式;

(3)若 A, B 都是合式公式，则 $A\wedge B, A\vee B, A\rightarrow B, A\leftrightarrow B$ 也都是合式公式。

(4)若 A 是合式公式，x 是任一个体变元，则$(\forall x)A$ 和$(\exists x)A$ 也都是合式公式。

在合式公式中，连接词的优先级别由高到低为: $\neg$、$\wedge$、$\vee$、$\rightarrow$、$\leftrightarrow$。位于量词后面的单个谓词或者用括弧括起来的合式公式称为量词的辖域。辖域内与量词中同名的变元称为约束变化，不受约束的变元称为自由变元。

3.2 谓词公式的解释

在命题逻辑中，对命题公式中各个命题变元的一次真值指派为命题公式的一个解释，一旦解释确定后，根据各联结词的定义就可求出命题公式的真值(T 或 F)。但在谓词逻辑中，由于公式中可能有个体常量，个体变元以及函数，因此不能像命题公式那样直接通过真值指派给出解释，必须首先考虑个体常量和函数在个体域中的取值，然后才能针对常量与函数的具体取值为谓词分别指派真值。由于存在多种指派组合情况，所以一个谓词公式的解释可能有很多个，对于每一个解释，谓词公式都可求出一个真值(T 或 F)。

下面给出谓词公式的解释的定义，然后用例子说明如何构造一个解释，以及根据解释求出谓词公式的真值。

定义 1 设 D 为谓词公式 P 的个体域，若对 P 中个体常量、函数和谓词按如下规定赋值:

(1)为每个个体常量指派 D 中的一个元素。

(2)为每个 n 元函数指派一个从 D^n 到 D 的映射，其中

$$D^n = \{(x_1,x_2, \ldots, x_n) \mid x_1,x_2, \ldots, x_n \in D\}$$

(3)为每个 n 元谓词指派一个从 D^n 到{F,T}的映射。

例如，设个体域 D=(2,3}，求公式 B=(∀x)(P(x)→Q(f(x),b))在 D 上的某一解释，并指出公式 B 在此解释下的真值。

解:设对个体常量 b，函数 f(x)指派的值分别为:

b=2; f(2)=3; f(3)=2;

对谓词指派的真值为:

P(2)=F; P(3)=F, Q(2,2)=T，Q(3,2)=F.

上述指派就是对公式 B 的一个解释。在此解释下，当 x=2 时，有

P(2)=F, Q(f(2),2)=Q(3,2)=F

所以，P(2) →Q (f(2),2)真值为 T;当 x=3 时有

P(3)= T, Q(f(3),2)=Q(2,2)=T

所以，P(3) →Q(f(3),2)的真值为 T，即对 D 中的任一 x 均有 P(x)→Q(f(x),b)的真值为 T，所以公式 B 在此解释下的真值为 T，若公式 P 在解释 I 下真值为 T，则称 I 为公式 P 的一个模型。

如果谓词公式 P 对个体域上的任何一个解释都取得真值 T，则称 P 在 D 上是永真的。如果 P 在每个非空个体域上均永真，则称 P 永真。

对于谓词公式 P，如果至少存在一个解释使得公式 P 在此解释下的真值为

T,则称公式 P 是可满足的，谓词公式的可满足性又称为相容性。

如果谓词公式 P 对于个体域 D 上的任何一个解释都取得真值 F，则称 P 在 D 上是永假的。如果 P 在每个非空个体域上均永假，则称 P 永假。永假性又称不可满足性或不相容性。

3.3 谓词公式的等价性与永真蕴涵

设 D 是 P 与 Q 两个谓词公式的共同个体域，若对 D 上的任何一个解释，P 与 Q 均有相同的真值，则称 P 与 Q 在 D 上是等价的;如果 D 是任意个体域，则称 P 与 Q 是等价的，记作 “P↔Q"。因为所有命题公式都是一阶公式，所以所有命题公式的等价式都是一阶逻辑的等价式，只不过其中的命题变元可用谓词代换和增加了如下关于量词的等价式而已:

$\neg(\exists x) \leftrightarrow (\forall x)\neg P;\neg(\forall x)P \leftrightarrow (\exists x)\neg P;$

$(\forall x)(P \wedge Q) \leftrightarrow (\forall x)P \wedge (\forall x)Q;$

$(\exists x)(P \vee Q) \leftrightarrow (\exists x)P \vee (\exists x)Q.$

对于谓词公式 P 和 Q，如果 P→Q 永真，则称 P 永真蕴涵 Q，称 Q 为 P 的逻辑结论，记作 P=>Q。谓词逻辑的永真蕴涵式除了命题逻辑形式的永真蕴涵式外还包括:

$(\forall x)\ P(x) => P(y)$(y 是个体域 D 中的任一个体);

$(\exists x)\ P(x) => P(y)$ (y 是个体域 D 中使 P(y)为真的个体)。

第 4 章 基于进程的稳定性定义

4.1 系统的形式化描述

首先，使用一阶谓词演算系统对系统以及其他概念进行形式化描述:

定义 1：一个系统 S 是一个有穷的谓词集合；

$S = \{q_1(x_1,x_2,\ldots,x_{k1}),q_2(x_1,x_2,\ldots,x_{k2}),\ldots,q_n(x_1,x_2,\ldots,x_{kn})\}$,$q_1,q_2,\ldots,q_n$ 为 n 个谓词符号;

系统中每个谓词 q_i（$x_1,x_2,\ldots,x_{ki}$）中$(x_1,x_2,\ldots,x_{ki})$的取值范围用集合 Q_i表示,它是有穷的;

状态为一个无变元公式，可表示为 $q_i(a_1, a_2,\ldots,a_{ki})$,其中 q_i 为系统 S 的一个谓词符号，且$(a_1, a_2,\ldots,a_{ki})\in Q_i$;

系统的环境用 SA 表示，$SA= \bigcup_{i=1}^{n}\{q_i(a_1,a_2,\ldots,a_{ki})\}$，其中 $q_i(a_1, a_2,\ldots,a_{ki})$为系统的一个状态,$(a_1, a_2,\ldots,a_{ki})\in Q_i$；系统当前环境用 SE 表示，$SE=\bigcup_{i=1}^{n}\{q_i(a_1,a_2,\ldots,a_{ki})\}$且状态 $q_i(a_1, a_2,\ldots,a_{ki})$，$(a_1, a_2,\ldots,a_{ki})\in Q_i$,i = 1..n,在系统当前时刻为真；

S 中的一个动作的表达式为 act_i:$\{q_{k1}(\alpha_{k1})\Rightarrow q_{k1}(\beta_{k1})$，$q_{k2}(\alpha_{k2})\Rightarrow q_{k2}(\beta_{k2}),\ldots, q_{kj}(\alpha_{kj})\Rightarrow q_{kj}(\beta_{kj})\}$，$\alpha_{k1}, \beta_{k1}\in Q_{k1}, \ldots, \alpha_{ki},\beta_{ki}\in Q_{ki}$,其中符号“$\Rightarrow$”表示把系统环境中一个状态更新为另一个新的状态，$q_{k1}$，$q_{k2}, \cdots, q_{kj}$ 为系统中的谓词符号，act_i 为动作标识，在系统中标识整个动作，$\{q_{k1}(\alpha_{k1})$，$q_{k2}(\alpha_{k2}),\ldots, q_{kj}(\alpha_{kj})\}$称为此动作的前驱状态集，$\{q_{k1}(\beta_{k1})$，$q_{k2}(\beta_{k2}),\ldots, q_{kj}(\beta_{kj})\}$称为此动作的后驱状态集.

我们知道，计算机系统都包含着这各种各样的属性，如内存量，输入输出接口等等，我们可以通过对这些属性的状态的描述来勾画出整个计算机系统当前所处的环境状况。

在我们的定义中使用谓词来描述系统的某一个属性，而由于一个系统是由有穷个属性所组成的，因此一个系统就可以被形式化描述为一个谓词的有穷集合，这个谓词集合包括了该系统的所有属性。

系统的状态是一个无变元公式,它描述的是系统某一个属性的赋值。

系统环境指的是系统所有属性的一个赋值，系统当前环境是系统环境的一个特殊情况，描述的是系统在当前时刻所有属性的赋值。

系统的动作指的是系统用来更新系统环境的行为，它可以说是一个状态集合到状态集合的映射，由于每个谓词 $q_i(x_1,x_2,\ldots,x_{ki})$中$(x_1,x_2,\ldots,x_{ki})$的取值范围 Q_i 是有穷的，且系统只有有限个属性，所以一个系统所能运行的所有动作的总数也是有穷的。我们可以用动作标识来表示系统中每一个动作，它是一个名词符号。

有了系统的各种要素的定义,下面我们将给在系统环境中运行的进程下一个定义：

定义 2: 进程是系统动作的一个非空有限序列，在系统的环境中运行,用来改变系统的环境状况，可表示为$<act_1,act_2,\ldots,act_n>$, $act_1,act_2,\ldots,act_n$ 为系统的动作标识。

4.2 稳定性定义

目前来说，稳定性的定义大都是针对整个系统而言的，也就是说一个系统

如果是稳定的，那么系统中的所有进程都必须依据一定的状态要求，要么对这个状态要求是封闭的，也就是说，进程完成后的系统仍满足状态要求；要么进程对这个状态要求是收敛的，也就是说，进程完成后的系统可以通过执行另一个进程使得自身满足状态要求。这给我们实际的在计算机中设计一个稳定的系统带来了非常大的难度，因为在大部分的计算机系统中，由于系统所处的当前环境不同，同样的进程得到的结果环境也不一定相同。所以要保证所有的进程运行完毕后，计算机系统都能够保持稳定是不太可能的。所以，在这里我们将建立一个新的基于进程的系统稳定性定义，并且在文章的后一个部分，通过这个稳定性定义我们将提出一个算法，它可以评估单个进程在某个初始环境下对于系统稳定性有何影响。

