基于 LTL 语义的可达性控制器合成工具①

景丽莎 [1,2], 项周坤 [1,3]

[1](中国科学院软件研究所 计算机科学国家重点实验室, 北京 100190)
[2](中国科学院 研究生院, 北京 100049)
[3](苏州大学 计算机学院, 苏州 215000)

摘 要: 控制器合成是针对给定的获胜目标, 在开放的实时系统环境中, 自动地寻找获胜策略的过程. 这个策略可以表述为一系列的符号化状态和动作的映射关系. 在本文中, 我们主要针对以线性时序逻辑(LTL)描述的可达性作为获胜目标, 进行合成策略的发现. 文中介绍了一种采用 on-the-fly 思路的合成算法, 以规避状态数目太多带来的内存溢出问题. 文中算法是对文献[1]的一种扩展, 该算法主要用于解决基于分支时序逻辑(CTL)的控制器合成. 另外, 我们实现了相关的控制器合成工具 CTAV/TGA(Timed Gamed Automata), 在实现的过程中, 使用 on-the-fly 的方式, 避免了穷尽状态空间, 同时, 通过使用 zone 和抽象, 大大缩减了状态数目, 使时空效率控制在可接受的范围内.

关键词: 时间博弈自动机; 控制器合成; 线性时序逻辑(LTL); on-the-fly; 符号化方法

Tool for Controller Synthesis Based on LTL Reachability

JING Li-Sha[1,2], XIANG Zhou-Kun[1,3]

[1](State Key Lab of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)
[2](Graduate University, Chinese Academy of Sciences, Beijing 100190, China)
[3](School of Computer Science, Soochow University, Suzhou 215000, China)

Abstract: Controller synthesis is the automatic process of searching for a winning strategy in an open real-time system on the basis of a given winning goal. This strategy can be expressed as a series of mapping relations of the symbolic states and the symbolic actions. In this paper, we focus on the reachability target which is described by Linear Temporal Logic(LTL) to discover if there is a synthetic strategy. This paper introduces a synthesis algorithm using on-the-fly method to avoid too many states. The algorithm is an extension of Ref. [1], which is mainly used to solve the problem based on branching temporal logic (CTL) for controller synthesis. Through the use of zone, our algorithm is greatly reducing the number of states, and the efficiency is acceptable.

Key words: timed game automata; controller synthesis; linear temporal logic (LTL); on-the-fly; symbolic method

实时系统所描述的是一种可以短时间内迅速响应外部发生的事件, 并可在一定的时间范围内做出应答、给予处理的应用系统. 保证实时系统设计和运行的正确性和可靠性至关重要, 在 20 世纪 90 年代前后, 模型检测等相关的形式化理论被广泛应用在实时系统可靠性和正确性的检测上[4,11]. 但是通常来说, 模型检测所针对的实时系统, 其运行状态只由系统内部迁移决定. 而实际生活中, 实时系统往往并不是封闭的, 其运行状态不仅仅取决于自身运行的结果, 环境因素往往也直接或间接地影响着系统的整体状况. 因此, 如何在外界环境因素的作用下, 依然保证系统的可靠性和正确性成为值得探究的问题.

控制器合成问题, 就是为了在开放的实时系统中, 寻找获胜目标的合成策略而产生的. 它可以被形式化地表述成两个选手的博弈问题[2,3,6,7]. 其中, 系统的控制器本身, 是一个迁移系统, 用来代表一个参赛选手, 而环境, 也是一个迁移系统, 用来代表另一个参赛选手. 控制器合成就是在这样的前提下, 寻找一个策略 f 使得环境无论如何变化迁移, 控制器总可以满足给定的性质.

