基于搜索的上下文一致性错误处理

江 磊 许 畅 陈小康
（南京大学计算机软件新技术国家重点实验室 南京 210023）
（南京大学计算机科学与技术系 南京 210023）

摘 要 近年来，随着智能设备的普及和传感技术的发展，上下文感知程序的应用越来越广泛。但是由于环境噪声难以预测和控制，程序所获得的上下文经常存在一致性错误。处理这类错误的方法很多，但大都忽视了两方面的问题：1)不同一致性约束之间存在相互干扰；2)处理这类错误的操作本身可能对程序的正常运行造成负面影响。以处理这两方面的问题为目标，提出了一种新的基于搜索的上下文一致性错误处理方法，亦即既设计出一个搜索空间来查找避免约束间相互干扰和对程序产生负面影响的解，又采用了一种增量式评估方案来加速搜索的效率。经实验评估，新方法能够在很短的时间内达到非常接近最优解的效果。

关键词 基于搜索的软件工程，上下文一致性错误，约束间干扰，副作用，普适计算

中图法分类号 TP311.5 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2014.11.009

Search-based Automated Resolution of Context Inconsistency

JIANG Lei XU Chang CHEN Xiao-kang
(State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China)
(Department of Computer Science and Technology, Nanjing University, Nanjing 210023, China)

Abstract In recent years, with the popularization of intelligent equipment and the development of sensor technologies, context-aware applications keep emerging. However, contexts available for applications are prone to inconsistency due to unpredictable and uncontrollable environmental noises. There have been many techniques proposed for resolving such inconsistency. However, they largely overlook two aspects. First, inconsistency resolution for one constraint may lead to violation of another constraint (i. e., constraint interfering). Second, inconsistency resolution may affect application functionality or quality (i. e., side effect). So a novel search-based automated resolution technique for context inconsistency was proposed to address these two aspects. Efforts were paid to run the inconsistency resolution efficiently via incremental computing. Experiments show that the newtechnique can produce satisfactory resolution results with negligible time cost.

Keywords Search-based software engineering, Context inconsistency, Constraint interfering, Side effect, Pervasive computing

1 引言

在普适计算(pervasive computing)中，有一些信息能反映周围的环境状态，如温度、湿度、位置等，我们把这些信息称为上下文(context)。

近年来，随着智能设备的普及，上下文感知程序(context-aware applications)的应用也越来越广泛。这些程序提供了导航、天气、日程安排等功能，给人们的生活带来了很大的便利。

但是，由于环境噪声的影响，程序获得的上下文通常包含一致性错误(context inconsistency)，这会导致上下文感知程序运行异常甚至出错，给用户带来不必要的麻烦。因此，检测并处理上下文一致性错误是比较重要的。

比如我们常用的汽车导航软件，它通过GPS获得位置信息上下文。但是由于噪声的影响，这些上下文往往是存在一致性错误的，如果不进行处理会使程序对当前的位置发生错误的判断，进而可能导致导航错误，给用户带来麻烦。如果我们能有效检测并处理这种错误，就能在很大程度上避免这样的麻烦，使导航软件更加好用。

上下文一致性错误检测和处理方面已经有很多研究工作，但是这些工作在两个方面上存在不足。首先，我们在检测

到稿日期：2013-09-16 返修日期：2013-11-16 本文受国家863计划(2012AA011205)，国家自然科学基金(青年基金项目)(61100038)，国家自然科学基金(中美软件合作研究项目)(61361120097)，国家基金委可信集成项目(91318301)，国家自然科学基金(创新群体)(61321491)，教育部新世纪优秀人才支持计划(NCET-10-0486)资助。

江 磊(1988－)，男，硕士，主要研究领域为普适计算中的上下文管理、软件工程，E-mail：ladd.cn@gmail.com；许 畅(1977－)，男，博士，副教授，CCF会员，主要研究领域为软件工程、软件测试与分析、普适计算，E-mail：changxu@nju.edu.cn(通信作者)；陈小康(1989－)，男，硕士，主要研究领域为普适计算中的上下文管理、软件工程。

上下文一致性错误时，需要设计若干条约束条件。但是，这些约束之间存在相互干扰，针对一条约束的一致性错误处理操作可能会导致另一条约束被违反。而现有的工作对于这个问题没有好的解决方法[9]。其次，针对上下文一致性错误的处理操作本身会对程序的正常运行造成影响。在一些情况下，这些影响还是非常严重的：我们在处理一致性错误时删掉的上下文可能会在后面用到，有一些对程序来说还是很重要的，这样的删除操作势必导致程序的运行受到大的影响甚至发生崩溃。而传统的方法基本上都忽视了这种影响[12]。