首先，我们需要定义一个无变元公式与一个无变元公式集合在环境中成立的概念：

定义 3： 关于系统 S 的一个无变元公式 A，如果 A 在 S 的环境 SA 中为真，那么 A 在 SA 中成立，反之，在 SA 中不成立。

例如：有关于计算机系统 S 的公式 p(0k)，p(x)表示“内存容量大于 x”，且 S 的当前环境中有状态 Memory(10k),表示“内存容量等于 10k”。我们知道状态 Memory(10k)蕴涵了公式 p(0k)，所以公式 p(0k)在系统当前环境中是成立的。

定义 4： 在系统 S 的一个环境中，如果无变元公式 P_1，P_2，…，P_m都成立，那么公式集合 $P = \{P_1,P_2,\ldots,P_m\}$在此环境中也成立,反之如果存在一个公式 P_i，$P_i \in P$, 在此环境中不成立，则 P 也在此环境中不成立。

在这里必须要说明的是系统中任何一个环境 SA 只会在自身中成立,而不能在其他任何一个不同的环境中成立,因为状态描述的是系统中某属性的一个赋值,所以不可能在一个环境中一个属性的两种赋值都为真。

下面，在给出稳定性定义之前，我们将先给出封闭性与收敛性的定义:

定义 5: Q 是系统 S 中的一个无变元公式集合;

Q 对于进程 PS 是封闭的是指当且仅当从任意一个 Q 成立的系统环境中开始运行进程 PS，当进程 PS 运行结束，公式集合 Q 在系统新的环境中仍然成立。

更直观的描述封闭性：公式集合 Q 实际上描述了对于系统状态一些要求，一个进程在系统某个环境中运行，这个环境满足 Q 中所有要求，且该进程运行完毕之后的新的系统环境仍满足这些要求，那么这些要求相对于这个进程就是封闭的。

例如：一个计算机系统的当前环境中有公式集合{Print(0):打印机空闲,Memory(10k):空闲内存大于 10k}成立，且这时某打印进程在系统当前环境中开始运行，打印完毕后，它释放了所有已占用的打印机资源与内存资源，因此公式集合{Print(0),Memory(10k)}仍然在该打印进程运行完毕后的系统当前环境中成立，从而公式集合{Print(0),Memory(10k)}相对于此打印进程在当前环境中就是封闭的。

定义 6: R 与 Q 都是关于系统 S 的无变元公式集合;

在系统 S 中，R 收敛到 Q 当且仅当对于每个 R 成立的系统环境，Q 要么成立，要么存在一个进程 PS,通过在环境中运行 PS，可以得到一个新的环境，Q 在此环境中成立。

例如：假设有一个系统 S,R 为公式集合$\{q_1(0), q_2(0)\}$,其中 $q_1(x)$为 S 中的一个谓词,表示“属性 A 的值等于 x”，$q_2(x)$也是 S 中的一个谓词，表示“属性 B 的值等于 x”； Q 为 S 的一个公式集合$\{ p_1(0), p_2(0)\}$， 其中 $p_1(x)$表示“属性 A 的值大于 x”，$p_2(x)$表示“属性 B 的值为大于 x”。如果系统 S 在 R 成立的任一当前环境中都可以运行进程 ps = <ACT1: $\{q_1(0) \Rightarrow q_1(1)\}$，ACT2: $\{q_2(0) \Rightarrow q_2(1)\}$>,

那么我们就可以说在系统 S 中，R 收敛到 Q。

定理 1:R,Q,T 都是系统 S 中的无变元公式集合，如果 R 收敛到 Q，且 Q 收敛到 T，那么 R 收敛到 T。

证明:

假设系统中任一个环境 SA, R 在 SA 中成立。

由 R 收敛到 Q 以及收敛的定义,我们可以得到 Q 要么在 SA 中成立，要么存在一个进程 PS,通过在当前环境中运行 PS，可以得到一个新的环境 SA'，Q 在 SA'中成立。

1.如果 Q 在 SA 中成立,那么由 Q 收敛到 T,我们可以得 T 要么在 SA 中成立，要么存在一个进程 PS_1,通过在当前环境中运行 PS_1，可以得到一个新的环境，T 在此环境中成立。

由收敛性定义有 R 收敛到 T。

2.如果 Q 在 SA'中成立，那么由 Q 收敛到 T 我们可以得 T 要么在 SA'中成立，要么存在一个进程 PS_2,通过在环境中运行 PS_2，可以得到一个新的环境 SA"，T 在 SA"中成立。

如果 T 在 SA'中成立,则存在进程 PS,它可以把 R 成立的环境 SA 更新为 T 成立的环境 SA',所以由收敛性定义有 R 收敛到 T。

如果 T 在 SE"中成立,则我们也就可以找到一个进程 $PS*PS_2$,其中""表示进程 PS 运行后接着运行 PS_2,它可以把 R 成立的环境 SA 更新为环境 SA",且有 T 在 SA"中成立,所以由收敛性定义有 R 收敛到 T。*

定理得证。

定理 1 表示收敛性具有传递性。

定义 7: 基本要求集是一个关于系统 S 的无变元公式集，可表示为 $P = \{P_1,P_2,P_3,\ldots,P_m\}$,其中 $P_1,\ldots,P_m$ 为关于 S 的无变元公式;它描述了系统 S 正常运转所需要的最基本条件。

可以更形象的描述基本要求集:它指的是在我们一个系统中,总存在一些比较关键的属性,如果希望系统能够正常的运转,这些关键的属性的状态必须满足一定的基本要求.例如:某一个计算机操作系统中,有两个关键的属性:空闲内存量,CPU 资源数量,只有当我们的空闲内存量大于 10k,以及 CPU 资源数量被占用少于 80% 时,我们的系统就能正常运行,所以我们用公式 Mem(10k)表示“空闲内存量大于 10k”，用公式 C(0.8)表示 “CPU 资源数量被占用少于 80%”，这个时候,这个系统的基本要求集就是{Mem(10k)，C(0.8)}。

定义 8: 依据一个基本要求集 Q，系统 S 对于一个进程 PS 是稳定的当且仅当从任意一个 Q 成立的系统环境出发，要么 Q 对于进程 PS 是封闭的，要么在此进程运行结束后的系统环境可以收敛到 Q。

这个定义也就是说，我们要评价一个系统对于某个进程是否是稳定的就必须满足关于系统状态的一些基本要求，这些要求不仅在进程运行之前必须被满足，且必须在进程执行之后要么仍然被满足，要么系统可以调用某个进程改变环境使这些要求被满足。如果无法达到上述要求，则此时系统对这个进程就不稳定。

例如：某计算机系统 W 的基本要求集为 P，P= {Mem(10k) }，其中公式 Mem(10k)表示“空闲内存量大于 10k”，且 P 在 W 的当前境中成立，一个进程 ps 在 W 的当前环境中运行，运行完毕之后并没有释放内存，且这个时候系统的当前环境中有状态 M(5k)成立，M(5k)表示“空闲内存量为 5k”。由于 M(5k)并不蕴涵公式 Mem(10k)且无法找到其他蕴涵 Mem(10k)的状态在当前环境中成立,所以基本要求集 P 在当前环境中是不成立的。如果这个时候假设我们可以找到一个内存

释放进程,它可以把系统当前环境中状态 M(5k)调整为一个状态 M(15k), 且由 M(15k)成立可以得到 Mcm(10k)也成立，所以 P 在调整后的当前环境中成立，那么这个系统依据 P 是稳定的,反之，如果无法找到这样一个内存释放进程可以使得 P 成立，那么这个系统依据 P 就是不稳定的。

第5章 评估算法

5.1 概括描述

在这一部分当中，我们将提出一个评估算法，这个评估算法用于评估依据基本要求集，系统是否对于某个进程是稳定的。我们的评估过程是通过模拟该进程在系统环境中运行的状况来进行的。

根据上文中进程的定义，它是一个动作的有限序列,它的运行过程是动作一个一个依顺序的在系统当前环境中运行,所以我们对于进程的评估将细分为对于进程中每个动作的评估。如果在评估过程中,进程的当前被评估动作的评估结果是依据系统的基本要求集，系统对于该动作是不稳定的，那么将停止整个评估过程，且评估结果为系统对于该进程是不稳定的。反之，如果在评估过程中,对于进程中每一个动作的评估结果都是依据系统的基本要求集,系统对于该动作是稳定的，那么评估结果为系统对于该进程是稳定的。

例如：假设有一个被评估进程PS，PS为<A,B>,系统的当前环境为SE，基本要求集为P,P在SE中成立。首先，我们必须评估动作A：假设评估结果为系统对于动作A是稳定的，则继续评估该进程的下一个动作B，且这个时候通过我们的评估算法得到此时的当前环境为SE'。接着评估动作B:假设评估结果为系统对于动作B是稳定的，那么系统对于整个进程PS是稳定的，反之如果评估结果为依据P,系统对于动作B是不稳定的,则系统对于进程PS是不稳定的。

有了以上的评估过程，在这里,我们还需要解决另一个问题，那就是算法中的评估标准问题。也就是说，在算法中，如何去判断依据系统的基本要求集，系统对于该动作是否是稳定的。

