doi:10.3969/j.issn.1672-1292.2018.04.007

一种基于测试的消息竞争故障定位方法

曾　奕

(南京师范大学计算机科学与技术学院,江苏 南京 210023)

[摘要]　针对并发程序中由消息竞争所引发的不确定性故障难以捕获与再现、定位结果不精确等问题,提出一种结合程序频域比对、Delta 调试策略的故障定位方法. 该方法首先建立并发程序执行模型,定义同步序列描述程序执行路径的不确定性,并将其抽象成简洁的消息竞争序列;而后收集、比对正确的测试执行与错误的测试执行中消息竞争序列间的差异,约简故障搜索空间,获得初始故障集合;最后,为初始故障集合中所对应的每一个失效执行,采用 Delta 调试策略设计并运行一组附加测试,从而逐步精准地锁定故障根源. 初步实验结果表明,该方法能够有效检测消息竞争故障,提高故障定位效率.

[关键词]　并发,不确定性,错误定位,消息竞争

[中图分类号]TP311　[文献标志码]A　[文章编号]1672-1292(2018)04-0044-08

Fault Localization Approach-Based Testing for Message Races

Zeng Yi

(School of Computer Science and Technology, Nanjing Normal University, Nanjing 210023, China)

Abstract: A fault localization method, which combines the program spectrum with the Delta debugging strategy, is proposed to solve the problem on the non-deterministic failure caused by the message race which is difficult to capture, reproduce, and accurately locate. This method first builds a concurrent program execution model, defines the non-deterministic execution path with the synchronous sequence, and turns it into a concise message race sequence. Secondly, it collects and compares the difference of the message race sequence between the correct test execution and the error execution, reduces the fault search space, and obtains the initial fault set. In the end, for every failure in the initial fault set, the Delta debug strategy is used to design and run a set of additional tests to locate the root of the fault gradually. Preliminary experimental results show the effectiveness and feasibility of our method in locating non-deterministic failures for message passing programs.

Key words: concurrent, non-deterministic, fault localization, message race

随着用户对软件系统性能要求的提高,以及多处理机与多核体系结构的流行,并发软件的开发和使用日益广泛. 并发程序可以充分利用系统资源,灵活性强. 然而,大量复杂的通讯和同步使得软件设计者在软件设计时易出现疏漏,而细小疏漏所引发的不确定性错误隐藏在多个并发单元的交互中,往往难以捕捉和定位,甚至造成严重的后果. 因此,在并发程序调试中,定位因不确定性所引发的程序失效是调试人员的首要任务[1-2].

基于测试的错误定位技术是软件调试的有效手段,与传统的算法调试策略、程序切片分析以及以模型为基础的故障诊断方法相比,具有自动化程度高、结果清晰明确、通用性好等优势. 目前,其研究主要分为几个方面:(1)基于频域(覆盖)的错误定位技术,将程序的动态执行轨迹抽象成特定的频域,通过提取、分析、比对频域间差异,确定出可能包含错误的程序范围[3-7];(2)Delta Debugging 技术,采用分而治之的策略辨别测试输入间差异,逐步约简搜索空间,定位出造成软件失效的输入片段[8-11];(3)基于统计的错误定位技术,利用机器学习、聚类分析、回归统计等方法分析、定位与错误的发生具有高相关性的谓词[12-14].

然而,上述研究工作主要集中在顺序程序领域,尚未对并发多进程、多线程程序提供足够支持,现有的

收稿日期:2018-06-15.

通讯联系人:曾奕,讲师,研究方向:软件测试、软件调试. E-mail:05277@ njnu.edu.cn

频域类型(语句覆盖频域、路径频域、分支频域、不变式频域等)并不能够精确地描述并发活动的交互行为. 此外,虽然 Choi 和 Zeller[1] 曾运用 Delta Debugging 技术隔离造成软件失效的线程调度,但其未能明确指出错误所对应的程序行为,并不能真正将程序员从繁重的调试工作中解脱出来.

为有效定位消息传递程序中因消息间不当竞争所导致的不确定性错误,本文提出了一种结合程序频域比对、Delta 调试策略的故障定位方法. 为此,首先建立简单的并发程序执行模型以描述消息竞争所引发的程序执行路径不确定性,从程序的测试执行信息中获取同步序列,计算消息竞争序列;然后,利用程序正确的测试执行过滤错误执行中符合程序员意图的有益竞争,初步圈定故障的排查范围;最后,从程序的错误执行出发,围绕初步的故障推断,迭代生成并运行一组新的附加测试,寻找错误出现规律,逐步锁定错误.

