基于动态内存和状态管理的模型检测新方法

吴立军[1] 骆翔宇[2] 陈清亮[3]
(电子科技大学计算机科学与工程学院 成都 610054)[1] (清华大学软件学院 北京 100084)[2]
(暨南大学计算机科学学院 广州 410074)[3]

摘 要 模型检测是并发系统验证的主要形式化方法之一,但其存在因状态空间爆炸而导致内存不够的问题,这也是大规模并发系统验证的瓶颈。很多研究人员尽管做了很多相关研究,但仍然没有很好地解决这个问题。在研究动态内存和状态管理的基础上,提出了一种新的模型检测方法,避免了因为内存不足而无法模型检测的问题。

关键词 形式化方法,模型检测,状态空间爆炸,状态和内存管理
中图法分类号 TP31 **文献标识码** A

New Approach of Model Checking Based on Management for Dynamic Memory and State

WU Li-jun[1] LUO Xiang-yu[2] CHEN Qing-liang[3]
(School of Computer Science, University of Electronic Science and Technology, Chengdu 610054, China)[1]
(School of Software, Tsinghua University, Beijing 100084, China)[2]
(Department of Computer Science, Jinan University, Guangzhou 410074, China)[3]

Abstract The model checking is one of main formalization methods. However there is the problem about state explosion and insufficient memory, which is bottleneck of verification of large scale systems. Though many researchers have done amount of work, the problem has not been settled well yet. The paper, based on the investigation of the management for fixed memory and state, presented a new approach of model checking, which avoids the problem that model checking can not proceed because of insufficient memory.

Keywords Formal method, Model checking, State explosion, Management of state and memory

1 引言

模型检测是并发系统验证的主要形式化方法之一,它最大的优势是能实现系统的自动化验证,自 Clarke 1981 年提出以来[1],已经被应用到工业界的验证中,然而其最大的缺点是状态空间爆炸,这也是大规模并发系统验证的瓶颈。近年来许多学者提出了缓解状态空间爆炸的方法,如偏序约简[2]、对称约简[3]、抽象技术[4]、组合理论[5]、演算技术[6]、符号模型检测[7]等技术。F. Raimondi、A. Lomusico 和苏开乐教授等利用 OBDD 研究了多智能体系统的自动验证问题[9,10],将 OBDD 方法从时态逻辑模型检测扩展到时态认知逻辑模型检测。然而对于更大的状态空间的情况,仍然存在指数爆炸的问题难以处理。C. Jard 和 T. Jeron 提出了 on the fly 模型检测 LTL 的方法[11],其系统状态在检测过程中按需动态生成,因此在找到反例之前往往只需生成小部分状态空间。但有的情况可能在找到反例之前仍需要产生大部分状态,所以仍然难以彻底解决状态空间爆炸问题。

以上方法的基本思路都是状态空间的约简或转换,然而当系统规模很大时,系统(包括约简后的系统)产生的状态数如果达到或超过了计算机内存的极限,就会因内存不足而导致模型检测无法进行。

为了解决以上不足,本文在"on the fly"模型检测方法的基础上,通过研究动态内存和状态的有效管理,提出了新的模型检测方法,使模型检测不受内存空间的限制,并能在具有任意大小内存的计算机上进行超大规模并发系统的模型检测。

本文第 2 节介绍了"on the fly" 模型检测;第 3 节研究动态内存和状态的管理;第 4 节研究基于动态内存和状态的管理的模型检测算法;第 5 节分析算法的正确性;最后是结论。

2 on the fly 模型检测

"on the fly" 模型检测方法包括 3 方面:时态逻辑公式转化为自动机;系统自动机和乘积自动机按需产生;强连通图(即环路)的搜索。不少作者研究了 on the fly 模型检测方法,Gerth, Peled, Vardi, and Wolper 在文献[12]提出了将 LTL 公式转化为自动机的算法,并且研究了系统自动机和乘积自动机按需要产生的思想和方法,从而实现 on the fly 模型检测。

"on the fly" 模型检测的主要优点是,一般情况下在找到反例之前只需产生小部分状态空间。其缺点是当系统规模很大时,仍然存在因内存不够模型检测无法进行的问题。

根据文献[13],"on the fly" 模型检测方法分 3 步:

到稿日期:2010-12-23 返修日期:2011-07-05 本文受国家自然科学基金项目(61073033,60763004)资助。
吴立军(1965—),男,博士后,副教授,主要研究方向为人工智能与网络安全,E-mail:wljuestc@gmail.com;骆翔宇(1969—),男,博士后,副教授,主要研究方向为网络与信息安全。

(1)将系统和规范(系统需满足的安全性质)转化为自动机;

(2)计算系统和规范自动机的乘积自动机;

(3)搜索从初始状态开始经过某个接受状态的回路,如果存在这样一条回路,则系统不满足规范,且该回路就是一个反例。

具体过程如下:

设系统自动机 $M_1=(\Sigma,V,\sigma,V_0,V)$,规范自动机 $M_2=(\Sigma,S,\rho,S_0,F)$,则 M_1 和 M_2 的乘积自动机 $M_1 * M_2$ 为如下 Büchi 自动机 M:

(1)状态集为 $V*S$;

(2)转移函数 $T:V*S*\Sigma\rightarrow 2^{V*S}$ 定义为:$(v_2,s_2)\in T((v_1,s_1),a)$,当且仅当 $v_2\in\sigma(v_1,a)$且 $s_2\in\rho(s_1,a)$;

(3)初始状态集为 V_0*S_0;

(4)接受状态集为 $V*F$。

实际上,系统自动机和乘积自动机是按需生成的。

路径搜索如下:首先产生乘积自动机 M 的初始状态集 V_0*S_0 和接受状态集 $V*F$。从初始状态 $t_0\in V_0*S_0$ 开始,采用第一个深度优先搜索法,找到一个达到 $V*F$ 中某个状态的路径(每个状态的后继状态按需产生),然后采用第二个深度优先搜索法检查是否存在从 $V*F$ 中这个状态出发的回路,如果存在,就可以获得一条路径,该路径就是一个反例。由于状态是按需产生,因此在获得一条运行路径之前,往往只需产生部分甚至小部分状态空间。

3 动态内存和状态的管理

这里主要研究动态内存状态管理。

3.1 阈值的确定

计算机的内存空间根据其用途可以分为程序区域和数据区域,状态存放在数据区域,将这部分称作内存有效状态区域。设内存有效状态区域为 S,S 最多能存放的状态数为 K,双 DFS 深度优先搜索法中的两个堆栈 $Stack_1$ 和 $Stack_2$ 共同动态使用 S。

在搜索过程中,当两个堆栈存放的状态数之和达到 K 后,再产生状态时就会因内存不够而产生溢出,从而导致模型检测无法进行。所以 K 是动态内存管理和状态控制的阈值。

为了避免因内存不够而产生溢出,当两个堆栈存放的状态数之和达到阈值 K 后,就应该将 S 中的状态调出内存,但是这样可能会导致内存抖动的问题。这里内存抖动是指状态调出内存后,紧接着又需要从状态数据库中将状态调入内存,这样在短时间内,多次反复调入调出状态,我们将这种状态的频繁调入调出称作内存抖动。内存抖动必然耗费大量的时间。

我们解决内存抖动的方法是将堆栈里的状态分为交换状态和缓冲状态。下面进行比较。

堆栈里的状态全部为交换状态的情况:当内存有效状态区域 S 已满时,将堆栈中全部状态调出内存,并将所占内存清空;此后,如果算法紧接着要弹出某个状态时,又需将状态数据库中的一批状态调入内存(作为交换状态),此时堆栈满;如果紧接着有某个状态要压入,又需将交换状态全部调出内存腾出空间。这样造成了内存中状态的频繁调入调出,从而出现内存抖动。

堆栈里的状态分为交换状态和缓冲状态的情况:当内存有效状态区域 S 已满时,只将堆栈中全部交换状态调出内存,并将交换状态所占内存清空,此时缓冲状态仍在堆栈中,堆栈非空;此后,如果紧接着有某个状态要弹出,只需将堆栈中最上面的缓冲状态弹出,而不需要将状态数据库中的状态调入内存,此时堆栈非满;如果紧接着有某个状态要压入,由于此时堆栈非满,因此只需将该状态存入堆栈中,而不需将堆栈中的状态调出内存腾出空间。这样避免了内存中状态的频繁调入调出,从而解决了内存抖动问题。