在这里，我们的评估标准为：依据我们的稳定性定义，如果评估结果为依据系统 S 的基本要求集 P，在某个系统环境 SA 中，S 对于某动作是稳定的，则要么 P 在此动作运行完成后的系统环境 SA_1 中成立，要么能够在 SA_1 中找到一个进程 Ps，通过运行 Ps，可以把 SA_1 恢复到一个 P 成立的环境 SA_2 或者直接恢复到该动作运行前的环境 SA。

在这里，需要指出的是对不同的动作进行评估，由于其在进程中运行次序的不同，它们所处的环境也可能是不同的，所以我们的评估算法必须能够模拟进程中每一个动作运行完成后的系统环境。此外，由评估标准可以知道，评估算法也必须具有寻找恢复进程的能力，所以我们的评估算法主要由两个功能块组成：模拟动作的结果环境与寻找恢复进程。

5.2 系统知识集

我们的评估过程必须模拟进程在系统环境中运行的状况，而完成这样的模拟过程就必须拥有关于这个系统以及进程中所有动作的知识。

可以把这些知识大致分为四类：

1. 所有关于系统的无变元公式与状态关系的知识；具体的关系指的是在系统环境中，某些状态在环境中成立时，则相应的某公式在此环境中成立；

2. 动作执行的前提条件。也就是说如果这个动作要在环境中被正常运行，它需要哪些基本条件在环境中成立；

3. 动作本身的描述。进程是一个动作标识的序列，所以我们评估的过程中得到的往往只是一个个动作的标识，所以必须能够找到与此动作标识相对应的动作的内容，如一个动作映射 act：$\{\alpha_1 \Rightarrow \beta_1,\ \alpha_2 \Rightarrow \beta_2\}$，我们如果知道标识“act”，那么就应该可以得到它的前驱状态集$\{\alpha_1,\ \alpha_2\}$与后驱状态集$\{\beta_1,\ \beta_2\}$；

4．系统中所有自动动作的知识。在实际的计算机系统中，往往会有一些系统自动引发的动作，这种自动动作只要它的引发条件在环境中满足，那么就会自动的被系统运行。这些知识我们可以根据实际的操作系统或者硬件相关资料，如说明文档等得到。

如：操作系统 windows 2000 中有一个调用缓存的自动动作，它的引发条件为当系统拥有的物理内存小于目前的内存需要量，如果在某当前环境中此条件为真，则系统将在这个当前环境中调用此自动动作以获取缓存。

根据上述知识分类，现在可以定义一个知识集，它主要由上述四类知识组成：

定义 9: KS 是一个有穷的公式集合，称为系统知识集，其中主要存储了四类知识：

1）　$\Gamma(p) = \{Q|Q$ 为状态子集合$\}$，其中 $\Gamma(p)$ 表示一个函数，p 为系统的一个无变元公式，函数值为一个系统状态子集的集合$\{Q|Q$ 为状态子集合$\}$，它表示如果 Q 中所有状态在某环境中成立，则 p 成立；其中包括关于基本要求集的所有知识；

例如要推断关于系统的无变元公式 A 是否在系统环境 SA 中成立，由知识集中关于公式 A 成立的知识有 1.如果状态 B_1, B_2 都在环境中成立，则 A 在这个环境中成立，2. 如果 B_3 在环境中成立，则 A 也成立，所以有 $\Gamma(p) = \{\{B_1, B_2\}, \{B_3\}\}$，只要 $\{B_1, B_2\} \subseteq SA$ 或者 $\{B_3\} \subseteq SA$，我们就可以推断 A 在此环境 SA 中成立。

2）　$\Phi(act) = \{p_1, p_2, \cdots, p_n\}$，其中 $\Phi(act)$ 表示一个函数，act 为系统的一个动作标识，函数值为一个无变元公式集合 $\{p_1, p_2, \cdots, p_n\}$，它包括所有动作 act 运行需要的前提条件，$p_1, p_2, \cdots, p_n$ 为关于系统的无变元公式；

如：一个打印动作 print，它在某系统中运行的前提条件有两个，1.

打印机为空闲状态，2.内存空闲量：大于 50k。

3)　act 为系统的一个动作标识，Φ_1(act) 表示一个函数,函数值为 act 所标识的动作的前驱状态集，Φ_2(act) 表示一个函数,函数值为 act 所标识的动作的后驱状态集;

如:Φ_1(act) = $\{a_1, b_1\}$，Φ_2(act) = $\{a_2, b_2\}$，a_1, a_2, b_1, b_2为系统的状态，则 act 标识的动作的前驱状态集为$\{a_1, b_1\}$,后驱状态集为$\{a_2, b_2\}$。

4)　μ(autoAct) = P,其中 μ(autoAct) 表示一个函数，其中 autoAct 为一个自动动作,它属于自动动作列表 AutoActList,而自动动作列表我们可以由操作系统或硬件的说明文档中得到,函数值为关于系统的无变元公式集合 P,它描述了自动动作 autoAct 的引发条件,即当 P 在某环境中成立时,则 autoAct 将被系统自动运行。

在这里我们仅仅对评估算法的可行性进行理论上的讨论，所以可以假设这个知识集是完备的，也就是说它涵盖了所有我们需要的知识。

下面我们将定义三个新的概念:**可执行动作**,**动作模拟结果集**,**模拟结果环境**,下面是具体的定义:

定义 10　**可执行动作**：在系统 S 的环境 SA 中，一个动作标识为 act 的系统动作,如果其前提条件Φ(act)在 SA 中成立,则此动作在环境 SA 中被称为系统 S 的一个**可执行动作**。

定义 11　**动作模拟结果集**：指的是在某个系统环境 SA 中，某个动作运行完成后的新的系统环境，用 Conseq(SA, act)表示，act 为此系统动作的动作标识,其中:

$$Conseq(SA, act) = (SA - \Phi_1(act)) \cup \Phi_2(act) \quad (5-1)$$

定义 12　模拟结果环境：指的是在某个系统环境 SA 中，某个动作运行完成且系统完成其所有自动动作后的系统环境，用 SMSE(SA, act)表示，act 为此系统动作的动作标识，设<$act_1, act_2, \cdots, act_n$>为一个从知识集中得到的由动作 act 所诱发系统自动动作序列，那么有：

$$SMSE(SA, act) = Conseq_{n+1} \quad (5-2)$$，其中：

$Conseq_1 = Conseq(SA, act)$，

$Conseq_2 = Conseq(Conseq_1, act_1)$，

$Conseq_3 = Conseq(Conseq_2, act_2)$，

… … … … … … …

$Conseq_{n+1} = Conseq(Conseq_n, act_n)$。

5.3 寻找模拟结果环境算法

5.3.1 算法描述

算法 5-1

设当前的系统为 S

功能：寻找某个动作运行完毕后的模拟结果环境

输入：S 的当前环境 se，系统知识集 ks，当前待评估动作标识 act

输出：标识为 act 的动作的模拟结果环境 SMSE(se, act)或 ERROR，ERROR 表示无法得到正常的模拟结果环境

算法思路：

由模拟结果环境的定义可以发现如果要在一个环境 se 中求动作 act 的模拟结果环境，我们就必须找到 act 在 se 中诱发的自动动作序列，而寻找这个序列的过程可以描述为：首先求得 act 的动作模拟结果集 $Conseq_1$，然后根据知识集与 $Conseq_1$ 找到系统在环境 $Conseq_1$ 中诱发的自动动作 act_1，求出 act_1 在环境 $Conseq_1$ 中运行的动作模拟结果集 $Conseq_2$，不断重复上述过程，直到在某一个自动动作的

动作模拟结果集中系统不再诱发任何自动动作，那么这个动作模拟结果集就是act 的模拟结果环境。

算法步骤：

第 1 步

我们在知识集中搜寻待评估动作执行的前提条件，并验证此条件是否在当前环境中成立，如果成立则此动作为可执行动作，进行下一步，否则输出 ERROR。

第 2 步

我们根据知识集得到待评估动作的前驱状态集与后驱状态集，并且由此得到此动作的动作模拟结果集，并赋给环境变量 se'。并且在这里我们引入一个动作标识集合 asSet，它用于记录待评估动作及其所诱发的所有自动动作的标识。

第 3 步

扫描系统的自动动作列表 AutoActList，对其中每一个自动动作 autoAct 验证其引发条件在 se'中是否成立，如果 autoAct 的引发条件成立，则表明系统在环境 se'中将引发自动动作 autoAct。

接下来判断系统是否同时还会引发其他不同的自动动作，如果存在第二个不同的自动动作 $autoAct_i$ 被引发，那么由于两个动作的执行顺序是随机的，所以此时系统状态可能出现混乱，此时程序输出 ERROR;

接着还必须验证的是 autoAct 是否已经被引发过，即是否属于集合 asSet，如果属于则我们在这里推断系统有可能出现循环，程序输出 ERROR;

如果上述验证都通过，则表示此自动动作将在 se'中被正常自动运行，接着求出 autoAct 的动作模拟结果集，并赋给环境变量 se'，更新动作标识集合 asSet;