1　不确定性分析模型

基于测试的错误定位技术通过分析、比对待调试程序在给定测试用例集上的执行信息,确定出故障的排查范围. 在实际应用中,研究人员首先需要依据给定的调试目标,选取合适的程序频域表示测试执行信息. 对于基于消息传递的并发程序,由于进程调度和消息延时等原因引发消息间竞争,使得即使在同一输入下,程序的不同执行其消息传递的顺序不尽相同,从而导致执行路径的不确定性[15]. 定位此类由消息间不当竞争所引发的不确定性故障,首先需要从多个进程的交互执行轨迹中判别出与不确定性行为相关的消息竞争状态.

Xu 和 Lei 等[11] 定义同步事件序列(简称同步序列)以表示并发程序的交互执行. 本文在此基础上分析引发不确定性的竞争状态,并将待调试程序在给定测试用例集下的执行信息抽象成简单、清晰的竞争消息发送事件序列.

1.1　相关定义

设并发程序 CP 包含 n 个独立的并发单元 P_1、P_2、…、P_n,记为 $\mathrm{CP}=\{P_1,P_2,\cdots,P_n\}$,在给定的测试输入 X 下,进行 m 次测试 $t_1,t_2,\cdots,t_m$. 将消息传递相关的消息发送、消息接收操作称为同步事件,进程 P_j 中的第 k 个同步事件记为 e_{jk},e.type 指明事件类型为发送(send)或接收(receive)操作;e.partner 表明发送(接收)操作相对应的消息接收(发送)进程.

定义 1　CP 的同步事件集合记为 E,E_j 为进程 P_j 所包含的同步事件集合,

$$E=\sum_{j=1}^{n}E_j,E_j=\{e_{j_1},e_{j_2},\cdots,e_{j_{a_j}}\},|E_j|=a_j. \tag{1}$$

定义 2　进程 P_j 在测试执行 t_i 中所有同步事件的执行序列定义为同步序列 $\mathrm{syn}_{i,j}$,

$$\mathrm{syn}_{i,j}=e_{i,j_1},e_{i,j_2},\cdots,e_{i,j_l},\quad e_{i,j_k}\in E_j. \tag{2}$$

定义 3　在给定的测试执行下,同步事件间的执行先后关系 $<_t$,$e_{i,j}$ 和 $e_{k,l}$ 满足 $e_{i,j}<_t e_{k,l}$,当且仅当以下 3 个条件之一成立:

$$\begin{aligned}&(1)\,i=k\wedge j<l,\\&(2)\,i\neq k\wedge e_{i,j}.\mathrm{type}=\mathrm{send}\wedge e_{k,l}.\mathrm{type}=\mathrm{receive}\wedge e_{i,j}.\mathrm{partner}=k,\\&(3)\,e_{i,j}<_t e_{h,g}\wedge e_{h,g}<_t e_{k,l}.\end{aligned} \tag{3}$$

待调试程序 CP 在给定的测试输入 X 下,进行 m 次测试 $t_1,t_2,\cdots,t_m$,记录各进程的同步序列,可构建描述并发程序中消息传递执行轨迹的频域,记为 $m*n$ 的二维执行矩阵 $\boldsymbol{E}(\mathrm{syn}_{i,j})$. 显然,$\boldsymbol{E}(\mathrm{syn}_{i,j})$ 记录了 CP 在给定测试用例集上的所有同步事件的时间序列集合,其中,$\mathrm{syn}_{i,j}$ 为测试执行 t_i 中进程 P_j 的同步序列.

1.2　消息竞争序列

为便于分析消息间的竞争关系,分别使用 s、t 表示消息的发送、接收事件,每个接收事件 r 匹配其相对应的发送事件 s,将 $\boldsymbol{E}(\mathrm{syn}_{i,j})$ 转换成更直观的消息序列矩阵 $\boldsymbol{M}(\mathrm{msgseq}_{i,j})$.

定义 4　测试执行 t_i 中,进程 P_j 有 a_j 个消息接收操作,其消息序列 $\mathrm{msgseq}_{i,j}$ 定义如下:

$$\mathrm{msgseq}_{i,j}=\begin{pmatrix} r_{j_1} & r_{j_2} & \cdots & r_{j_{a_j}} \\ \vdots & \vdots & \cdots & \vdots \\ s_{l_1} & s_{l_2} & \cdots & s_{l_{a_m}} \end{pmatrix}. \tag{4}$$

并发程序执行中,由进程调度和消息延时等原因引发消息间竞争,导致执行不确定性的本质条件是当且仅当一组同时在传递的消息未明确其先后顺序,则其中任何一个都有可能首先到达而被接收[15].