我们的工作是在开放的实时系统中上, 针对 LTL

① 收稿时间:2016-03-11;收到修改稿时间:2016-04-05 [doi:10.15888/j.cnki.csa.005434]

描述语言所描述的系统性质进行控制器合成算法和相关工具的研究. LTL 相较于 CTL 而言, 在描述与活性相关的性质方面具有一定的优势. 为了针对 LTL 性质表述的获胜目实现标控制器合成的过程, 需要将 LTL 性质转化为 Rabin 自动机, 并通过自动机的复合来达成验证的目标. 当前, 已有工具可使我们从复杂的 LTL 公式中通过计算得到它对应的 Rabin 自动机[13]. 我们知道, 假设有命题 p, 对于 Rabin 自动机来说, 其接收条件将$<>[]p$ 和$[]<>p$ 作为复杂性质基础. 这两种性质则是以$<>p$和$[]p$为实现基础的. 为了在之后的工作中, 完善工具对于性质的表达能力, 使其所能表达的各种复杂性质均得以描述, 应首先着手对$<>p$、$[]p$、$<>[]p$ 和$[]<>p$ 这四种最为基本的性质内容予以研究和讨论. 而$<>p$ 则是全部四种性质研究的起点.

因此, 在本文中, 我们主要讨论的是控制器合成针对于以$<>p$ 为合成目标的合成算法及对应的工具. 对于可达性而言, 合成算法就是总可以在系统的形式化模型中, 找到一个策略, 使得 LTL 所描述的目标可以被到达[1].

时间自动机常被用作刻画实时系统的形式化模型. 时间博弈自动机, 在其基础上增加了对环境因素的描述, 通过对系统可控因素和环境不可控因素的描述, 可用来刻画开放的实时系统. 对于系统的获胜目标, 可使用逻辑语言描述, 主要有分支时序逻辑 CTL 和线性时序逻辑 LTL. Uppaal-Tiga[12]就是一个时间博弈自动机的合成工具, 它是对 UPPAAL 工具的扩展, 对获胜目标主要是针对分支时序逻辑CTL作为描述语言描述的. 本文的算法也是在其基础上进行的扩展与改进.

本文的第一部分介绍控制器合成理论的一些基础的概念原理和符号, 第二部分介绍了控制器合成工具的实现过程, 合成算法的详细介绍和实现思路在第三部分提及, 第四部分是相应的实验结果, 最后在文章的结尾给出了全文的总结, 并提出今后的工作方向.

1 基本概念和相关原理

1.1 开放的实时系统和时间博弈自动机

时间自动机常常被用作实时系统的抽象模型, 它可以看做是一个拥有节点和迁移的有向图. 而开放的实时系统其运行的正确性和可靠性会受到外界环境因素的影响, 为了对其进行形式化建模, 将时间自动机的迁移引入可控因素和不可控因素, 这也就是时间博弈自动机.

图 1 所示表示了工厂机器一次加工的过程. 工厂中的机器在加工一件产品完毕后若不及时取出, 将很可能造成机器卡死的情况. 图中, 机器加工完毕一件成品后, 若在固定时间内没有被取出, 则可能面临崩溃的危险, 这一点是不可控的, 因而在图中使用虚线来表示.

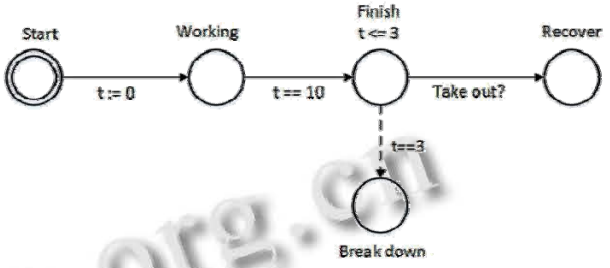

图 1 机器 Machine 加工一次成品的过程

若此时, 工人可以及时取出产品, 则机器正常运转不会故障. 工人的行为同样可以形式化为图 2 的自动机.

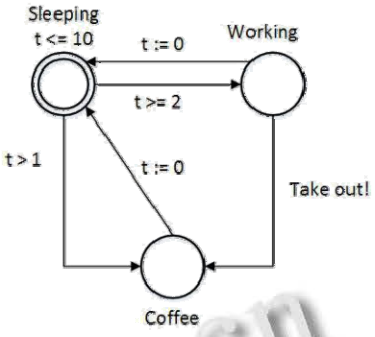

图 2 工人 Worker 活动的自动机

时间博弈自动机可以形式化的用一个六元组$M=(L, l_0, Inv, Act, X, E)$来表示[5,6], 其中:

(1) L 是一个有限集合, 它包含了时间博弈自动机 M 的所有位置节点.