针对这两个问题，我们设计了基于搜索的方法。先对这两个方面进行量化，并合并成一个综合打分函数。然后我们利用优化搜索方法在可行的解空间内寻找一个综合影响低的较优解。最后用这个解去真正执行便能有效解决这两个问题。

其中，在上下文一致性错误检测和处理时，会针对设计的约束条件建立相应的语法分析树[6]。搜索打分时需要对这些树进行大量的真值计算，而这些真值计算本身是十分耗时的。为此我们采用了增量计算，并在文献[11,12]中所用方法的基础上做了进一步的优化，显著地提高了计算效率。在后面的实验中，我们通过有效的增量计算，将时间开销降到了非增量的 1/20。

实验表明，我们的方法能在较短的时间内达到接近于最优的效果。在系统方面，我们实现了一个上下文管理模块。通过这个模块，开发者可以很方便地处理上下文一致性错误。

本文第 2 节介绍了背景知识；第 3 节详细介绍了我们提出的方法，并进行了复杂度分析；第 4、5 节介绍了系统实现以及实验；第 6 节介绍了相关工作；最后进行了总结和展望。

2 背景知识

2.1 上下文

上下文可以用多种数据格式表示，我们选取了一种比较常用的格式[5]，如下所示：

$context::=\{(att_1, val_1), (att_2, val_2), \cdots\}$

在众多的上下文中，有一些具有相同的特性，比如在同一个房间内采集到的所有位置信息。这些上下文可以组成一个集合，我们称之为模式(pattern)，定义如下：

$pattern::=\{ctx_1, ctx_2, \cdots\}$

2.2 上下文一致性错误检测

通过设计上下文一致性约束，并计算当前的上下文是否符合这些约束，可以检测上下文一致性错误。这些约束一般是对场景中的一些物理规律和逻辑规范的抽象。比如一个 LBS 应用场景中，可以基于物理规律设计这样一条约束(记作约束 1)，“在同一时间，一个人不可能位于两个不同的房间”。如果当前上下文违反了这条约束，我们就认为发生了一致性错误。

为了方便分析和处理，我们约定上下文一致性约束必须按照下面这套一阶逻辑文法来表示：

$$f::=\forall x\in pat(f)\mid \exists x\in pat(f)$$
$$\mid (f)\ \text{and}\ (f)\mid (f)\ \text{or}\ (f)\mid f)\ \text{implies}\ (f)$$
$$\mid \text{not}\ (f)\mid \text{equals}(v_1.att_1, v_2.att_2)$$

则约束 1 可以写成下面的形式：

$$\forall x\in pat(not(\exists y\in pat(equals(x.id, y.id))))$$

具体地判断当前的上下文是否违反了某条约束，可以利用语法分析树来计算[6]。

2.3 上下文一致性错误处理

通过有选择地删除不一致的上下文，可以有效地处理上下文一致性错误。

C. Nentwich 等[6]提出了一种叫做 xlinkit 的方法，它不仅能检测出哪些约束被违反，还能给出是哪些上下文的组合导致了这些约束被违反。这些组合被称为线索(link)，格式如下：

$link::=(\text{violated}, \{ctx_1, ctx_2, \cdots\})$

同时由线索组成的集合被称为线索集(linkset)，对应的格式为：

$linkset::=\{link_1, link_2, \cdots\}$

比如，约束 1 在一定的场景下，我们可能会得到下面的线索集：

$\{(\text{violated}, \{ctx_1, ctx_3\}), (\text{violated}, \{ctx_5, ctx_7\})\}$

线索集能为下一步生成错误处理策略提供很大的帮助。对于独立的一条约束，我们先计算得到其线索集，然后从每条线索选取一或多条上下文进行删除就能使当前上下文符合这条约束。

在从线索中如何选取要删除的上下文方面，比较常见的方法有 DelAll[1]（从线索中选取所有相关上下文）、DelRandom[3]（从线索中随机选取一条上下文）、DelLatest[1]（从线索中选取最新的一条上下文）、DelFewer[10]（从线索中选取在线索集出现次数最多的一条上下文）等。

针对某个线索集，通过不同的选择方法可以生成不同的上下文一致性错误处理策略。首先我们将处理策略定义如下：

$strategy::=(action: \text{“remove”}, \text{ctxset}: \{ctx_1, ctx_2, \cdots\})$

那么对于某个线索集 $linkset_1$，所有可行的处理策略可以组成如下这么一个集合：

$AST=\{strategy\mid$ 对于 $linkset_1$ 中的每个 $link$，$strategy$ 中至少存在一条上下文属于该 $link\}$