定义 5　$\mathrm{msgeq}_{i,j}$中消息接收事件 $r_{j,k}$、$r_{j,h}$,所对应的消息 msg_1、msg_2 互为竞争消息,当且仅当以下 3 个条件均成立:

$$\begin{aligned} &(1)\,k<h,\\ &(2)\,r_{j,k}.\mathrm{partner}=s_{l_i}\wedge\ r_{j,h}.\mathrm{partner}=s_{l_{i'}},\\ &(3)\,r_{j,k}\not<_t s_{l_{i'}}. \end{aligned} \tag{5}$$

定义 6　竞争消息 msg_1、msg_2 引发消息竞争故障,当且仅当以下 3 个条件均成立:

$$\begin{aligned} &(1)\,k<h,\\ &(2)\,r_{j,k}.\mathrm{partner}=s_{l_i}\wedge\ r_{j,h}.\mathrm{partner}=s_{l_{i'}},\\ &(3)\,r_{j,k}、s_{l_{i'}}\text{应满足 }r_{j,k}<_t s_{l_{i'}}\text{且故障程序中 }r_{j,k}\not<_t s_{l_{i'}}. \end{aligned} \tag{6}$$

定义 7　测试执行 t_i 中,进程 P_j 的消息序列 $\mathrm{msgseq}_{i,j}$中有 k 个消息互为竞争消息,其消息竞争序列 $\mathrm{rmsgseq}_{i,j}$定义如下:

$$\mathrm{rmsgseq}_{i,j}=\begin{pmatrix} r_{j_{l_1}} & r_{j_{l_2}} & \cdots & r_{j_{l_k}} \\ \vdots & \vdots & \cdots & \vdots \\ s_{l_1} & s_{l_2} & \cdots & s_{l_k} \end{pmatrix}. \tag{7}$$

显然,消息竞争序列矩阵 $\boldsymbol{RM}$ 能够清晰地刻画出并发程序中由消息间竞争所引发的并不确定性行为. 其中,横向量 $\boldsymbol{RM}_i=(\mathrm{rmsgseq}_{i,1},\mathrm{rmsgseq}_{i,2},\cdots,\mathrm{rmsgseq}_{i,n})$ 表示并发程序 CP 在测试执行 t_i 中各个进程的消息竞争序列集,即 CP 在一次测试执行中的竞争消息的执行序列;列向量 $\boldsymbol{RM}_j=(\mathrm{rmsgseq}_{1,j},\mathrm{rmsgseq}_{2,j},\cdots,\mathrm{rmsgseq}_{m,j})^{\mathrm{T}}$ 则表明进程 P_j 在 m 次测试执行中的消息竞争序列集,即单个进程在测试用例集 T_s 下竞争消息的执行序列,其 m 个元素的不同取值恰好反应出该进程的不确定性行为.

进一步分析 $\boldsymbol{RM}_j$,其具有以下性质:

性质 1　消息竞争序列 $\mathrm{rmsgseq}_{i,j}$是一个 l_k 元置换.

证明　进程 P_j 的消息接收事件的顺序是由程序的逻辑结构所决定的,即 $r_{j,l_1},r_{j,l_2},\cdots,r_{j,l_k}$是恒定的. 设 $\mathrm{rmsgseq}_{ij}$中 h 个消息 $\mathrm{msg}_{l_i},\mathrm{msg}_{l_{i+1}},\cdots,\mathrm{msg}_{l_{i+h}}(h\leqslant k)$,是一组相互竞争的消息,则根据定义 5 可知,该序列中第一个接收操作 r_{j,l_i},对于 $\forall s\in\{s_{l_{i+1}},s_{l_{i+2}},\cdots,s_{l_{i+h}}\}$ 均满足 $r_{j,l_i}\not<_t s$,则 $\{s_{l_{i+1}},s_{l_{i+2}},\cdots,s_{l_{i+h}}\}$ 中任何发送操作均有可能在执行中先到达,而匹配接收操作 r_{j,l_i},不妨设 s'被 r_{j,l_i}所接收. 同理,序列中的第二个接收操作 $r_{j,l_{i+1}}$则可匹配$\{s_{l_i},s_{l_{i+1}},\cdots,s_{l_{i+h}}\}-\{s'\}$ 中的任意发送操作. 以此类推,可得 $\mathrm{rmsgseq}_{i,j}$是 l_k 个消息发送事件对恒定的消息接收事件的一一变换.

性质 2　消息竞争序列 $\mathrm{rmsgseq}_{i,j}$可以分解成若干个相互具有竞争关系的发送事件子序列的乘积:

$$\mathrm{rmsgseq}_{i,j}=(s_{l_1},s_{l_1},\cdots,s_{l_i})(s_{l_{i+1}},s_{l_{i+2}},\cdots,s_{l_{i+d}})\cdots(s_{l_h},s_{l_{h+1}},\cdots,s_{l_k}). \tag{8}$$

证明　由轮换分解定理,每一置换可唯一表示为若干个不相交轮换的乘积.

利用性质 1、性质 2,省略恒定的接收事件序列,可将矩阵 $\boldsymbol{RM}$ 中的 $\mathrm{rmsgseq}_{ij}$分解成若干个互不相交的发送事件子序列,分别对应相互竞争的消息接收顺序,并为每个子序列赋以唯一的标识,使用轮换形式表示.