接着，重复第 3 步，重新扫描系统的自动动作列表，寻找下一个可能引发的自动动作。当扫描完自动动作列表，没有新的自动动作的引发条件在 se'中成

立时，我们就可以输出 se'，它就是最后的模拟结果环境。

5.3.2 算法实现（伪代码）

```
GETCON(se,act, ks):
BEGIN
/*验证此动作是否是可执行动作，不是则返回ERROR 值*/
for each fomula p in Φ(act) do
if (不存在一个 状态子集 Q,Q∈Γ(p)，使得 Q ⊆se) then
  return ERROR;
/*se' 为一环境变量,存储动作模拟结果集*/
se':=  (se - Φ1(act)) ∪ Φ2(act);//由公式 5-1
/* asSet 为被评估动作及其所诱发的自动动作的标识的集合*/
asSet := asSet ∪ {act};

/*扫描知识集中自动动作列表中的每一个动作*/
for every autoAct in AutoActList of ks do
begin
/*验证自动动作 autoAct 的引发条件在 se' 中是否成立 */
    if (μ(autoAct)中的每一个公式p都存在一个Q,有Q∈Γ(p)且Q⊆se') then
      begin
/*扫描自动动作列表,检验在 se'中是否同时存在第 2 个自动动作 autoAct1 被引发
*/
        for every autoAct1 in AutoActList of ks and autoAct1 != autoAct
do
          if (μ(autoAct1)中的每一个 p 都存在一个 Q,Q∈Γ(p)且Q⊆se')
          then return ERROR;//存在 2 个将被引发的自动动作,此时输出错误标识
/*检验 autoAct 是否已经被引发过,是则认为可能存在死循环,输出错误标识*/
```

```
      if autoAct∈asSet then
        return ERROR;
/*得到autoAct的动作模拟结果集,同时更新se'与asset*/
      se':=  (se' - ℂ1(autoAct)) ∪ ℂ2(autoAct);  //由公式5-1
      asset := asset ∪{ autoAct };
    end
end

/*返回最终模拟结果环境*/
return se'
END
```

5.3.3 算法流程图

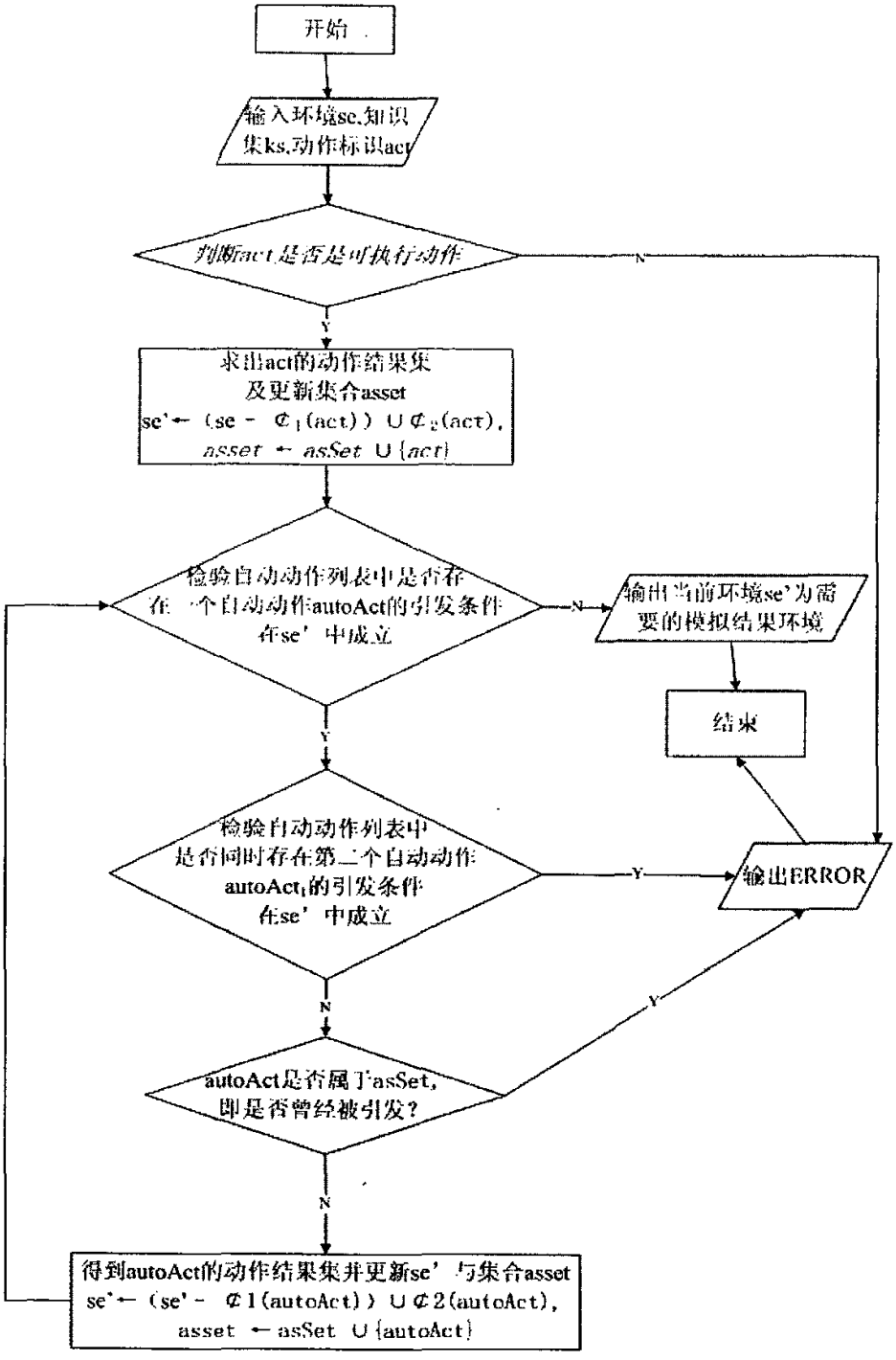

图 5 - 1

5.4 寻找恢复进程算法

5.4.1 算法目的

定义 13： 系统 S 中有被评估动作 ACT 的模拟结果环境 SMSE(SE,ACT)，以及基本要求集 P, P 在 SMSE(SE,ACT)不成立；假设存在一个进程 rps，通过运行 rps 能够把 SMSE(SE,ACT)要么恢复到 P 成立的另一个环境 SE_1,要么直接恢复到环境 SE，则 rps 就被称为关于被评估动作 ACT 与模拟结果环境 SMSE(SE,ACT)的一个恢复进程;恢复进程中的动作称为恢复动作。

通过算法 5-1，现在已经可以找到某个动作 act 在初始环境 se 下执行所得到的模拟结果环境 SMSE(se, act)，由我们的评估标准可知，当基本要求集在模拟结果环境 SMSE(se, act)中不成立的情况下：

如果能在环境 SMSE(se, act)中找到一个恢复进程，那么评估结果为系统对于动作 act 是稳定的；

反之，如果在当前环境 SMSE(SE, act)中不能够找到一个恢复进程，那么评估结果为该系统对于动作 act 是不稳定的。

5.4.2 算法描述

算法 5-2

设当前的系统为 S

输入:S 的当前环境 se，系统知识集 ks，当前待评估动作标识 act，基本要求集 P，$P=\{p_1,p_2,\ldots,p_m\}$

输出:如果可以找到恢复进程，则输出此恢复进程，否则输出 null 值

功能:寻找关于动作 act 与模拟结果环境 SMSE(SE, act)的恢复进程

算法思路：

两类环境差异集：其中类型 1 的环境差异集表示在恢复动作的模拟结果环境中成立而在被评估动作的运行初始环境中不成立的所有状态的集合，类型 2 的环境差异集表示在被评估动作的运行的初始环境中成立而在恢复动作的模拟结果环境中不成立的所有状态的集合。它们的初始值为被评估动作的运行初始环境与它的模拟结果环境的差异集。

算法的过程就是不断的寻找恢复动作缩小环境差异集。

定义一个用于存储当前恢复路径的栈。它的栈元素中包括恢复动作的标识，该恢复动作执行前的环境，以及类型 1 与类型 2 的环境差异集；栈的操作包括栈顶元素出栈 pop，元素进栈 push，以及输出栈中所有动作标识序列的操作 getActSeq，输出序列的顺序为从栈底元素的动作标识到栈顶元素的动作标识；

当找到一个动作，它可以把类型 1 环境差异集中的状态更新为类型 2 中的状态，换句话说，该动作运行后新的模拟结果环境与被评估动作运行的初始环境的差异集为原来差异集的真子集时，这就是我们要找的恢复动作，并把它压入栈中，接着更新环境差异集与当前环境；

当基本要求集在当前模拟结果环境中不成立且差异集不为空，同时再找不到新的恢复动作时表明当前的恢复路径是失败的，弹出栈中最后一个动作，恢复所有差异集与环境为该恢复动作未执行前状态，选择另一条路径进行尝试，直到要么基本要求集在模拟结果环境中成立，要么差异集为空，要么尝试了所有路径都是失败路径。

因为我们的环境差异集中元素个数是有穷的，同时，限定每个恢复动作只能使环境差异集中元素个数减少，且不会有新的元素加入集合中，所以可尝试的恢复路径的条数也是有穷的，算法一定可以在有穷步内得到结果。

算法步骤：

第 1 步

首先，定义一个动作标识序列集合 FailSeqSet，它用于记录所有已经证明失败的恢复路径；

接着，调用算法 GETCON 得到被评估动作的模拟结果环境，赋给一个环境变量 conSe，并求出这个时候的两类环境差异集 cxSet 与 sxSet 的初值，同时把环境 conSe 赋给环境变量 se_1。