(2) X 也是一个有限集合, 它包含了所有的为时钟变量.

(3) $l_0 \in L$ 是时间博弈自动机 M 的初始位置节点.

(4) Inv 是一个映射函数, 为每个位置节点指明了时钟约束, 称为位置的不变式.

(5) $Act = Act_c \cup Act_u$, 是动作集合, 包含了可控动作集合 Act_c 和不可控动作集合 Act_u.

(6) $E \subseteq (L \times Act \times g \times \mathrm{Re}set \times L)$ 是迁移的集合. 其中, $g = x \odot c \mid x_1 - x_2 \odot c \mid g_1 \wedge g_2$, 均为时钟变量, $c \in N$ 为一个常量, $\odot \in \{<,>,\geq,\leq\}$为逻辑运算符, g_1 和 g_2 都是时钟约束. $Reset$ 表示时钟重置集. 迁移 $e \in E$ 表示节点

在满足条件 g 的约束下, 通过将时钟重置的条件, 迁移至它的后继状态, 与此同时, 对应的 *Reset* 集合中的时钟被重置.

1.2 时间博弈自动机的获胜策略及获胜目标

开放的实时系统的行为可以描述在时间博弈自动机的节点位置上进行迁移. 系统运行的状态既包括了系统的位置节点信息, 也包括有运行的时间信息, 即 $S=L\times R^n_{\geq 0}$[1,7,10], 其中 n 为时钟变量的个数. 环境因素所采取的动作可表示为迁移集合 Act_u, 系统本身可控的动作表示为迁移集合 Act_c. 在 $t\in R^n_{\geq 0}$ 这个时刻, 将环境和控制器看做是博弈的双方选手, 那么这个选手 P 所可能采取的动作具体而言有两种可选择的方案:

(1) 设 v 为时钟变量到实数向量的映射, 记为 $v: X\text{-}> R^n_{\geq 0}$, 在满足 $v\mid= g$ 并且对于迁移 $e = (l, \alpha, g, Reset, l')$ 中所涉及的动作, 有 $\alpha\in\alpha_P$. 并且, $t\ [Reset]\ \mid= Inv(l')$, 即在经过时钟重置后时钟条件满足节点位置 l' 的时钟约束, 则发生 e 所指示的迁移.

(2) 不发生任何动作, 只是等待, 此时所有的时钟变量的值将会以相同速度不断增长.

上文所提到的 P 可以是外部环境也可以是系统控制器自身. 但是相较于后者, 环境具有发生迁移的优先权, 也就是说, 当外部环境和系统控制器同时有可能执行迁移的时候, 外部环境的迁移可以优先发生[8,9].

对于开放的实时系统, 其控制器获胜, 在本文中可以表述为无论外界的环境因素如何变化, 系统运行的状态会改变, 但系统始终满足给定的 LTL 性质这个结论却不会因此而改变.

系统运行的状态既包括了系统的位置节点信息, 也包括有运行的时间信息, 即 $S=L\times R^n_{\geq 0}$, 其中时间约束使用 Zone 表示. 对于一个没有记忆能力的获胜策略而言, 合成策略的制定应仅与当前系统所处的状态 $S=L\times R^n_{\geq 0}$ 有关. 对选手 P 所采取的策略可以描述为右边的偏函数: $f_P: S=L\times R^n_{\geq 0}\text{-}> Act_P\cup\{\lambda\}$, 即:

(1) 对 P 来说, 当处于系统状态 $s\in S$ 时, 由 s 出发的迁移 e 上有 $\alpha\in Act_P$ 动作, 若 P 必须要进行迁移的话, 则有 $f_P(s)=\alpha$, 即 P 在 s 状态上, 采取了动作 α.

(2) 或者与上述不同, P 在这一时刻, 可以选择不去迁移. 那么此时的策略将制定为: 任由时间发生延迟迁移, 但是 P 保持在此状态, 即 $f_P(s)=\lambda$.

本文中主要研究的是使用 LTL 表示的可达性问题, 因而使用 *Goal* 来描述自动机中的节点位置集合, 则有系统控制器可通过一系列的策略, 令系统最终到达 *Goal* 集合, 即令<>*Goal* 为真则可说明控制器获胜. 对于上述的工厂问题, 其最终目标是要到达 Recover 的状态, 可以使用 LTL 性质将其描述为: <>*Machine.Recover*.