定义 8　待调试并发程序 CP,执行测试用例集 T_s,执行矩阵 $\boldsymbol{E}(\mathrm{syn}_{i,j})$ 中所有引发不确定行为的竞争消息的发送事件序列记为 SendSeq_Set,其定义如下:

$$\mathrm{SendSeq\_Set}(\mathrm{sendseq}_{i,j})=\sum_{i=1}^{m}\sum_{j=1}^{m}\mathrm{rmsgseq}_{i,j}\{(s_{l_1},s_{l_1},\cdots,s_{l_i}),(s_{l_{i+1}},s_{l_{i+2}},\cdots,s_{l_{i+d}}),\cdots,(s_{l_h},s_{l_{h+1}},\cdots,s_{l_k})\},$$

$$|\mathrm{SendSeq\_Set}|=\sum_{i=1}^{m}\sum_{j=1}^{n}\mathrm{rmsgseq}_{i,j}. \tag{9}$$

定义 8 将待调试并发程序 CP 在测试用例集 T_s 上的不确定性行为抽象成简单、清晰的消息发送事件序列集合,为后续的故障定位奠定基础.

2　基于测试的故障定位策略

当待测并发系统 CP 在测试过程中出现不确定性错误,即运行失败或同一输入下运行结果不相同时,若能够隔离出失效执行中与故障相关的消息竞争序列,便可为故障的定位与修复提供有益的线索.

2.1　计算初始故障集合

在给定的测试用例集下,揭示出不确定故障的测试执行在检测出软件错误的同时也说明其执行路径中的某些竞争序列便是诱发故障的原因;而那些运行通过的测试执行中所包含的大量正确的竞争序列体现出设计人员正确的设计意图,同样能够为故障的定位提供很多有价值的信息. 因此,一个简单而有效的方法便是比对程序测试执行间的差异,过滤失效执行中符合设计人员意图的竞争序列,从而获得初步的故障排查集合.

假设待测并发程序 CP,使用测试用例集 T_s 进行测试. 具体流程如下:

(1)使用向量时间戳[16]记录各进程中同步事件的逻辑时钟,获得程序执行矩阵 $\boldsymbol{E}(\mathrm{syn}_{i,j})$;

(2)根据测试结果将测试用例集 T_s 划分为两个子集 T_{s1} 和 T_{s2},分别包含运行失败和运行通过的测试执行,计算相应的竞争消息发送事件序列集合 $\mathrm{SendSeq\_Set}_1$、$\mathrm{SendSeq\_Set}_2$;

(3)利用集合 $\mathrm{SendSeq\_Set}_2$ 过滤 $\mathrm{SendSeq\_Set}_1$,从 $\mathrm{SendSeq\_Set}_1$ 中删除不可能导致软件错误的序列,生成初始错误集合:$\mathrm{Fault\_Set}=\mathrm{SendSeq\_Set}_1-\mathrm{SendSeq\_Set}_2$.

在步骤(1)中,使用向量时间戳标识同步事件的逻辑时钟,记录同步事件间的偏序关系. 为每个进程设置一个 n 维的本地向量时间戳 $\boldsymbol{VC}_{pi}=[VC_{pi}^1,\cdots,VC_{pi}^j,\cdots,VC_{pi}^n]$,其中每一个分量均为整数,分别对应于系统中的每一个进程. 在程序的任何执行点,进程 P_i 中本地向量时钟 $\boldsymbol{VC}_{pi}$ 的第 j 个分量的值 VC_{pi}^j 始终对应进程 P_j 中发生于 P_i 中当前同步事件的上一个同步事件. 具体实现分为两个部分:

①进程 P_i 的发送操作 s 之前:计算 $\boldsymbol{VC}(s)=\boldsymbol{VC}_{pi}+\boldsymbol{\Delta}_i$($\boldsymbol{\Delta}_i$ 是一个 n 维向量,第 i 个分量的值为 1,其余分量值为 0),将 $\boldsymbol{VC}(s)$ 的值赋给 $\boldsymbol{VC}_{pi}$,并随消息 msg 发送;

②进程 P_j 的接收操作 r 之后:计算 $\boldsymbol{VC}(r)=\mathrm{Max}(\boldsymbol{VC}_{pi},\boldsymbol{VC}_{\mathrm{msg}})+\boldsymbol{\Delta}_i$(Max 函数求两个向量时间戳每个对应分量的最大值,$\boldsymbol{VC}_{\mathrm{msg}}$是通过消息 msg 传递过来的发送操作的时间戳),将 $\boldsymbol{VC}(r)$ 的值赋给 $\boldsymbol{VC}_{pi}$. 向量时间戳记录了同步事件间的偏序关系$<_t$,即给定任意两个独立的同步事件 $e_{i,j}$、$e_{k,l}$ 满足:$e_{i,j}<_t e_{k,l}\Leftrightarrow\boldsymbol{VC}(e_{i,j})\neq\boldsymbol{VC}(e_{k,l})\wedge\exists m\in[1,n]:\boldsymbol{VC}(e_{i,j})[1,n]:\boldsymbol{VC}(e_{i,j})[m]<\boldsymbol{VC}(e_{k,l})[m]$.