3.2 动态内存和状态管理

(1) 堆栈 $Stack_1$ 和 $Stack_2$ 共同占用内存有效区 S,当 $Number(Stack_1)$(即堆栈 $Stack_1$ 中存放的状态数)与 $Number(Stack_2)$(即堆栈 $Stack_2$ 中存放的状态数)之和等于阈值 K 时,需要调出内存中的状态,但如果全部调出,就会引起内存的抖动。因此从 $Stack_1$ 中调出 $M_{11}=Number(Stack_1)/2$ 个状态,其余 $M_{12}=Number(Stack_1)-M_{11}$ 个状态仍保留在内存中,由于这些状态是为了避免内存抖动的,因此称这些状态为 $Stack_1$ 的缓冲状态。M_{11} 个状态调出后存放在数据库 DB_1 中,它们在 $Stack_1$ 中的存储区被释放,栈底指针指向第 $M_{11}+1$ 个状态,同样,从 $Stack_2$ 中调出 $M_{21}=Number(Stack_2)/2$ 个状态,其余 $M_{22}=Number(Stack_2)-M_{21}$ 个状态仍保留在内存中,由于这些状态是为了避免内存抖动的,因此称这些状态为 $Stack_2$ 的缓冲状态。M_{21} 个状态调出后存放在数据库 DB_2 中,它们在 $Stack_2$ 中存储区被释放,栈底指针指向第 $M_{21}+1$ 个状态。相应的算法描述如图 1 所示。

```
Function D-memtodb()
    Q:={s1,…sM11 | s1,…,sM11 are M11 states starting from bottom
        of Stack1};
    append Q to DB1;
    bottom pointer points to sM11+1;
    set free memory from s1 to sM11;
    Q:={t1,…,tM21 | s1,…,tM21 are M21 states starting from bottom
    of Stack2};
    append Q to DB2;
    bottom pointer points to tM21+1;
    set free memory from t1 to tM21 in Stack2;
End function
```

图 1 函数 D-memtodb()的描述

(2) 当堆栈 $Stack_1$ 为空时,需要从数据库 DB_1 中将最后存入的 $N_1=(K-Number(Stack_2))/2$ 个状态调入内存中,并且在数据库 DB_1 中删除这 N_1 个状态。相应的算法描述如图 2 所示。

```
Function D-Dbtomem1()
    Q:={s1,…,sN1 | s1,…,sN1 are the data of the last N1 states in
        DB1};
    push Q into stack1;
    Delete Q from DB1;
End function
```

图 2 函数 D-Dbtomem1()的描述

同样,当堆栈 $Stack_2$ 为空时,需要从数据库 DB_2 中将最后存入的 $N_2=(K-Number(Stack_1))/2$ 个状态调入内存中,并且在数据库 DB_2 中删除这 N_2 个状态。相应的算法描述如

图 3 所示。

```
Function D-Dbtomem2()
    Q:={s1,…,sN2 | s1,…,sN2 are the data of the last N2 states in
        DB2};
    push Q into stack2;
    Delete Q from DB2;
    End function
```

图 3　函数 D-Dbtomem2()的描述

4　基于动态内存和状态管理的模型检测算法

设系统规范用时态公式 φ 表示，为了表述的方便，将 $\neg\varphi$ 对应的自动机称为规范自动机 M_1。设 $M_1=(\Sigma,V,\sigma,V_0,V)$，系统自动机 $M_2=(\Sigma,S,\sigma,S_0,F)$，那么 M_1 和 M_2 的乘积自动机 $M_1 * M_2$ 为如下 Büchi 自动机 M：

(1)状态集为 $V * S$；

(2)转移函数 $T:V * S * \Sigma \rightarrow 2^{V*S}$ 定义为：$(v_2,s_2)\in T((v_1,s_1),a)$ 当且仅当 $V_2\in\sigma(v_1,a)$ 且 $S_2\in\rho(s_1,a)$；

(3)初始状态集为 $V_0 * S_0$；

(4)接受状态集为 $V * F$。

要证明系统是否满足指定的规范，只需要判定能否在乘积自动机内找到一条经过接受状态(也是可达状态)的环路，如果存在这样一条环路，那么系统就不满足规范，而且这条路径就是一个反例，否则系统满足规范。