2.4 传统方法存在的不足

在现实场景中，约束往往不止一条，这样，针对一条约束的一致性错误处理操作可能会导致另外一条约束被违反。我们将这种相互干扰简称为约束间干扰(constraint interfering)。

我们举例来说明这种干扰。场景(见图 1)中有 A，B 两个房间，用户只有先通过房间 A，才有可能进入房间 B。用户身上贴的 RFID 标签，可以被门口的阅读器读到。我们定义两条一致性约束，为了方便说明，我们用自然语言描述：

约束 a：“一个人不能同时出现在两个房间内”。

约束 b：“在房间 B 出现的人，必定在房间 A 出现过”。

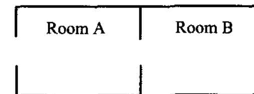

图 1 只有两个房间的简单场景

假定在 14:00 的时候 Tom 进入了 A，20 分钟后进入了房间 B。但是由于环境噪声的影响，在 Tom 进入房间 A 的时候，B 房间的传感器也错误地检测到了 Tom 进入的信息，并生成了相应的上下文。则 A、B 两个房间内的上下文集合如下：

$pattern_{roomA}=\{\{(\text{“name”}, \text{“Tom”}), (\text{“time”}, \text{“14:00”})\}\}$

$pattern_{roomB}=\{\{(\text{“name”}, \text{“Tom”}), (\text{“time”}, \text{“14:00”})\},$

{("name","Tom"),("time","14:20")}}

当前的上下文符合约束2,但是违反了约束1。为了消除这种上下文一致性错误,我们可以选择将 $pattern_{roomA}$ 中的第一条上下文删除,这样约束1得以满足。但是这个消除操作导致了约束2被违反,这就发生了约束间干扰。

通过进一步的研究,我们可以发现,如果把 $pattern_{roomB}$ 中的第一条上下文删除,能使两条约束同时被满足,避免了约束间干扰。由此可知,如果能合理地选择要删除的上下文,约束间干扰是可以被减小或避免的。

我们把上下文一致性错误处理操作对程序正常运行造成的影响限定在程序行为和可用上下文两方面,同时我们简称这种影响为副作用(side effect)。

3 基于搜索的解决方法

3.1 引入搜索算法

正如前面提到的,对于某个线索集我们可以得到所有可能的处理策略组成的集合 AST。对于其中的每个策略,我们能够计算出其执行后会造成的约束间干扰和副作用。如果我们从中选取两方面综合影响最小的策略来执行,就能把这两种影响降下来。但是在普适计算中,AST 中的策略总数通常是十分庞大的(根据后面的实验,在数据量不大的场景中已经能达到 10^7 这样的数量级),遍历的方法在时间上不可行。为此,我们设计了基于遗传算法的搜索方法,在时间和效果上进行折中,并在文献[11,12]所用方法的基础上设计了一种新的增量计算方法,显著地提高了计算效率。

3.2 构造打分函数

对于约束间干扰,我们将处理操作执行后依然被违反的一致性约束条数作为指标。同时为了标准化,我们将其除以处理操作前违反的约束条数。具体如下,其中 n_{cia} 是操作执行后被违反的约束条数,n_{cib} 是操作执行前被违反的条数:

$$score_{ci}=\begin{cases}\dfrac{n_{cia}}{n_{cib}}, & n_{cib}\neq 0\\ 0, & n_{cib}=0\end{cases}$$

对于副作用,需要对两个方面进行量化:一个是处理操作对可用上下文数量产生的影响;另外一个是对程序行为产生的影响。

对于第一种影响,我们认为,被删除的上下文越多,对程序正常运行的影响越大。形式化定义如下,其中 N 是指处理操作之前的上下文总数,n_{del} 是指消除操作中要被删除的上下文条数:

$$score_{ctx}=\begin{cases}\dfrac{n_{del}}{N}, & N\neq 0\\ 0, & N=0\end{cases}$$

对于第二种影响,我们认为处理操作执行后,程序被激发的行为数改变越大,对程序的影响也就越大。具体的形式化如下,其中 n_{sea} 是指处理操作执行后程序被激发的行为数,n_{seb} 是指操作执行前被激发的行为数目:

$$score_{bhv}=\begin{cases}\dfrac{|n_{seb}-n_{sea}|}{n_{seb}}, & n_{seb}\neq 0\\ 0, & n_{seb}=0\end{cases}$$

我们使用构造语法分析树的方法来判断哪些程序行为会被激发。具体地,我们先对程序的逻辑进行分析,归纳出一些程序行为激活规则。然后我们通过语法分析树对当前上下文进行真值计算,就能得知哪些行为会被激发。