步骤(2)判断同步事件间的$<_t$关系,使用算法 1 计算程序执行矩阵 $\boldsymbol{E}(\mathrm{syn}_{i,j})$ 中竞争消息的发送事件序列,并根据测试结果正确与否,划分成两个集合 $\mathrm{SendSeq\_Set}_1$、$\mathrm{SendSeq\_Set}_2$. 其中,$\mathrm{SendSeq\_Set}_1$ 包含错误测试执行中引发不确定性的发送事件序列,即所有引发故障的序列均应包含在此集合中;集合 $\mathrm{SendSeq\_Set}_2$ 中包含正确执行中引发不确定性的发送事件序列,即符合程序员设计意图的竞争序列集合.

步骤(3)利用未发现软件错误的集合 $\mathrm{SendSeq\_Set}_2$ 来过滤集合 $\mathrm{SendSeq\_Set}_1$,利用算法 2 从 $\mathrm{SendSeq\_Set}_1$ 中删除不可能导致软件错误的消息竞争序列.

算法 1　计算竞争消息发送事件序列集合

输入:$\boldsymbol{E}(\mathrm{syn}_{i,j})$

输出:$\mathrm{SendSeq\_Set}_1$、$\mathrm{SendSeq\_Set}_2$

for each $\mathrm{syn}_{i,j}\in\boldsymbol{E}$ do

　while($k\leqslant a_j$) do//进程 P_j 有 a_j 个消息接收事件

　$l=k$;

　while($l<=a_i$ and $r_{j,k}\not<_t r_{j,l}$.parnter) l++;//判别消息竞争

　if($t_i\in T_{s1}$) then

　$\mathrm{SendSeq\_Set}_1=\mathrm{SendSeq\_Set}_1+\{(r_{j,k}.\mathrm{parnter}\sim r_{j,l}.\mathrm{parnter})\}$;

　else

　$\mathrm{SendSeq\_Set}_2=\mathrm{SendSeq\_Set}_2+\{(r_{j,k}.\mathrm{parnter}\sim r_{j,l}.\mathrm{parnter})\}$;

end if
$k=l+1$;
end while
end for

算法 2　过滤算法

输入:SendSeq_Set_1、SendSeq_Set_2
输出:Fault_Set
$\text{Fault\_Set}=\text{SendSeq\_Set}_1$;
for each sendseq in SendSeq_Set_1 do
if sendseq in SendSeq_Set_2 then
$\text{SendSeq\_Set}_1=\text{SendSeq\_Set}_1-\{\text{sendseq}\}$;
end if
end for

2.2　Delta 调试策略定位故障

初始的故障集合为调试人员提供了一个较小的、可供人工筛选的范围. 然而,不确定性错误的调试是一个非常复杂的过程:一方面,不确定性错误具有偶发性、不可再现性的特征,故障的触发可能仅与某次特定的错误执行相关,因而后续的定位工作应围绕揭示错误的测试执行展开;另一方面,系统中存在的故障个数是未知的,每一条错误执行中所包含的故障个数也是未知的,为了进一步提高效率,可以假设错误的测试执行中包含多个故障,采用渐增式方法逐次生成附加的测试执行.

基于以上两点,本文采用 Delta 调试策略定位故障. 为 Fault_Set 中的各个元素构造其候选值 alter;然后以每一条运行失败的测试执行为原型,逐次替换可能引发错误的消息发送序列,生成一组附加测试,借助确定性重演机制迭代执行,直至观察到执行结果的改变,从而直接定位出导致不确定性故障的消息竞争序列.

在实际的测试过程中,有多种候选值的构造方法,本文仍利用前面已有的正确的测试执行信息. 不妨设 Fault_Set 中共有 N 个元素,每个元素记为 σ_i,设$|\sigma_i|=f_i$,由置换的性质可知 σ_i 共有$(f_i!-1)$个同阶置换,记为集合 $D(\sigma_i)$. 如果存在 $\bar{\sigma}_i \in \text{SendSeq\_Set}_2 \cap D(\sigma_i)$,则 $\text{alter}(\sigma_i)=\{\bar{\sigma}_i\}$;否则,$\text{alter}(\sigma_i)=D(\sigma_i)-\{\sigma_i\}$.