根据上述动态状态及内存管理技术，在文献[13,14]的基础上，提出基于动态内存和状态调度的模型检测方法，该算法能避免因内存不足模型检测无法进行的问题。

算法分两步：

第一步　首先从初始状态 $q_0\in V_0 * S_0$ 开始，找到一个达到某个接受状态 s 的路径，在搜索过程中，需要考虑状态空间爆炸的问题，具体算法见 4.2 节的 function $dfs1()$。

第二步　检查是否存在从状态 s 出发的回路，求出该回路，并得到一条运行路径. 在搜索过程中，同样需要考虑状态空间爆炸的问题，具体算法见 4.2 节的 function $dfs2()$。

4.1　路径的搜索

首先从 $V_0 * S_0$ 的任意初始状态 q_0 开始，通过第一次深度优先搜索，找到 $V * F$ 中某个接受状态。然后通过嵌套深度优先搜索检查是否存在从这个状态出发的回路，这里 H_1 和 H_2 都是 hash 表，其中 H_1 存放第一个深度优先算法 $dfs1()$中已分析过但不在堆栈 $Stack_1$ 中的状态，H_2 存放第二个深度优先算法 $dfs2()$中已分析过但不在堆栈 $Stack_2$ 中的状态；$Stack_1$ 和 $Stack_2$ 是全局堆栈变量，$Stack_1$ 存放初始状态到所找的 $V * F$ 中状态的路径，$Stack_2$ 存放从 $V * F$ 中这个状态到 Q_1 中某个状态的路径，由 $Stack_1$ 和 $Stack_2$ 可以得到一条运行路径，这条运行路径就是一个反例。为了提高搜索速度，使用状态集合 Q_1，在搜索过程中，Q_1 的值为 $Stack_1$ 中的状态和经过的接受状态的集合，如果当前状态的后继状态在 Q_1 中，那么显然就存在一条经过某接受状态的环路，算法就终止。图 4 描述基于动态内存和状态管理的路径搜索算法(其中 cpupath()见 4.2 节)。下面代码中 $successor(\chi)$ 表示 χ 的后继结点的集合。

```
Procedure emptiness
    For all q0∈V0 * S0 do
    Stack1=∅;H1:=∅;Stack2=∅;H2:=∅;
    Dfs1(q0);
    Terminate(false);
End procedure
function dfs1(p)
    Q:=∅;
    Push p into Stack1;
    while Stack1≠∅ do
        χ:=top(Stack1);
    Q:=successor(χ);
      If ∃s'∈Q and s' is not in H1
    Begin
        If number(Stack1)+number(Stack2)=K then
        D-Memtodb();
      Push s' into Stack1;
    end
    else
    begin
      Pop χ from Stack1;
      put χ in H1;
      If number(Stack1)=0 then
        D-Dbtomem1();
        If χ∈V * F then
          dfs2(();
    end
  End while
End function

function dfs2(q)
    Q:=∅;Q1:=Stack1;Path:=∅;
    If number(Stack1)+number(Stack2)=K then
      D-Memtodb();
        Push q into Stack2;
        While Stack2≠∅ do
          v:=top(Stack2);
          Q:=successor(v);
        If Q1⋂Q≠∅ then
          Path:=cpupath();
          return(Path);
        else
        begin
        If ∃w∈Q and w is not in H2 and Stack2
        then
        begin
        If number(Stack1)+number(Stack2)=K
          then
          D-Memtodb();
      Push w into Stack2;
      If w∈V * F then Q1:=Q1⋃{ w };
    End
    Else
    begin
      Pop v from Stack2;
      put v in H2;
      If number(Stack2)=0 then
        D-Dbtomem2();
    End
  End
 End while
End Function
```