第 2 步

寻找一个恢复动作，它的前驱状态集包含于 cxSet，后驱状态集包含于 sxSet，具体过程如下：

扫描集合 cxSet 的所有子集，并提取出其中一个子集 frmCxe，标记该子集为已扫描；

搜索知识集 ks，找到一个动作，它前驱状态集为 frmCxe，后驱状态集为 sxSet 中的一个子集 frmSxe，如果搜索不到这样的动作，则提取 cxSet 中下一个子集，同样标记其为已扫描；如果搜索到一个满足需要的动作 z，这个时候就需要对 z 进行一系列的验证：

验证当前的恢复路径在增加动作标识 z 后是否属于集合 FailSeqSet，如果属于则表明此路径是已经尝试过的失败的路径；

调用算法 GETCON 求出 z 的模拟结果环境，并赋值给环境变量 se_2，如果 se_2 为 ERROR 则说明此动作在环境中无法正常得到模拟结果环境；

验证模拟结果环境 se_2 是否曾在当前恢复路径中出现过，如果是则表示过程中出现了相同的中间环境，恢复过程有可能陷入循环；

验证运行 z 后新的环境差异集 $cxSet_1$ 与 $sxSet_1$（$cxSet_1 := se_2 - se$，$sxSet_1 := se - se2$）是否分别是原环境差异集的真子集，是则说明 z 是合理的恢复动作。

如果动作 z 通过上述验证，则 z 就是当前路径所需要的恢复动作，把动

作标识 z，环境变量 se_1，差异集 cxSet,sxSet 压入栈中，接着更新集合 sSet，环境差异集 cxSet,sxSet，以及环境变量 se_1；最后，验证基本要求集是否在 se_1 中成立或者 cxSet 是否为空，成立或者为空则输出当前恢复路径为需要的结果；反之，寻找下一个恢复动作。

如果 z 未通过验证，则提取 cxSet 中下一个标记为未扫描的状态子集，寻找另一个可能的恢复动作；如果此时，所有的状态子集都标记为已扫描，则需要验证栈是否为空：如果不为空，则表明我们在当前路径下已无法找到下一个恢复动作，记录此失败路径，路径回退，弹出栈顶元素，寻找下一个可能的方向，且 cxSet 中所有状态子集重新标记为未扫描；如果栈为空，则表明已经尝试了所有可能的路径，无法找到恢复进程，输出 null 值。

5.4.3 算法实现（伪代码）

栈元素数据结构：

```
ACTS
Begin
    Action Signal actSN;
    Environment Set Se;
    Environment Set cxSet;
    Environment Set sxSet;
end
```

其中，*actSN* 为恢复动作的标识，*Se* 为动作 actSN 执行的当前环境，*cxSet* 为动作 *actSN* 执行前的类型 1 的环境差异集，*sxSet* 为动作 *actSN* 执行前的类型 2 的环境差异集。

栈的操作：

Actstack

begin

ACTS　pop();

push(Action Signal actSN, Environment Set Se, cxSet. sxSet);

Action Signal Sequence getActSeq();

End

GETSEQ(se, act, ks, P)

Actstack　L;//*L 是一个栈*

Act Sequence FailSeqSet; //*FailSeqSet 是一个动作标识序列集合,用于记录已失败的恢复路径*

BEGIN

/**由算法 GETCON(se, act, ks)得到模拟结果环境*/*

conSe := GETCON(se, act, ks) ;

/**模拟结果环境赋给一个环境集合变量*/*

se_1 := conSe;

/**求出差异集*/*

cxSet := conSe – se;

sxSet := se – conSe;

/**扫描 cxSet 中所有状态子集,并提取一个子集 frmCxe，标记其为已扫描 */*

for each state formula subset frmCxe **of** cxSet **do**

begin

/**搜索知识集,寻找一个动作 z */*

if((∉1(z) = frmCxe)and(∉2(z) = frmSxe)and frmSxe⊆sxSet) then

begin

/**求动作 z 的模拟结果环境*/*

```
        se2 := GETCON(se1, z, ks) ;
/*验证动作 z */
        if((L.getActSeq()*z ∉ FailSeqSet) and (se2!=ERROR)and(se2 ∉ sSet))
        then
          begin
/*求得假设运行 z 后新的差异集,并分别赋给变量 cxSet1 与 sxSet1*/
            cxSet1:= se2 – se;
            sxSet1:= se – se2;
/*验证z 运行完毕,新的差异集是否是原差异集的真子集,是则z 是有效的恢复动作
*/
            if(cxSet1 ⊂ cxSet) and (sxSet1 ⊂ sxSet) then
            begin
/*更新集合 sSet*/
              sSet := sSet ∪ { se2} ;
/*把新的栈元素入栈*/
              L.push(z, se1,cxSet, sxSet);
/*执行动作 z 后新的当前环境与差异集*/
              cxSet := cxSet1;
              sxSet := sxSet1;
              se1 := se2 ;
/*判断基本要求集是否已满足，满足则返回当前路径为恢复进程*/
              if (基本要求集 P 中每一个公式 p 都存在一个 Q,Q∈Γ(p) 且 Q
                 ⊆ se1) then
              return L.getActSeq();
/*当 cxSet 为空则恢复完毕，输出恢复路径*/
              if(cxSet 为空) then
              return L.getActSeq();
            end
          end
        end
```

```
/*当 L 不为空且 cxSet 已扫描完毕时,表明当前恢复路径已经无法找到下一个恢复
动作,所有变量值回到路径中最后一个恢复动作执行前的状态,重新扫描 cxSet*/
    if ((cxSet 中所有子状态集都扫描完毕) and L != null) then
    begin
/*记录当前恢复路径为失败路径*/
      FailSeqSet := FailSeqSet∪{L.getActSeq()};
/*栈顶元素出栈*/
      actStack := L.pop();
/*恢复所有变量到栈顶元素未执行前状态以便重新选择动作执行路径*/
      cxSet := actStack.cSet;
      sxSet := actStack.sSet;
      se1 := actStack.Se;
      sSet := sSet – { se1}
    end
end
/*当 L 为空且 cxSet 不为空时,则表明无法找到恢复进程*/
if (L = null and cxSet 不为空) then
return null
else
/*返回恢复进程*/
return L.getActSeq();
END
```