考虑测试用例集中包含多个错误执行,且每一条错误执行中包含多个故障的情形,不妨设 T_{sl} 中包含 l 个错误执行,$T_{sl}=\{t_1,t_2,\cdots,t_l\}$,$t_i$ 中可能引发故障的消息发送序列为初始故障集合 Fault_Set 在测试执行 t_i 上的投影,记为 $F(t_i)$,其中有 $x_i(1\leqslant x_i\leqslant|F(t_i)|)$个序列是引发系统故障的原因. 调试中,只有将这 x_i 个错误序列同时替换成正确值时,程序的执行结果才能由错误变为正确. 由于并不能事先预见 x_i 的具体取值,因此采用渐增式迭代的策略,从 $1\sim x_i$ 逐次增加替换个数,依次替换$|F(t_i)|$中的 j 个序列.

算法 3 给出了这一故障定位方法的详细流程. 构造集合 Fault_Set 中的各个元素的候选值集合 alter,然后针对 T_{sl} 中的每一个错误执行,定位引发不确定性故障的消息竞争序列.

算法 3　Delta 调试故障定位算法

输入:Fault_Set,//初始故障集合
　　$\boldsymbol{E}$ 在 T_{s1} 上的投影//揭示不确定性故障的错误执行轨迹
输出:V_{err}//引发不确定性错误的发送事件序列
while($i<=|\text{Fault\_Set}|$)do
if($\bar{\sigma}_i \in \text{SendSeq\_Set}_2 \cap D(\sigma_i)$)then//构造候选值集合
$\text{alter}(\sigma_i)=\bar{\sigma}_i$;
else
$\text{alter}(\sigma_i)=D(\sigma_i)$;
end if
end while

for each $t_i \in T_{sl}$ do
　while($j \leqslant |F(t_i)|$) do
　$j=1$
　依次从 $F(t_i)$ 中选取 j 个 σ_i,获得待替换序列 τ_k,及其对应的 $|\text{alter}(\sigma_{i1})| \times |\text{alter}(\sigma_{i2})| \times \cdots \times |\text{alter}(\sigma_{ij})|$ 个替换值;
　　for each 替换值 τ' do
　使用 τ' 替换 t_i 中的 τ_k,执行附加测试;
　if 执行结果变得正确 then　//找到错误根源
　　输出 $V_{\text{err}}=\tau_k$;
　　goto 8;　//继续定位其他错误执行
　end if
　　end for
　end for
　$j=j+1$;
　end while
end for

3　实例与实验分析

3.1　实例分析

图 1 所示的错误代码片段中,有 4 个并发进程 P_1、P_2、P_3 和 P_4;func_send 与 func_receive 分别为消息的发送与接收函数;msg_i 抽象地代表发送的消息;ANY 表示接收操作可以接收任意进程的消息(为描述方便,此处使用一维下标标识消息和同步事件). 代码片段中的两个故障分别为:(1)进程 P_2 中,未限定接收事件 r_7、r_8 应先后依次接收进程 P_1 发送的消息 msg_7、进程 P_3 发送的消息 msg_8;(2)进程 P_3 中,未限定接收事件 r_{10}、r_{11} 应先后依次接收进程 P_2 发送的消息 msg_{10}、进程 P_4 发送的消息 msg_{11}.

Process P_1	Process P_2	Process P_3	Process P_4
……	……	……	……
func_send(P_2,msg_2); //s_2	func_receive ANY; //r_1	func_send(P_2,msg_3); //s_3	func_send(P_2,msg_1); //s_1
……	func_receive ANY; //r_2	……	……
func_receive(P_2); //r_5	func_receive ANY; //r_3	func_receive ANY; //r_4	func_send(P_3,msg_4); //s_4
……	……	func_receive ANY; //r_6	……
func_send(P_2,msg_7); //s_7	func_send(P_1,msg_5); //s_5	……	func_receive(P_3); //r_9
……	……	func_send(P_2,msg_8); //s_8	……
func_send(P_3,msg_6); //s_6	……	func_send(P_3,msg_{11}); //s_{11}	
……	func_send(P_4,msg_9); //s_9	……	
func_receive ANY; //r_7	……		
func_receive ANY; //r_8	func_receive ANY; //r_{10}		
……	func_receive ANY; //r_{11}		
func_send(P_3,msg_{10}); //s_{10}	……		
……			