图 4　基于动态内存和状态管理的路径搜索

4.2 反例的路径计算

反例的路径 path 由两条子路径组成：从初始状态到所找到的 $V*F$ 中状态的路径 $path_1$ 和从所找到的 $V*F$ 中状态到该状态的回路 $path_2$。$path_1$ 由两部分组成：数据库 DB_1 中有序状态名称序列(从 DB_1 中第一个状态到最后一个状态)和堆栈 $Stack_1$ 中有序状态名称序列(从 $Stack_1$ 中第一个状态到最后一个状态)。而 $path_2$ 也由两部分组成：数据库 DB_2 中有序状态名称序列(从 DB_2 中第一个状态到最后一个状态)和堆栈 $Stack_2$ 中有序状态名称序列(从 $Stack_2$ 中第一个状态到最后一个状态)。相应的算法描述如图 5 所示。

```
Function cuppath()
N11:=number(DB1);
N12:=number(Stack1);
N21:=number(DB2);
N22:=number(Stack2);
  P11:=s1 s2…sN11 (here, s1, …, sN11 are the names of the N11 states
       from the first state to the last state in DB1 respectively);
  P12:=t1 t2…tN12 (here, t1, …, tN12 are the names of the N12 states
       from the bottom state to the top state in Stack1 respectively);
  P21:=u1 u2…uN21 (here, u1, …, uN21 are the names of the N21 states
       from the first state to the last state in DB2 respectively);
  P22:=v1 v2…vN12 (here, v1, …, vN12 are the names of the N22 states
       from the bottom state to the top state in Stack2 respectively);
  Path:=P11 + P12 + P21 + P22;
  Return(Path);
End function
```

图 5 函数 cuppath()的描述

这里，$P_{11}+P_{12}$意味着字符串 P_{12} 连接在字符串 P_{11} 的后面。

5 算法正确性的证明

我们的证明类似文献[14]。由以下几个引理和定理组成。

引理 1 深度优先算法 $dfs1()$和 $dfs2()$采用的是后序遍历。

这个引理是显然的。

引理 2 DB_1 和 $Stack_1$ 中状态构成一条路径。即设 t_1，t_2，…，t_k 是 DB_1 和 $Stack_1$ 中的状态，其中 t_1 是 DB_1 中的第一个状态，t_k 是 $Stack_1$ 中最后一个状态，那么对任意 $i\in\{1, k-1\}$，$(t_i, t_{i+1})\in R$(R 是乘积自动机的状态转移集合)。

根据第 4 节的算法，很容易得到这个结论。同样有下面的引理 3。

引理 3 DB_2 和 $Stack_2$ 中状态构成一条路径。即设 t_1，t_2，…，t_k 是 DB_2 和 $Stack_2$ 中的状态，其中 t_1 是 DB_2 中的第一个状态，t_k 是 $Stack_2$ 中最后一个状态，那么对任意 $i\in\{1, k-1\}$，$(t_i, t_{i+1})\in R$(R 是乘积自动机的状态转移集合)。

将算法中第 k 次调用 $dfs2(\ )$的实参记作 f_k，显然 f_k 是可接受状态。

引理 4 设 $f_1, f_2, \cdots, f_k$是如上定义的状态序列。对任意的 $i<j$，如果存在一条从 f_i 到 f_j 的路径，那么一定存在经过 f_i 的环路。

证明：假设存在一条从 f_i 到 f_j 的路径 R。如果在调用 $dfs2(f_i)$之前，这条路径上没有状态在深度优先搜索 $dfs1()$中标记过，那么在 $dfs1()$中 f_j 应该出现在 f_i 之后，因此 $dfs2(f_j)$应该在 $dfs2(f_i)$之前调用，因此 $j<i$，矛盾。所以 R 上必有某个状态 p 在调用 $dfs2(f_i)$之前被标记过。因此在 $dfs1()$中 p 出现在 f_i 之前，因此在 $dfs1()$中存在一条从 p 到 f_i 的路径，由 R 可知，存在从 f_i 到 p 的路径，所以存在一条从 f_i 到 f_i 的环路。

定理 1 乘积自动机存在一个反例的充分必要条件是算法返回一条经过某个接受状态的路径(该路径由初始状态到该接受状态的子路径和经过该接受状态的环路组成)。

证明：(充分性)假设算法返回一条经过某个接受状态的路径(该路径由初始状态到该接受状态的子路径和经过该接受状态的环路组成)。设环路为 R_1，接受状态为 p，从初始状态达到 p 的路径 R_2，那么显然这条 R_1+R_2 就是一个反例。