5.4.4 算法流程图

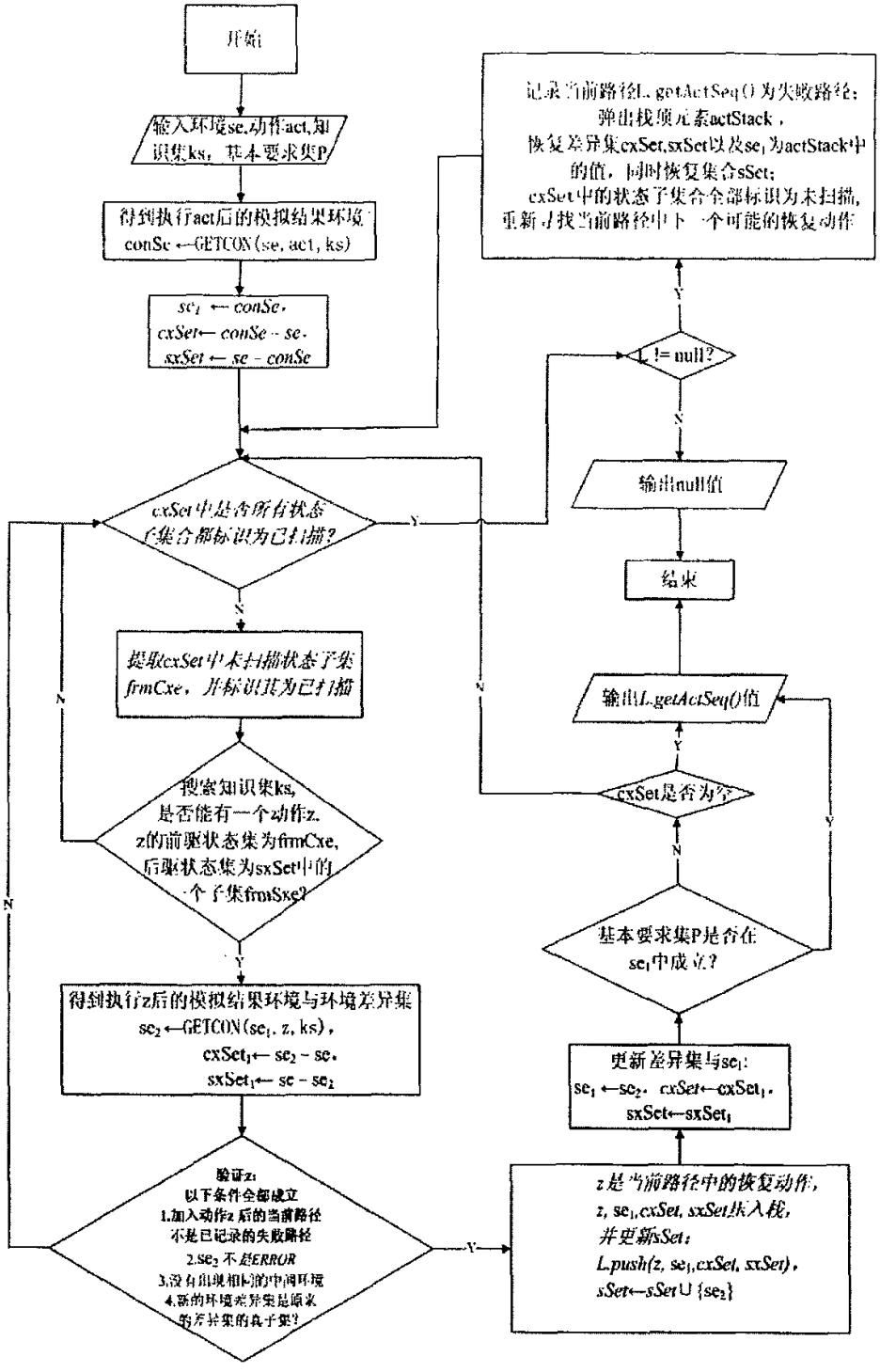

图 5 - 2

5.5 完整的进程评估算法

由算法 5-1 与算法 5-2，现在我们已经可以评估依据基本要求集，系统对单个动作是否稳定，下面我们将由这两个算法得到系统对于一个进程是否稳定的完整的评估算法。

5.5.1 算法功能与步骤描述

算法 5-3

设当前的系统为 S

输入：S 的当前环境 se，系统知识集 ks，当前待评估进程 PS，基本要求集为

$P = \{p_1, p_2, \ldots, p_m\}$

输出：如果评估结果为稳定的则输出值 ISSTABLE，否则输出 NOTSTABLE

功能：评估依据基本要求集 P，系统 S 对于进程 PS 是否是稳定的

算法步骤：

第 1 步

把当前环境 se 赋给环境变量 se_1；

第 2 步

依进程运行顺序对 P 中每一个动作 act 进行评估，如果基本要求集 P 在 act 运行后的模拟结果环境 se_2 中不成立且无法找到恢复进程，则返回 NOTSTABLE 值；如果基本要求集 P 在 se_2 中成立或者可以找到恢复进程，则继续评估进程中下一个动作；

第 3 步

如果对所有动作的评估结果都是稳定的，则返回 ISSTABLE 值。

5.5.2 算法实现(伪代码)

```
StbCheck(se, PS, ks, P)
begin
/*把当前环境赋给环境变量 se1*/
  se1 := se;
/*依进程运行顺序取 PS 中动作 act 进行评估*/
  for each act in PS do
  begin
    se2 := GETCON(se1 ,act, ks);
    PISTRUE := true;// 布尔变量 PISTRUE 用于标识 P 是否在 se2 中成立
/*如果该动作无法得到正常的模拟结果环境,无法完成评估，则认为是不稳定的*/
    if (se2 = ERROR) then
      return NOTSTABLE;
/*检验基本要求集 P 在动作执行后的模拟结果环境中是否成立,不成立则
PISTRUE 赋值为 false*/
   for i = 1 to m do
   begin
     if (不存在一个 Q,使得 Q∈Γ(pi)且 Q ⊆se) then
     begin
          PISTRUE := false;
         break;//退出循环
     end
   end

/*如果基本要求集在动作执行后的环境中不成立,则须继续检验此动作执行后是
否有恢复进程*/
   if (! PISTRUE) then
```

```
    begin
/*由算法 GETSEQ 得到动作 act 运行后的恢复进程*/
        rsmPs := GETSEQ(se1 ,act,ks, P);
/*如果无法找到恢复进程,则返回值 NOTSTABLE,评估结果为不稳定*/
        if rsmPs = null then
        return NOTSTABLE;
    end
/*通过以上所有检验,更新当前环境,评估下一动作*/
        se1 := se2;
  end
/*所有动作都评估为稳定的,则对此进程的评估结果为稳定的*/
  return ISSTABLE;
end
```

5.6 评估结论与定义的一致性

5.6.1 提出定理

在文章的上一部分构造了我们的评估算法，它可以用于评估依据系统的基本要求集，系统是否对于某个进程是稳定的。下面我们提出一个关于算法的评估结论与稳定性定义的一致性的定理，并且将证明该定理。

定理 2: 有一个系统 S, SE 为 S 的当前环境，被评估进程为 ps，关于 S 的知识集 KS，基本要求集 P，$P = \{P_1, P_2, \cdots, P_n\}$，且 P 在 SE 中成立。如果由评估算法 5-3 得到的评估结果为 ISSTABLE，那么依据 P，系统 S 对于此进程 ps 是稳定的。

5.6.2 定理证明

证明：

假设 ps 为 $\langle act_1, act_2, \cdots, act_M \rangle$，我们的评估算法的评估结论为 ISSTABLE。

假设 k 表示评估算法对进程$<act_1, act_2, \cdots, act_k>$进行评估得到的结果是 ISSTABLE，k 小于或等于 M。

1' 当 k = 1 时

假设这个时候 act_1 的模拟结果环境为 SE_1，由评估算法 5 - 3 知，**情况 1**． P 在 SE_1 中成立；**情况 2**．P 在 SE_1 中不成立，我们可以找到关于 act_1 与 SE_1 的一个恢复进程 rps_1，把 SE_1 恢复到 SE；**情况 3**. P 在 SE_1 中不成立，我们可以找到关于 act_1 与 SE_1 的一个恢复进程 rps_2，把 SE_1 恢复到 P 成立的一个环境 SE'．

情况 1：P 在 SE_1 与 SE 中都成立，由稳定性定义有，依据 P，系统 S 对于进程$< act_1>$是稳定的。

情况 2：下面首先证明环境 SE_1 收敛到 P

因为有一个恢复进程 rps_1，它可以把 SE_1 恢复到 SE，且 SE_1 在且仅在自身中是成立的，同时有 P 在 SE 中成立，所以由收敛性定义，SE_1 收敛到 P。

由 SE_1 收敛到 P 及稳定性定义，有依据 P，系统 S 对于进程$< act_1>$是稳定的。

情况 3：与情况 2 同理，可以证明环境 SE_1 收敛到 P，所以由稳定性定义，有依据 P，系统 S 对于进程$< act_1>$是稳定的。

2' 假设当 k = n - 1 时，定理 2 成立：

也就是说如果由算法 5-3 对于动作序列〈$act_1, act_2, \cdots, act_{n-1}$〉得到的评估结果为 ISSTABLE，那么那么依据 P，系统 S 对于此进程是稳定的。假设 act_{n-1} 的模拟结果环境为 SE_{n-1}，由收敛性定义知，此时存在两种情况：

情况 1：P 在 SE_{n-1} 中成立；

情况 2：P 在 SE_{n-1} 中不成立同时 SE_{n-1} 收敛到 P。

那么当 k = n 时

假设这个时候动作 act_n 的模拟结果环境为 SE_n，由评估算法知，如果算法评估动作 act_n 的评估结果为 ISSTABLE，则存在三种情况：

情况Ⅰ： P 在 SE_n 中成立；

情况Ⅱ： P 在 SE_n 中不成立，我们可以找到一个恢复进程 rps_3，把 SE_n 恢复到 SE_{n-1}；

情况Ⅲ： P 在 SE_n 中不成立，我们可以找到一个恢复进程 rps_4，把 SE_n 恢复到另一个环境 SE_{n-1}' 且 P 在 SE_{n-1}' 中成立。

所以有六种情况必须证明：

1）在情况Ⅰ，1 下有，P 在 SE，SE_{n-1}，SE_n 中都成立：

故由稳定性定义有，依据 *P*，系统 *S* 对于进程〈$act_1, \cdots, act_n$〉是稳定的。

2）在情况Ⅰ，2 下有，P 在 SE，SE_n 中都成立：

故由稳定性定义有，可以得到依据 *P*，系统 *S* 对于进程〈$act_1, \cdots, act_n$〉是稳定的。

3）在情况Ⅱ，1 下，P 在 SE 与 SE_{n-1} 中成立且 P 在 SE_n 中不成立，同时我们可以找到一个恢复进程 rps_3，把 SE_n 恢复到 SE_{n-1}：

因为有一个恢复进程 rps_3，它可以把 SE_n 恢复到 SE_{n-1} 且 P 在 SE_{n-1} 中成立，同时 SE_n 在且仅在自身中成立，由收敛性定义，可得 SE_n 收敛到 P；

故由稳定性定义，依据 P, 系统 S 对于进程$< act_1, \cdots, act_n >$是稳定的。

4）在情况 II，2 下，P 在 SE 中成立且 P 在 SE_n 与 SE_{n-1} 中不成立，我们可以找到一个恢复进程 rps_3, 把 SE_n 恢复到 SE_{n-1}:

因为有一个恢复进程 rps_3，它可以把 SE_n 恢复到 SE_{n-1}, 且 SE_n 与 SE_{n-1} 在且仅在自身中成立，所以由收敛性定义，SE_n 收敛到 SE_{n-1}.

又由情况 2 中有 SE_{n-1} 收敛到 P，由定理 1 得 SE_n 收敛到 P, 故由稳定性定义，依据 P, 系统 S 对于进程$< act_1, \cdots, act_n >$是稳定的。

5）在情况III，1 下，P 在 SE 与 SE_{n-1} 中成立且 P 在 SE_n 中不成立，我们可以找到一个恢复进程 rps_4, 把 SE_n 恢复到 SE_{n-1}' 且 P 在 SE_{n-1}' 中成立：

证明法与 3）基本相同，可得 SE_n 收敛到 P；

故由稳定性定义，依据 P, 系统 S 对于进程$< act_1, \cdots, act_n >$是稳定的。

6）在情况III，2 下，P 在 SE 中成立且 P 在 SE_n 与 SE_{n-1} 中不成立，同时，我们可以找到一个恢复进程 rps_4, 把 SE_n 恢复到 SE_{n-1}' 且 P 在 SE_{n-1}' 中成立：

因为有一个恢复进程 rps_4，它可以把 SE_n 恢复到 SE_{n-1}'，P 在 SE_{n-1}' 中是成立的且 SE_n 在且仅在自身中是成立的，所以由收敛性定义，SE_n 收敛到 P；

故由稳定性定义，依据 P, 系统 S 对于进程$< act_1, \cdots, act_n >$是稳定的。

由 1),2),3)，4)，5)，6) 评估算法对于进程$< act_1, \cdots, act_n >$都有定理 2 成立；

综合上述证明，由 1'，2'评估算法对于进程 ps 的任意前缀进程都有定理 2 成立，所以由评估算法对于进程 ps,定理 2 成立.