图 1　错误代码片段

Fig. 1　Error code snippet

图 2 表示该代码片段在同一测试输入下的 3 次测试执行 t_1、t_2 和 t_3,其中,$T_{s1}=\{t_3\}$. 测试执行中消息发送与接收同步事件的向量时间戳如表 1 所示,使用算法 1 判别出引发不确定性行为的竞争消息执行序列为:$\text{SendSeq\_Set}_1=\{(s_2,s_3,s_1),(s_6,s_4),(s_8,s_7),(s_{11},s_{10})\}$、$\text{SendSeq\_Set}_2=\{(s_1,s_2,s_3),(s_3,s_1,s_2),(s_4,s_6),(s_6,s_4),(s_7,s_8),(s_{10},s_{11})\}$,则初始故障集合为 $\text{Fault\_Set}=F(t_3)=\{(s_2,s_3,s_1),(s_8,s_7),(s_{11},s_{10})\}$. 采用算法 3 计算 $F(t_3)$ 中 3 个元素的候选值:$\text{alter}(\sigma_1)=\{(s_1,s_2,s_3)/(s_3,s_1,s_2)\}$、$\text{alter}(\sigma_2)=\{(s_8,s_7)\}$、$\text{alter}(\sigma_3)=\{(s_{10},s_{11})\}$;利用渐增式迭代的策略,按照 σ_1、σ_2、σ_3、$\sigma_1\sigma_2$、$\sigma_1\sigma_3$、$\sigma_2\sigma_3$、$\sigma_1\sigma_2\sigma_3$ 的顺序依次替换初始的错误执行 t_3,执行新的附加测试,直到同时将 $\sigma_2\sigma_3$ 使用正确的候选值替换,程序的执行结果由错误变为正确,输出引发程序不确定性错误的消息竞争序列 (s_8,s_7) 和 (s_{11},s_{10}).

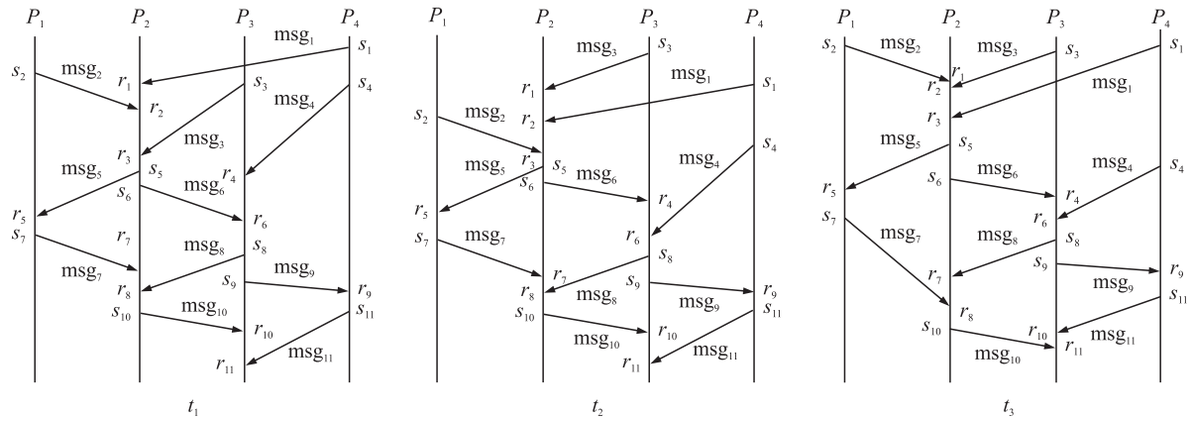

图 2　测试执行片段

Fig. 2　Test execution fragment

3.2　实验分析

为进一步验证本文方法的有效性,选取 3 个轻量级 java 程序,分别注入单个、2 个消息竞争故障,其基本信息如表 2 所示. 实验程序使用 MPJ Express 并行消息传递库实现线程间异步通信. 对待测程序的消息传递语句进行源代码插装,以获得给定测试集合下的执行结果及执行信息. 故障定位相关信息描述如表 3 所示.

本文采用定位到消息竞争故障所执行的附加测试的个数与消息竞争序列个数的比值这一错误定位代价指标对错误定位的效率进行评价,比值越小表示故障定位的效率越高. 从表 2 可以看出,对于轻量级的单故障程序,本文的故障定位效率约在 15%~20%左右,而对于多故障程序,本方法依然有效,具有约 30%的效率.

表 1　同步事件 s、r 的向量时间戳

Table 1　Vector timestamp of synchronization event s and r

s	$t_1/t_2/t_3$	r	t_1	t_2	t_3
s_1	[0,0,0,1]	r_1	[0,1,0,1]	[0,1,1,0]	[1,1,0,0]
s_2	[1,0,0,0]	r_2	[1,2,0,1]	[0,2,1,1]	[1,2,1,0]
s_3	[0,0,1,0]	r_3	[1,3,1,1]	[1,3,1,1]	[1,3,1,1]
s_4	[0,0,0,2]	r_4	[0,0,2,2]	[1,5,2,1]	[1,5,2,1]
s_5	[1,4,1,1]	r_5	[2,4,1,1]	[2,4,4,1]	[2,4,1,1]
s_6	[1,5,1,1]	r_6	[1,5,3,2]	[1,5,3,2]	[1,5,3,2]
s_7	[3,4,1,1]	r_7	[3,6,1,1]	[3,6,1,1]	[1,6,4,2]
s_8	[1,5,4,2]	r_8	[3,7,4,2]	[3,7,4,2]	[3,7,4,2]
s_9	[1,5,5,2]	r_9	[1,5,5,3]	[1,5,5,3]	[1,5,5,3]
s_{10}	[3,8,4,2]	r_{10}	[3,8,6,2]	[3,8,6,2]	[1,5,6,4]
s_{11}	[1,5,5,4]	r_{11}	[3,8,7,4]	[3,8,7,4]	[3,8,7,4]