(必要性)如果乘积自动机存在一个反例，那么显然乘积自动机存在一条经过某个接受状态的环路，而且存在一条从初始状态达到该接受状态的路径。我们现在证明算法返回一条经过某个接受状态的路径(该路径由初始状态到该接受状态的子路径和经过该接受状态的环路组成)。

假设 f_m 是如上定义的状态序列 $f_1, f_2, \cdots$中第一个属于某个环路的状态(即对任意 $f_i(i<m)$，不存在经过 f_i 的环路)。设这条环路为 R_1(从 f_m 到 f_m 的环路)。R_1 中没有状态是可以从某个 $f_i(i<m)$可达的。假设某个状态 q 是可以从 f_i 可达的，那么从 f_i 也可达到 f_m。由引理 4 知道存在经过 f_i 的环路，这与 f_m 的选取矛盾，所以 R_1 中没有状态是可以从某个 $f_i(i<m)$可达的。因此在第 m 次调用 $dfs2(\ )$(即 $dfs2(f_m)$)时，$R1$ 上的状态都没被第二个深度优先算法 $dfs2(\ )$访问过(即都不在 H_2 中)。因此第 m 次调用 $dfs2(\ )$后，我们能找到一条经过 f_m 的环路 R_1。设在 $dfs1()$中，$dfs2(f_m)$调用之前，算法获得的从初始状态到 f_m 的路径为 R_2，那么算法返回 R_1+R_2，这就是自动机的一个反例。

结束语 本文先采用约简方法(如抽象方法)将系统约简，然后通过内存和状态的恰当管理，提出了基于动态内存和状态管理的模型检测新方法。该方法在任意大小内存的计算机上，能对任意规模的复杂系统进行模型检测。与以往“on the fly”模型检测方法相比，算法的优点主要有：

(1) 通过动态内存调度和状态控制，使得算法在任何计算机上，对于任何规模的系统都能进行模型检测，解决了因内存不足模型检测无法进行的问题；

(2) 内存中的状态整块调度，通过在内存中设置内存缓冲区，解决了内存抖动的问题；

(3) 使用了状态集合 Q_1，使得算法比其它 on the fly 方法更快找到经过接受状态的环路。

参 考 文 献

[1] Clarke E M, Emerson E A. Synthesis of synchronization skeletons for branching time temporal logic[C]// Kozen D. Logic of Programs Workshop. German: Springer-Verlag Lecture Notes in Computer Science, 1981, 131: 52-71

[2] Gueta G, Flanagan C, Yahav E, et al. Cartesian partial-order reduction[C]// Proc. of SPIN Workshop, volume 4695 of LNCS. Spain: Springer, 2007: 95-112

[3] Sistla A P, Godefroid P. Symmetry and reduced symmetry in model checking [C] // CAV, volume 2102 of LNCS. Spain: Springer, 2007: 91-103

[4] Clarke E M, Grumberg O, Long D E. Model checking and abstraction[J]. ACM Transaction on Programming Languages and Systems, 1993, 16(5): 1512-1542

[5] Krimm J P, Mounier L. Compositional state space generation from Lotos programs[C] // Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 1997), volume 1217 of LNCS. Bolin: Springer, 1997: 239-258

[6] Biere A. Efficient μ-calculus model checking[C] // CAV, volume 1254 of LNCS. Spain: Springer, 1997: 468-471

[7] BurchJ R, Clarke E M, McMillan K L. Symbolic model checking: 10^{20} states and beyond[J]. Information and Computation, 1998, 14(2): 142-170

[9] Raimondi F, Lomusico A. Automatic verification of multi-agent systems by model checking via OBDDs[J]. Journal of Applied Logic, 2005, 21(3): 125-137

[10] Su Kai-le. Model Checking Temporal Logics of Knowledge in Distributed Systems[C] // The Nineteenth National Conference on Artificial Intelligence (AAAI-04). America: Springer, 2004: 354-368

[11] Jard C, Jeron T. Bounded-memory Algorithms for Verification On-the-fly[C] // 3rd Workshop on Computer-Aided Verification. America: Springer, 1991: 210-230

[12] Gerth R, Pled D, Vardi M Y, et al. Simple on-the-fly automatic verification of linear temporal logic[C] // the Conference of Protocol Specification Testing and Verification. Chapman Chapman & Hall: Springer, 1995: 3-18