故定理得证。

第 6 章 评估过程示例

下面将通过我们的评估算法来评估一个进程，判断系统对其是否是稳定的。

假设我们现在有一个计算机系统 S，S = { CPU(x):CPU 空闲 x*100%，Mem(x):空闲内存量为 xk, valueX(x):变量 X 的值为 x，PRINTER(x):x 为 0 表示打印机空闲，x 为 1 表示已被占用}；

系统 S 的基本要求集为 P, P = {NeedCPU(0.2):CPU 空闲量至少 20%, NeedMem(30):空闲内存量至少 30k}；

S 有系统知识集 KS, S 的当前环境为 SE = {CPU(0.8)，Mem(100)，valueX(0), PRINTER(0)}；

待评估进程 PS, PS = <ACTa, ACTb, ACTc>. 下面我们将演示评估算法评估进程 PS 的过程。

6.1 评估过程描述

第 1 步. 评估动作 ACTa:运行环境为 SE

由知识集 KS 有:

前驱状态集 $\mathcal{C}_1$(ACTa) = { CPU(0.8)，Mem(100)，valueX(0)}，后驱状态集 $\mathcal{C}_2$(ACTa) = { CPU(0.5)，Mem(80)，valueX(1)}；

ACTa 执行的前提条件 Φ(ACTa) = { P_1(0.1):CPU 空闲量至少 10%, P_2(10):空闲内存量至少 10k }, 有

$\Gamma(P_1(0.1))$ 中存在一个状态子集{ CPU(0.8)}且{ CPU(0.8)} $\subseteq$SE;

$\Gamma(P_2(10))$ 中存在一个状态子集{ Mem(100)} 且 Mem(100) $\subseteq$SE;

所以, 动作 ACTa 在当前环境中是可执行动作;

由定义求出动作 ACTa 的动作模拟结果集:

$$Conseq(SE, ACTa) = (SE - \mathcal{C}_1(ACTa)) \cup \mathcal{C}_2(ACTa)$$
$$= \{CPU(0.5),\ Mem(80),\ valueX(1),\ PRINTER(0)\}$$

又在知识集 ks 中没有自动动作的引发条件在 Conseq(SE, ACTa)中成立,所以这个时候系统没有引发自动调整动作,所以由定义求得模拟结果环境:

$$SMSE(SE, ACTa) = Conseq(SE, ACTa)$$
$$= \{CPU(0.5),\ Mem(80),\ valueX(1),\ PRINTER(0)\}.$$

同时知识集有:

Γ(NeedCPU(0.2)) 中存在一个状态子集{ CPU(0.5)}且{ CPU(0.5)} $\subseteq$ SMSE(SE, ACTa);

Γ(NeedMem(30)) 中存在一个状态子集{ Mem(80)} 且 Mem(80) $\subseteq$ SMSE(SE, ACTa) 。

所以, 基本要求集 P 在当前环境 SMSE(SE, ACTa)中成立的,故对动作 ACTa 的评估结果为稳定的。

第 2 步. 评估动作 ACTb:运行环境为 SMSE(SE, ACTa)

由知识集 KS 有:

前驱状态集 $\mathcal{C}_1$(ACTb) = { CPU(0.5), Mem(80) ,PRINTER(0)}, 后驱状态集 $\mathcal{C}_2$(ACTb) = { CPU(0.15), Mem(20) ,PRINTER(1)};

ACTb 执行的前提条件Φ(ACTb) = { P_1(0.4):CPU 空闲量至少 40%, P_2(60):空闲内存量至少 60k },有

$\Gamma(P_1(0.4))$ 中存在一个状态子集{ CPU(0.5)} 且{ CPU(0.5)} $\subseteq$ SMSE(SE, ACTa);

$\Gamma(P_2(60))$ 中存在一个状态子集{ Mem(80)} 且 Mem(80) $\subseteq$ SMSE(SE, ACTa);

所以,动作 ACTb 在当前环境中是可执行动作;

由定义求出动作 ACTb 的动作模拟结果集:

$$Conseq(SMSE(SE, ACTa), ACTb) = (SMSE(SE, ACTa) - \mathcal{C}_1(ACTb)) \cup \mathcal{C}_2(ACTb)$$
$$= \{CPU(0.15), Mem(20), valueX(1), PRINTER(1)\}$$

又知识集的自动动作列表可以找到一个自动动作 SETFREE,它的引发条件 $\{PRINTER(1)\} \subseteq Conseq(SMSE(SE, ACTa), ACTb)$,所以这个时候系统会引发自动调整动作 SETFREE,且 $\mathcal{C}_1(SETFREE) = \{PRINTER(1)\}$, $\mathcal{C}_2(SETFREE) = \{PRINTER(0)\}$,所以 SETFREE 的动作模拟结果集为:

$$Conseq(Conseq(SMSE(SE, ACTa), ACTb), SETFREE)$$
$$= (Conseq(SMSE(SE, ACTa), ACTb) - \mathcal{C}_1(SETFREE)) \cup \mathcal{C}_2(SETFREE)$$
$$= \{CPU(0.15), Mem(20), valueX(1), PRINTER(0)\}.$$

这时，不存在其它自动动作的引发条件在 Conseq(Conseq(SMSE(SE, ACTa), ACTb), SETFREE)中成立，所以由模拟结果环境定义有:

$$SMSE(SMSE(SE, ACTa), ACTb)$$
$$= Conseq(Conseq(SMSE(SE, ACTa), ACTb), SETFREE)$$
$$= \{CPU(0.15), Mem(20), valueX(1), PRINTER(0)\}$$

同时知识集 ks 中有:

$\Gamma(NeedCPU(0.2))$中所有状态子集都不包含于 SMSE(SMSE(SE, ACTa), ACTb);

$\Gamma(NeedMem(30))$ 中所有状态子集都不包含于 SMSE(SMSE(SE, ACTa), ACTb);

所以，基本要求集 P 在当前环境 SMSE(SMSE(SE, ACTa), ACTb)中不成立,故必须验证是否能够找到恢复进程,它可以把 SMSE(SMSE(SE, ACTa), ACTb)恢复到 SMSE(SE, ACTa) 。

下面首先求出两种类型的差异集分别为{CPU(0.15), Mem(20)}, {CPU(0.5), Mem(80)},假设通过扫描差异集以及系统知识集,我们得到唯一的动作 SETRES,有 $\mathcal{C}_1(SETRES) = \{CPU(0.15), Mem(20)\}$, 后驱状态集 $\mathcal{C}_2(SETRES)$

= { CPU(0.5), Mem(80)};

同时,前提条件 Φ(SETRES) = { P_1(0.1):CPU 空闲量至少 10%, P_2(10):空闲内存量至少 10k },有

Γ(P_1(0.1)) 中存在一个状态子集 {CPU(0.15)} 且 {CPU(0.15)} ⊆SMSE(SMSE(SE, ACTa), ACTb);

Γ(P_2(10)) 中存在一个状态子集 { Mem(20)} 且 {Mem(20)} ⊆SMSE(SMSE(SE, ACTa), ACTb);

所以动作 SETRES 在当前环境中是可执行动作,且由知识集有环境 SMSE(SE, ACTa)不会引发其他自动动作,所以由定义:

SMSE(SMSE(SMSE(SE, ACTa), ACTb), SETRES)

=Conseq(SMSE(SMSE(SE, ACTa), ACTb), SETRES)

={CPU(0.5), Mem(80) , valueX(1), PRINTER(0)} = SMSE(SE, ACTa);

故得到恢复进程<SETRES>,所以对 ACTb 的评估结果为稳定的。

第 3 步.评估动作 ACTc:运行环境为 SMSE(SMSE(SE, ACTa), ACTb)

由知识集 KS 有:

前驱状态集 $\mathcal{C}_1$(ACTc) = { CPU(0.15), Mem(20) , valueX(1)}, 后驱状态集 $\mathcal{C}_2$(ACTc) = { CPU(0.08), Mem(9) , valueX(0)};

ACTc 执行的前提条件Φ(ACTc) = { P_1(0.1):CPU 空闲量至少 10%, P_2(5):空闲内存量至少 5k },有

Γ(P_1(0.1)) 中存在一个状态子集 { CPU(0.15)} 且 {CPU(0.15)} ⊆SMSE(SMSE(SE, ACTa), ACTb);

Γ(P_2(5)) 中存在一个状态子集 { Mem(20)} 且 {Mem(20)} ⊆SMSE(SMSE(SE, ACTa), ACTb);

所以,动作 ACTc 在当前环境中是可执行动作;

由定义求出动作 ACTc 的动作模拟结果集:

Conseq(SMSE(SMSE(SE, ACTa), ACTb), ACTc)

= (SMSE(SMSE(SE, ACTa), ACTb) - $\mathcal{C}_1$(ACTc)) ∪ $\mathcal{C}_2$(ACTc)

= {CPU(0.08), Mem(9) , valueX(0), PRINTER(0)}

又在知识集 ks 中没有自动动作的引发条件在 Conseq(SMSE(SMSE(SE, ACTa), ACTb), ACTc) 中成立，所以这个时候系统没有引发自动调整动作，所以由定义求得模拟结果环境：

SMSE(SMSE(SMSE(SE, ACTa), ACTb), ACTc)

= Conseq(SMSE(SMSE(SE, ACTa), ACTb), ACTc)

= {CPU(0.08), Mem(9) , valueX(0), PRINTER(0)}

同时对基本要求集 P 有：

Γ(NeedCPU(0.2)) 中所有状态子集都不包含于 SMSE(SMSE(SMSE(SE, ACTa), ACTb), ACTc)；

Γ(NeedMem(30)) 中所有状态子集都不包含于 SMSE(SMSE(SMSE(SE, ACTa), ACTb), ACTc)；

所以，基本要求集 P 在当前环境 SMSE(SMSE(SMSE(SE, ACTa), ACTb), ACTc) 中不成立，故必须验证是否能够找到恢复进程，它可以把 SMSE(SMSE(SMSE(SE, ACTa), ACTb), ACTc) 恢复到 SMSE(SMSE(SE, ACTa), ACTb).