比如程序有这样一种行为逻辑"如果 Tom 在房间 A 内,则把咖啡机打开"。我们可以设计如下的行为激活规则:

$$\exists\, ctx\in pat_A(\text{equals}(ctx.name,\text{"Tom"}))$$

我们通过计算这条规则在当前环境下是否为真,就可以得知咖啡机打开这个行为是否会发生。

最后,我们把副作用的这两种影响结合在一起,并认为这两种影响同样重要,可以得出如下的表达式:

$$score_{se}=0.5*score_{ctx}+0.5*score_{bhv}$$

同样,我们将约束间干扰和副作用结合起来,构造一个综合的打分函数:

$$score_{fit}=0.5*score_{se}+0.5*score_{ci}$$

3.3 基于遗传算法的搜索方法

遗传算法具有鲁棒性强、效率高的特点,在优化搜索领域有着广泛的应用,我们用它作为基础算法。

3.3.1 *编码*

为了应用遗传算法,我们对可行的处理策略进行编码。假设得到的线索集有 n 个线索,则相对应我们的编码有 n 位,每一位是一个十进制整数。又假设某个线索含有 m 个上下文 $\{ctx_0,\cdots,ctx_k,\cdots,ctx_m\}$,则我们先计算策略对应的二进制编码 b,具体地,如果策略中第 k 个上下文要被删除,则 b 中第 k 位为1,否则为0。然后将 b 转化为十进制整数 i。比如某个线索中上下文集合为 $\{ctx_1,ctx_2,ctx_3\}$,我们的策略中要删除 ctx_1 和 ctx_3,则 b 为101,i 为5。

3.3.2 *搜索算法*

我们按照遗传算法的几个步骤,设计了优化搜索算法。

一开始,我们先从所有可行的处理策略集合 AST 中随机生成一定数量的策略作为最初的种群。然后进行迭代,每次先对当前种群代表的所有策略进行打分,找出得分最低的一个。然后进行选择,从当前种群中选取同样数量的策略作为父种群。其中被选取的策略可以是重复的,之前得分最高的策略会有更大的概率被选中(见算法1)。接下来,进行杂交,父种群中相邻的两个策略依次进行杂交(见算法2)。最后,从杂交得到的子种群中随机选取几个进行变异。这样一次迭代完成,子种群作为当前种群进行下一次迭代。当迭代次数达到设定值或者已经超过程序允许的时间要求时,停止迭代。将当前种群中的最优解作为结果返回。

算法1 待杂交父种群选取算法

输入:s_{best}(得分最高的策略),individuals(当前种群),individualsNum(当前种群大小)

输出:parentIndividuals(待杂交的父种群)

```
1. FOR i←0 TO individualsNum
2.   IF i%2=0
3.     THEN add s_best to parentIndividuals
4.     ELSE
5.       randomly select an index j
6.       get the j's strategy s_j from individuals
7.       add s_j to parentIndividuals
8. END FOR
9. RETURN parentIndividuals
```

算法2 杂交算法

输入:parentIndividuals(待杂交的父种群),individualsNum(父种群大小),$n_{linkset}$(前面通过 link generation 得到的线索集大小)

输出:childIndividuals(杂交产生的子种群)

```
1. FOR i←0 TO individualsNum/2
```

2. strategy s_a←parentIndividuals. get(i * 2)
3. strategy s_b←parentIndividuals. get(i * 2+1)
4. strategy s_{childA}, s_{childB}
5. randomly select a num k from 1 to $n_{linkset}$
6. add the first k bits of s_a to s_{childA}
7. add the last $n_{linkset}$-k bits of s_b to s_{childA}
8. add the first k bits of s_b to s_{childB}
9. add the last $n_{linkset}$-k bits of s_a to s_{childB}
10. add s_{childA}, s_{childB} to childIndividuals
11. END FOR
12. RERURN childIndividuals

3.4 利用增量计算提高效率

3.4.1 传统增量计算方法

C. Xu 等[11,12]提出了一种增量计算的方法,它可以显著提高计算效率。我们在这里将这种方法称为传统增量方法。

具体地,每次进行上下文一致性检测时会得到一定数量的语法分析树,其中每个节点的真值都计算好了[11]。对于每个策略,会有一个要删除的上下文集合。传统的增量计算方法在计算策略执行后语法树的真值时,不需要重新构造树,只需要将要删除的上下文会影响到的节点重新计算一下即可。这其中主要涉及到删除(对删除某条上下文在语法分析树中会影响到的相关节点标记为删除)和回滚(去除之前受影响节点的删除标记)两种操作[11,12]。