[13] Clarke E M, Orna Grumberg and Doron A. Peled. Model Checking[C] // 1th Conference of Model Checking. Cambridge: MIT Press, 1999: 121-140

[14] Courcoubetis C, Vardi M Y, Wolper P, et al. Memory efficient algorithms for verification of temporal properties[C] // The Conference of Formal Methods in System Design. German: Springer, 1990: 275-288

(上接第 175 页)

划分下的数量关系，却又可以超越时态元仅有正值的内涵，使得时态跨度之间的运算以及时态关系演算等变得复杂。但由于时态粒度全集中往往包含弹性粒度，使得构成时态跨度的各时态元之间难以或甚至不能进行粒度转换处理，从而使时态跨度之间缺乏有效且准确的演算方法。本文通过对时态粒度系统做向量化处理，使其同构到 n 维向量空间，并将时态跨度做完备化和平滑化处理后，映射为向量空间中的自由向量。同时证明了这种处理方式能够保证时态跨度的语义及数值意义，从而可以通过向量间的运算法则以及性质，简单有效地处理各种复杂组合形式下的时态跨度间的运算。不过，鉴于时态跨度的复杂性，对于时态跨度之间的确切时态关系，必须依赖于具体绑定后的运算，限于篇幅，将另文讨论。

参 考 文 献

[1] Lin T Y. Granular Computing: From Rough Sets and Neighborhood Systems to Information Granulation and Computing in Words [C] // European Congress on Intelligent Techniques and Soft Computing. Aachen, Germany, 1997: 1602-1606

[2] Zadeh, et al. Granular Computing and Rough Set Theory [J]. Lecture Notes in Computer Science, 2007, 4585: 1-4

[3] 左亚尧，汤庸，舒忠梅，等. 时态的粒度刻画及演算问题研究[J]. 计算机科学，2010，37(12)：114-119

[4] Pfennigschmidt S, Voisard A. Handling Temporal Granularity in Mobile Services [C] // IEEE International Conference on Wireless and Mobile Computing, Networking and Communications. Marrakech, 2009: 295-300

[5] Zhang Xue-jie, Ng K-W. A temporal partitioning approach based on reconfi-guration granularity estimation for dynamically configurable systems[C] // IEEE International Conference on Field-Programmable Technology. Tokyo, 2003: 344-347

[6] 万静，郝忠孝. 具有多时间粒度的强全序时态模式中多值依赖问题研究[J]. 计算机研究与发展，2008，45(6)：1064-1071

[7] Egidi L, Terenziani P. A flexible approach to user-defined symbolic granularities in temporal databases [C] // Proceedings of the 2005 ACM Symposium on Applied Computing. New Mexico, 2005: 592-597

[8] Merlo I, Bertino E, Ferrari E, et al. Querying multiple temporal granularity data [C] // Seventh International Workshop on Temporal Representation and Reasoning (TIME 2000). July 2000: 103-114

[9] Bertino E, Ferrari E, Guerrini G, et al. Navigating through multiple temporal granularity objects [C] // Eighth International Symposium on Temporal Representation and Reasoning. Italy, 2001: 47-155

[10] Belussi A, Combi C, Pozzani G. Formal and conceptual modeling of spatio-temporal granularities [C] // Proceedings of the 2009 International Database Engineering & Applications Symposium (IDEAS '09). Italy: ACM, 2009: 275-283

[11] Orgun M A, Liu Chu-chang, Nayak A C. Representation and Integration of Knowledge Based on Multiple Granularity of Time Using Temporal Logic[C] // IEEE International Conference on Information Reuse and Integration. Hawaii, 2006: 256-261

[12] Cotofrei P, Stoffel K. Temporal granular logic for temporal data mining[C] // IEEE International Conference on Granular Computing. Beijing, 2005: 417-422

[13] Chen Xiao-qing, Qiu Tao-rong, Liu Qing, et al. The Framework of Temporal Granular Logic Based on Information System[C] // IEEE GrC 2006. Atlanta, 2006: 604-606

[14] Goralwalla I A, Leontiev Y, et al. Temporal granularity for unanchored temporal data[C] // ACM Proceedings of the Seventh International Conference on Information and Knowledge Management (CIKM1998). Bethesda MD USA, 1998: 414-423