表 2　实验对象描述

Table 2　Experimental object description

程序名称	程序说明	线程数量	消息事件	故障个数
Pi	求圆周率	3	9	1
Sort	排序	4	18	1
MatrixMul	矩阵相乘	4	26	2

表 3　故障定位相关信息

Table 3　Fault localization related information

程序名称	消息竞争序列个数	初始故障集合大小	执行附加测试个数	定位故障个数	故障定位代价比值
Pi	26	3	4	1	15.38%
Sort	54	8	11	1	20.37%
MatrixMul	87	14	27	2	31.03%

4　结语

本文针对进程间消息传递为主要通信方式的并发程序,采用一种基于测试的方法,收集、抽象、比对程序正确执行与错误执行之间的差异,将错误确定在较小的排查范围内;并依据初步的故障推断,以程序的错误执行为原型,生成并运行新的测试以逐步锁定故障根源. 初步的实验表明,方法能够精准地定位故障,提高定位效率.

本文方法充分利用已有的测试执行信息,并通过渐增式方法迭代生成、运行附加测试,在一定程度上减少了调试人员手工分析错误的工作量. 然而,并发程序的调试是非常复杂和困难的,以后的工作中将进一步探讨多种并发机制下的不确定性故障定位方法.

[**参考文献**](**References**)

[1]　CHOI J D, ZELLER A. Isolating failure inducing thread schedules[C]//Proceedings of the ACM SIGSOFT International

Symposium on Software Testing and Analysis. Rome,Italy,2002.
[2] DALLMEIER V,LINDIG C,ZELLER A. Lightweight defect localization for Java[C]//Proceedings of the 19th European Conference on Object-Oriented Programming. Glasgow,UK,2005.
[3] JONES J A,HARROLD M J. Empirical evaluation of the tarantula automatic to assist fault localization[C]//Proceedings of the 20th International Conference on Automated Software Engineering. California,USA,2005.
[4] YU Y,JONES J,HARROLD M J. An empirical study of the effects of test-suite reduction on fault localization[C]//Proceedings of the 30th International Conference on Software Engineering. Leipzig,Germany,2008.
[5] KOCA F,SOZER H,RUI A. Spectrum-based fault localization for diagnosing concurrency faults[C]//Proceedings of the 25th IFIP WG 6.1 International Conference on Test Software and systems. Istanbul,Turkey,2013.
[6] WONG W E,GAO R,LI Y,et al. A survey of software fault localization[J]. IEEE transactions on software engineering,2016,42(8):707-740.
[7] TANG C M,CHAN W K,YU Y T,et al. Accuracy graphs of spectrum-based fault localization formulas[J]. IEEE transactions on reliability,2017,66(2):403-424.
[8] ARTHO C. Iterative delta debugging[J]. International journal on software tools for technology transfer,2011,13(3):223-246.
[9] HAMMOUDI M,BURG B,BAE G,et al. On the use of delta debugging to reduce recordings and facilitate debugging of web applications[C]//Proceedings of the 10th Joint Meeting on Foundations of Software Engineering. Bergamo,Italy,2015.
[10] GROCE A,ALIPOUR M A,ZHANG C,et al. Cause reduction:delta debugging,even without bugs[J]. Software testing verification and reliability,2016,26(1):40-68.
[11] XU J,LEI Y,CARVER R. Using delta debugging to minimize stress tests for concurrent data structures[C]//Proceedings of the IEEE International Conference on Software Testing,Verification and Validation. Tokyo,Japan,2017.
[12] LIU C,FEI L,YAN X,et al. Statistical debugging:a hypothesis testing-based approach[J]. IEEE transactions on software engineering,2006,32(10):831-848.
[13] LI J. Nonparametric multivariate statistical process control charts:a hypothesis testing-based approach[J]. Journal of nonparametric statistics,2015,27(3):384-400.
[14] DING Z,WANG R,HU J,et al. Detecting bugs of concurrent programs with program invariants[C]//Proceedings of the IEEE International Conference on Software Quality,Reliability and Security Companion. Vienna,Austria,2016.
[15] SATO K,DONG H A,LAGUNA I,et al. Noise injection techniques to expose subtle and unintended message races[C]//Proceedings of the 22th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. Texas,USA,2017.
[16] LAMPORT L. Time,clocks,and the ordering of events in a distributed system[J]. Communications of the ACM,2008,21(7):558-565.

[责任编辑:严海琳]