和其他处理方法相比,我们的方法涉及到了更多的真值计算,对这方面的效率要求也就更高。为此,我们在传统增量方法的基础上,做了一些改进,以进一步提高效率。

3.4.2 避免重复计算来提高效率

我们的方法在每次执行时,会对大量的策略进行打分,其中,有一些策略是相同的。对于这些策略,我们如果能避免重复计算就可以节省很多时间。

对此,我们先构造了一个映射表 *scoreTable*,用来保存策略对应的 *key* 及其得分。

每次计算得分时,我们先利用策略的编码数组计算得到一个 long 类型的整数值 *key*。这里我们设计了一个映射函数(其中 N 为线索集的大小,n_j 为线索集中第 j 个线索中上下文集合的大小,为了公式的格式整齐,我们将原本不存在的 n_0 设定为了 0 作为辅助量,$code_i$ 为策略编码的第 i 位):

$$key=\sum_{i=1}^{N}(code_i*\prod_{j=0}^{i-1}2^{n_j})$$

接下来,我们先在 *scoreTable* 中查找 *key* 是否已经存在,如果存在则直接返回对应的得分;否则,计算得分,并将其和 *key* 保存到 *scoreTable* 中,然后返回得分。

3.4.3 利用局部相似性来降低计算规模

我们的方法每次要计算的大量策略中除了有一些是相同的,还有很多是比较类似的,我们可以利用这种局部相似性来降低计算规模。

具体地,我们在计算某个策略打分时,会和上一个策略要删除的上下文集合进行比较,然后只对不同的部分进行删除和回滚操作。比如,我们之前计算的策略 S_1 要删除的上下文集合为$\{ctx_1, ctx_2, ctx_3\}$,我们这次要计算的策略 S_2 要删除的上下文集合为$\{ctx_1, ctx_2, ctx_5\}$,则我们只需要对 ctx_3 进行回滚,对 ctx_5 进行删除,然后计算语法树真值以及得分。其中,在第一次计算时,我们对集合中出现的上下文全部执行删除操作;在最后一次计算的时候,我们在计算完得分后,对其集合中的所有上下文执行回滚操作。

3.5 时间复杂度分析

我们要和两类方法进行时间复杂度上的对比,一类是 DelAll[1]、DelRandom[3]这些传统的启发式的处理方法;另一类是对解空间进行全遍历找最优解的方法以及 DelEffect(每次选择 DelAll、DelRandom、DelLatest、DelFewer 这 4 种方法中效果最好的一个执行)。

我们假设上下文一致性约束和程序的行为激发规则总共有 num_r 条,则相应的语法分析树为 num_r 个,其总节点数为 num_{node} 个。

第一类方法主要的时间消耗在生成语法分析树和 link generation 上,时间复杂度为 $O(num_{node})$。

第二类方法要分开分析,其中 DelEffect 方法和第一类方法相比,只是多了一个打分的步骤,并且只对 4 种方法进行打分,如果采用增量计算,打分的时间消耗很小。因此 DelEffect 的时间复杂度也为 $O(num_{node})$。而全遍历方法的时间消耗主要花在对数量庞大的策略进行打分计算上。我们假设得到的 linkset 中有 n 个 link,每个 link 中上下文的个数为 $num_1, num_2, \cdots, num_n$。那么遍历算法需要计算真值的次数为 $n_{all}=(2_1^{num}-1)*(2_2^{num}-1)*\cdots*(2_n^{num}-1)$。在现实场景中 n 经常为大于 10 的数,num_k 约为 2,这时 $n_{all}>60000$。每次真值计算时间复杂度为 $O(num_{node})$,则总的时间复杂度为 $O(n_{all}*num_{node})$,即使采用增量计算,总的时间开销在需要在线计算的应用中也是无法接受的。

我们的方法通过有效的增量计算,大幅降低了每个策略进行真值计算所消耗的时间。C. Xu 等[11,12]提出的传统增量计算方法在实际场景中可以将效率提高 a 倍,我们的新方法在此基础上又可以将效率提高 m 倍。遗传算法中种群数目为 i,迭代次数为 j,计算规模 n_{cal} 为 $i*j$。同时,我们的方法在计算之前也要生成语法树,并进行 link generation,则总的时间复杂度为 $O(num_{node})+O(n_{cal}*num_{node}/(a*m))$,通过实验我们得知,$a$ 约为 20,m 约为 3,n_{cal} 一般设为 256,则总的计算规模为 $O(num_{node})$,其中常数项系数小于 4。

总体来说,我们的方法在时间复杂度上,比第一类方法和 DelEffect 略大,但远小于遍历的方法。从后面的实验中也可以看出,我们的方法在取得了接近于全遍历方法效果(最优效果)的同时,只花费了比传统方法略多一点的时间。