下面首先求出两种类型的差异集分别为 {CPU(0.08), Mem(9), valueX(0)}, {CPU(0.15), Mem(20), valueX(1)}，假设通过扫描两种差异集以及系统动作集合，我们得到唯一的释放资源的动作 SETRES，有 $\mathcal{C}_1$(SETRES) = {CPU(0.08), Mem(9)}，后驱状态集 $\mathcal{C}_2$(SETRES) = {CPU(0.15), Mem(20)}；

同时，前提条件 Φ(SETRES) = {P_1(0.1):CPU 空闲量至少 10%, P_2(10):空闲内存量至少 10k}，有

Γ(P_1(0.1)) 中所有状态子集都不包含于 SMSE(SMSE(SMSE(SE, ACTa), ACTb), ACTc)；

Γ(P_2(10)) 中所有状态子集都不包含于

SMSE(SMSE(SMSE(SE, ACTa), ACTb), ACTc);

所以动作 SETRES 在当前环境中不是可执行动作，且我们找不到其他的释放资源的动作可以恢复环境，所以无法得到恢复进程，对动作 ACTc 的评估结果为不稳定的。

所以，最终评估算法对进程 PS 的评估结果为 NOSTABLE，不稳定的。

由于操作系统没有足够的资源运行资源调度进程而造成系统资源无法释放，进而造成系统无法响应用户要求，这是使用电脑过程中最常见的导致系统崩溃的原因。

第 7 章 总结及展望

7.1 总结

本文基于一阶谓词逻辑演算系统对计算机的系统，状态，系统环境，动作以及进程进行了形式化的定义，并且描述了系统的基本要求集。同时在这个形式化系统之上，提出了依据基本要求集，我们的基于进程的系统稳定性定义。

依据形式化系统与稳定性定义，本文提出了一个基于知识的进程评估算法，它可以用于评估单个进程对于稳定性的影响。这个算法主要包括：

1. 知识集的定义。在我们的知识集中主要包括四类知识：公式与状态关系的知识，动作本身的知识，动作的执行前提条件，以及系统中的自动动作的知识。
2. 模拟动作运行的结果环境的算法。这个算法能够依据系统的当前环境，通过知识集模拟出系统中某动作执行完毕后的环境，我们称之为模拟结果环境。
3. 寻找恢复进程的算法。这个算法主要用于当某动作的模拟结果环境不满足基本要求集的情况下，寻找到一个恢复进程，该进程可以把该模拟结果环境要么恢复到动作运行的初始环境，要么更新到满足基本要求集的某一个环境。

提出了一个关于评估算法的评估结论相对于我们的稳定性定义的一致性的定理，并证明了该定理的正确性。文章的最后,演示了一个进程的具体评估过程。

7.2 研究展望

本文根据我们系统关于进程的稳定性定义，提出了一个算法去评估单个进程对于系统稳定性的影响，其中对于这个算法我们只进行了理论上可行性的探讨，并没有详细考虑复杂度的问题，在今后的文章中，可以进一步考虑其算法优化的问题。在这里,有一个初步的想法就是可以借鉴多级稳定性[8]中对系统中所有动作进行分级的方式,这样就可以大大减少算法的复杂度。

此外，由于研究的周期比较短，所以目前算法只能针对系统环境中只有单个进程的情况进行评估，在今后的研究中，我们的工作应该进一步考虑系统环境中同时存在多进程时，如何对进程进行更有效的评估。

参考文献

1. E.W.DIJKSTRA. Self-stabilization systems in spite of distributed control. Communications if the ACM, 1974 , 17(11), 643-644.

2. M.S.ABADIR, M.G.GOUDA. The stabilizing computer. Proceedings of the international conference on parallel and distributed systems, 1992.

3. ANISH.ARORA. stabilization. http: //www. cse. ohio-state. Edu / siefast / group / publications / stb.ps

4. A.ARORA, M.G.GOUDA. Closure and Convergence: A Foundation for Fault-Tolerant Computing. IEEE Trans. Software Eng., Mar,1993, vol.19, no.3, 1015-1027.

5. A.BUI, A.K.DATTA, F.PETIT, V.VILLAIN. State-Optimal Snap-stabilizing PIF in Tree Networks. Proc. Third Workshop self-stabilizing systems, 1999, 78-85.

6. S.GHOSH, A.GUPTA, T.HERMAN, S.V.PEMMARAJU. Fault-Containing Self-Stabilizing Algorithms. Proc.ACM Symp.Principles of Distributed Computing, 1996, 45-54.

7. M.SCHNEIDER. Self-stabilization. ACM Computing Surveys, 1993, 25(1), 45-67.

8. M.G.GOUDA. Multiphase Stabilization. IEEE Transactions on Software

Engineering, 2002, vol.28,No.2, 201-208.

9. D.LEHMAN, M.RABIN. On the advantages of free choice: A symmetric and fully distributed solution of the dining philosophers problem. Proceeding of the 8th Annual ACM Symposium on Principles of Programming Languages, New York, 1981, 133-138.

10. J.E.BURNS, M.G.GOUDA, R.E.MILLER. Stabilization and pseudo-stabilization. Tech.Rep.TR-90-13, Dept. of Computer Sciences, Univ. of Texas at Austin, 1990.

11. M.G.GOUDA. The Theory of Weak Stabilization. WSS 2001, LNCS 2194, 2001, 114-123.

12. JORG.KIENZLE. Software Fault Tolerance: An Overview. Ada-Europe 2003, LNCS 2655, 2003,45-67.

13. 李娜. 现代逻辑的方法. 河南大学出版社, 1997.

14. 宋文?金 符号逻辑基础. 北京师范大学, 1993.

15. 刘素姣. 一阶逻辑在人工智能中的应用. 河南大学硕士学位论文，2004.

16. M.G.GOUDA. The triumph and tribulation of system stabilization. Invited Lecture, Proceedings of 9th International Workshop on Distributed Algorithms, Springer-Verlag:972, 1995, 1-18.

17. T.HERMAN. Self-stabilization bibliography: Access guide. http: // www. cs. Uchicago.edu/cgi-bin/cjtcs/get/working/papers/1.html,1996.

18. F.F.HADDIX. Alternating Parallelism and the stabilization of Cellular Systems.

PhD dissertation, Dept. of Computer Sciences, Univ. of Texas at Austin, 1999.

19. S.GHOSH. Understanding self-stabilization in distributed systems. Tech.Rep.TR-90-02, Dept. of Computer Science. Univ. of Iowa.

20. B.AWERBUCH, B.PATT, G.VARGHESE. Self-stabilization by local checking and correction. Proceedings of the 32nd IEEE Symposium on Foundations of Computer Science, 1991.

21. N.S.CHEN, P.F.YU, T.S.HUANG. A self-stabilizing algorithm for constructing spanning trees. Inf.Process.Lett.,39,1991,147-151.

22. RAY.GIGUETE, JOHNETTE HASSELL. Designing a resourceful fault-tolerance system. ELSEVIER, The Journal of Systems and Software, 62, 2002, 47-57.

23. A.Arora, M.G.Gouda. Delay-Insensitive Stabilization. Proceedings of the Third Workshop on Self-Stabilizing Systems, 1997, 95–109.

24. S.Dolev. Self-Stabilization. 1st edn. MIT Press, Cambridge Massachusetts, 2000.

致　谢

首先感谢我的导师周青副教授，本论文所有工作都是在周老师的鼓励支持和精心指导下完成的。在论文的选题、研究和撰写过程中，周老师都给予了充分的指导、支持，在我遇到困难的时候，周老师总是给予热情的鼓励和有益的启示。周老师渊博的专业知识、一丝不苟的治学态度、诲人不倦的精神都将使学生铭记终身。在此，请周老师接受我衷心的感谢和崇高的敬意！

感谢李磊教授，姜云飞教授，倪德明副教授，邓克像老师，他们在软件所中努力营造的良好学术氛围和学习环境使我获益非浅，而他们严谨和勇于创新的治学态度，更是我今后学习工作的榜样。

最后感谢在研究生两年里一起学习，生活的同学们，他们是罗伟，程丽明，黄寿锋，王立坤，胡若旻，周海军等，他们的鼓励和帮助使我克服了学习和生活中遇到的各种各样的困难。我们一起生活和学习的快乐时光会给我留下永远的回忆。

原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含其他个人或集体中已经发表或撰写过的作品成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：王斌强

日期：2006年5月15日