4 系统实现

为了方便开发者处理上下文一致性错误,我们实现了一个上下文管理模块(见图 2)。这个模块具有下面的特点。

首先,使用方便。用户可以通过编辑 xml 文件来提供一些必要的配置信息。然后添加两行简单的代码就能让我们的模块完成配置和启动。要想将上下文交给我们的模块管理,用户只需要在自己程序中产生上下文的部分添加一行代码,将上下文以键值对的形式传递给我们的模块即可。同样,想要使用上下文时,只需要另外一行代码就能获得处理过一致性错误的上下文。

其次,逻辑简单。用户在使用我们的模块时,只需要把我们的模块当成一个自动运行的仓库即可,需要托管的时候将相关的上下文存进去,需要使用的时候将想要的上下文取出来。上下文的刷新和一致性错误处理等操作都由我们的模块在后台进行,不需要用户进行操作。

此外,功能齐全。我们的模块不仅实现了提出的上下文

一致性错误处理方法，还实现了常见的传统方法，用户可以根据需要自由选择。

我们后面的实验就是基于此系统模块实现的。该模块由 Java 7 实现，代码量约 6000 行。

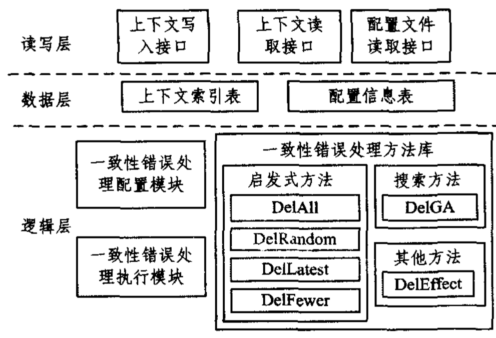

图 2 系统架构图

5 实验评估

5.1 场景设计

我们模拟了一个简单的场景来评估我们的方法。如图 3 所示，这是一个货物中转站，有 5 条传送带，每条都有若干人负责检查、分拣货物。同时每条传送带起始和终止位置都有 RFID 阅读器，可以读取在附近通过的货物标签。这些阅读器会有一定的概率漏读，同时可能会误读到相邻传送带上的货物标签。为了保证员工的工作质量，我们设计了一个程序，它可把传送带终端 RFID 读取器读到的货物标签信息，通过时间、传送带编号一起保存到数据库中。同时，通过这个程序，我们可以方便地查找每件货物对应的责任人。此外，每隔 5 秒钟，程序会把当天处理的货物总量刷新在中转站的大屏幕上。

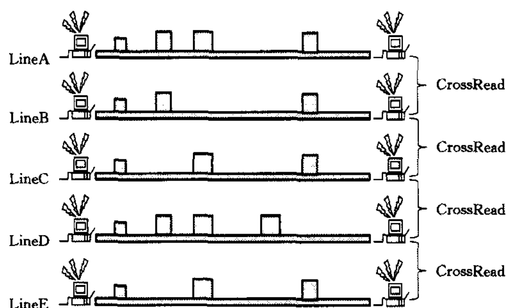

图 3 货物中转站的场景

由于漏读和误读的存在，程序获得的上下文存在一致性错误，我们要先处理这些错误，再将其保存到数据库。我们一共设计了 9 条约束来检测漏读、误读导致的上下文一致性错误。这些约束和 2.4 节中的约束 a、b 类似。同时，我们将前面提到的程序用 6 条规则来表示其行为逻辑。

我们根据现实场景中 RFID 阅读器的错误率统计情况，将漏读率固定为 30%，误读率分别设置为 20%、30%和 40%进行 3 组实验[5]。其中各组实验中均是每 100ms 产生 1 件货物，依次分配到 5 条传送带上。由于在程序允许的范围内可适当地延迟处理时间，而不是每产生一条上下文就处理一次，因此可以利用更多信息，达到更好的效果[2,10]。我们将两次相邻上下文一致性错误处理操作的时间间隔定为 2s。

我们的实验机器的硬件配置为 Intel Core 2 Quad 2.83GHz CPU，4GB RAM。操作系统为 Windows 7 Professional SP1，Java 虚拟机版本为 Oracle/Sun JRE 1.7。

5.2 结果分析

5.2.1 和传统启发式方法的比较

我们方法（DelGA）与 DelAll、DelRandom、DelLatest、DelFewer 在效果和效率上进行了对比。从图 4 中可以看出，在 3 组不同的实验中，我们的方法在降低综合影响的效果上都明显优于传统方法。同时，从图 5 中可以看出，我们的方法由于在选择执行方案时考虑了更多的情况，因此在时间消耗上略大于传统的方法。

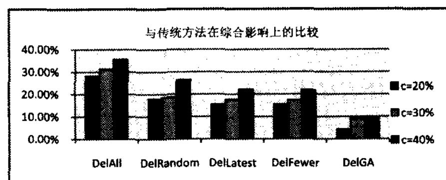

图 4 与传统方法在综合影响上的比较

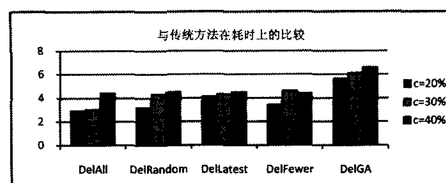

图 5 与传统方法在耗时上的比较

5.2.2 非增量、传统增量和新增量计算方法的比较

首先，我们将新增量计算方法和传统增量计算方法[11,12]在计算规模和时间消耗上进行了比较。增量计算中主要的两个操作是删除和回滚。我们用这两个操作的执行次数来评估计算规模。从表 1 可以看出，我们的新方法将计算规模降为了传统增量的约 1/7，同时在时间消耗方面降为了约1/2。时间消耗上的效果看起来不显著，这是因为我们在计算每个策略对应的索引值时消耗了一定的时间。

表 1 非增量、传统增量和新增量计算方法的效率比较（数据均为 3 组实验的平均值）

方法	改变数	耗时(ms)
非增量方法		125.2
传统增量方法	1403.3	11.4
我们的增量方法	209.1	6.1

其次，对于非增量方法[6]，由于主要的操作类型不同，我们仅对耗时进行比较。我们的方法耗时仅为非增量的 1/20 左右。

5.2.3 与全遍历方法以及 DelEffect 的比较

最后，我们与全遍历以及 DelEffect 方法在效率和降低综合影响的效果上进行了对比。

从表 2 可以看出，我们的方法（92.4%）在降低综合影响的效果上接近于全遍历的方法（94.8%），达到了其97.4%，但在时间消耗上有明显的优势（约为其 1/3740）。全遍历的方法耗时已经远超过了设定的两次处理操作的时间间隔（2s），而我们的方法能够较好地符合时间要求。对于 DelEffect，我们的方法时间消耗略多，但效果要好得多。

表 2 和全遍历方法以及 DelEffect 的效率、效果比较（数据均为 3 组实验的平均值）

方法	耗时(ms)	综合影响
全遍历方法	22814.9	5.2%
DelEffect	4.8	14.3%
我们的方法	6.1	7.6%

6 相关工作

上下文一致性错误检测方面有两个比较重要的工作。其中一个是我们前面提到的 xlinkit 方法[6]，另一个是 C. Xu 等[11]提出的通过有效增量计算来提高检测效率的方法。

上下文一致性错误处理方面，有一些工作是有关约束间干扰的。其中 C. Nentwich 等[7]在未来工作中提到过约束间干扰，但没有给出具体的解决方法。Y. Xiong 等[9]对这个问题有深入的研究，并提出了 3 种不同的策略。第一种是直接忽略，在约束间干扰比较严重的场景中这是行不通的。第二种是在计算的过程中完全避免生成会造成约束间干扰的解。这个策略存在大量空解，有效性不好。第三种是通过复杂的计算，利用增删改 3 种操作得到一个不会造成约束间干扰的解。这种方法最大的缺点是非常耗时[9]，另外在普适计算环境中，导致一致性错误发生的原因是非常复杂的，增加和修改操作很难保证有效性。相比而言，我们的方法能够保证每次都会产生一个接近最优的解，保证了有效性。同时还利用了新的增量计算方法，保证了效率。至于副作用，相关的研究工作不多，C. Xu 等[12]提出了这个问题，并提供了一个解决方法。在每次执行操作时，从几种传统方法中选取副作用最小的一个真正执行，能有效地降低副作用。与这个方法相比，我们的方法优化的范围更大(从所有可行的解中选取一个较优解)，时间开销上稍微大一点，但效果会更好。同时，我们的方法还能降低约束间干扰。

搜索方法在软件测试、需求分析等软工方向已经得到了广泛的应用[4]。在上下文一致性管理领域，目前还没有相关的工作。一个重要原因是，上下文感知程序对一致性错误处理的时效性要求严格，而语法树的真值计算十分耗时，导致打分效率低，难以进行有效的搜索。我们通过新的增量计算方法提高了效率，使搜索方法得以应用。

结束语 在本文中，我们提出了一种基于搜索的上下文一致性错误处理方法，它能够同时降低约束之间的相互干扰和处理操作本身对程序正常运行造成的影响。同时，我们设计了一种新的增量计算方法，显著提高了计算的效率。实验表明我们的方法可以在较短的时间内达到接近最优的效果。在系统方面，我们实现了一个上下文管理模块，用户可以很方便地用它来解决上下文一致性错误。

当然，我们的方法还有几个方面有待完善：首先，我们只考虑了利用删除操作来处理上下文一致性错误，在未来的工作中可以试着加入增加和修改操作；其次，评估上下文一致性错误处理操作本身对程序正常运行所造成的影响，我们只考虑了两个方面，未来可以加入其它的方面；最后，在系统实现方面我们还要做进一步的完善。

参 考 文 献

[1] Bu Y, Gu T, Tao X, et al. Managing quality of context in pervasive computing[C]// Sixth International Conference on Quality Software, 2006. QSIC 2006. IEEE, 2006: 193-200

[2] Chen C, Ye C, Jacobsen H A. Hybrid context inconsistency resolution for context-aware services[C]// 2011 IEEE International Conference on Pervasive Computing and Communications (PerCom). IEEE, 2011: 10-19

[3] Demsky B, Rinard M C. Goal-directed reasoning for specification-based data structure repair[J]. IEEE Transactions on Software Engineering, 2006, 32(12): 931-951

[4] Harman M, Mansouri S A, Zhang Y. Search based software engineering: A comprehensive analysis and review of trends techniques and applications[R]. Tech. Rep. TR-09-03. Department of Computer Science, King's College London, 2009

[5] Julien C, Roman G C. Egospaces: Facilitating rapid development of context-aware mobile applications[J]. IEEE Transactions on Software Engineering, 2006, 32(5): 281-298

[6] Nentwich C, Capra L, Emmerich W, et al. xlinkit: A consistency checking and smart link generation service[J]. ACM Transactions on Internet Technology (TOIT), 2002, 2(2): 151-185

[7] Nentwich C, Emmerich W, Finkelstein A. Consistency management with repair actions[C]// Proceedings. 25th International Conference on Software Engineering, 2003. IEEE, 2003: 455-464

[8] Sullivan L. RFID implementation challenges persist, all this time later[J]. Information Week, 2005, 1059: 34-40

[9] Xiong Y, Hubaux A, She S, et al. Generating range fixes for software configuration[C]// 2012 34th International Conference on Software Engineering (ICSE). IEEE, 2012: 58-68

[10] Xu C, Cheung S C, Chan W K, et al. Heuristics-based strategies for resolving context inconsistencies in pervasive computing applications[C] // 28th International Conference on Distributed Computing Systems, 2008 (ICDCS'08). IEEE, 2008: 713-721

[11] Xu C, Cheung S C, Chan W K, et al. Partial constraint checking for context consistency in pervasive computing[J]. ACM Transactions on Software Engineering and Methodology (TOSEM), 2010, 19(3): 9

[12] Xu C, Ma X, Cao C, et al. Minimizing the Side Effect of Context Inconsistency Resolution for Ubiquitous Computing[M] // Mobile and Ubiquitous Systems: Computing, Networking, and Services. Springer Berlin Heidelberg, 2012: 285-297

(上接第 11 页)

[2] National Intelligent Transportation Systems Program Plan: A Ten-Year Vision[M] // The Intelligent Transportation Society of America and U. S. Department of Transportation, 2002

[3] Http://europa. eu. int/comm/transport/themes/network/english/its/html/vision_policy. html

[4] Http://www. ertico. com/its_basi/eu_polic. htm

[5] 杨冰，等. 智能运输系统[M]. 北京：中国铁道出版社，2000

[6] Japan ITS Hand Book[M]. Supervised by Road Bureau, The Ministry of Land, Infrastructure and Transport, 2001

[7] 陈桂香. 国外智能交通系统的发展情况[J]. 中国安防，2012(6)：103-108

[8] 张可，齐彤岩，等. 中国智能交通系统(ITS)体系框架研究进展[J]. 交通运输系统工程与信息，2005，5(5)：5-11

[9] 杨琪. 智能运输系统标准化状况及发展趋势综述[J]. 交通标准化，2011(24)：8-10

[10] 孙其博，刘杰，等. 物联网：概念、架构与关键技术研究综述[J]. 北京邮电大学学报，2010，33(3)：1-9

[11] 陈全，邓倩妮. 云计算及其关键技术[J]. 计算机应用，2009，29(9)：2562-2567

[12] 岳建明. 我国智能交通产业的发展及技术创新模式探讨[J]. 中国软科学，2012(9)：188-192