2 控制器合成工具的实现原理

控制器合成工具就是基于上述两点, 在针对开放的实时系统所构建的时间博弈自动机模型上, 通过不断对时间博弈自动机的状态空间进行探索的同时, 以不同的算法规则对满足条件的路径回溯, 从而最终判断对获胜目标是否可以到达. 整个过程包含了前向和后向的两个过程.

利用控制器合成工具进行合成策略的寻找, 一般而言需要经过以下的几个阶段:

(1) 模型建立阶段. 对所有待检测的开放实时系统进行深入地分析, 形式化地抽象并为之建立对应的时间博弈自动机模型.

(2) 描述获胜目标. 使用特定的逻辑描述语言, 对获胜目标进行刻画. 获胜目标实际上就是一个命题, 为真的话则说明性质得以满足.

(3) 合成阶段. 合成算法是整个工具的核心, 依据不同的获胜目标, 工具将执行不同的算法过程, 从而验证待检验的实时系统是否可以满足获胜目标.

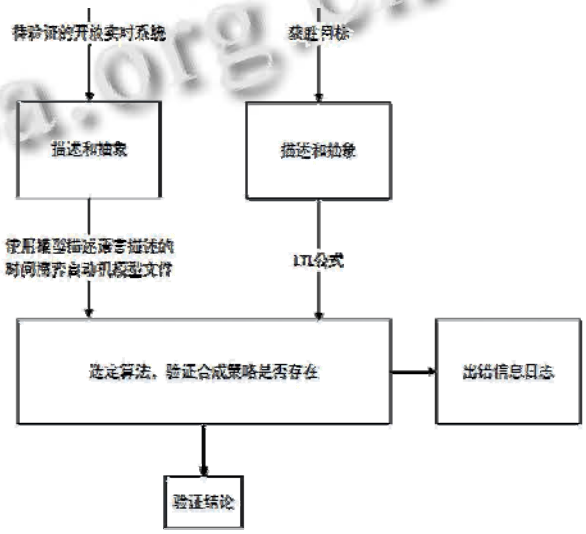

图 3　控制器合成过程

图 3 中详细说明了本文中进行控制器合成的各个阶段及其所对应的产物.

3 基于LTL可达性的控制器合成算法

系统运行的每个符号化状态包含有位置信息和时钟信息两部分, 对于每一个状态 S, 可以定义其结构为表 1 中包含位置和时钟信息的节点结构.

表 1 符号化状态节点结构

```
typedef struct SymbolicState{
    LocIndexVec;    //当前位置的索引向量
    ClockInfo;      //时钟信息, Zone
}
```

除了上述的时钟信息, 对于系统运行中的每个状态 S, 为之设定一个对应的 DBM: S_reach_dbm, 用来表示状态所在的位置 S.LocIndexVec, 只有处于时间变量满足 S_reach_dbm 约束的前提下, 才能够到达获胜目标. 在实现时, 为方便查找, 使用哈希结构来存储这样的映射关系.

算法的整个过程可以简单的描述为状态空间展开和状态空间回溯两个部分, 这两个部分交替进行. 每当遇到已经访问过的状态时, 则会引发一次回溯, 回溯的过程, 会修改状态所对应的 reach_dbm.

只有当初始状态的 reach_dbm 包含起始时刻时, 才表明由初始状态出发, 可以到达给定的获胜目标.

同样地, 当所有可用状态都已穷尽而 reach_dbm 依旧不包含起始状态时, 算法过程也会结束, 此时说明系统无法通过控制器的控制到达获胜目标, 也就间接说明, 控制策略并不存在.

需要注意的是以下图中的几种情况, 考虑图 4 中六种情况的时间博弈自动机, 可以很容易的判断出, 对于获胜目标“<>Good”:(a)在当系统处于 Working 位置的时候, 控制系统在 t<=3 时经可控迁移至 Good 位置; (b)、(c)和(e)与(a)的控制策略相同, 只是(b)(e)Working 位置没有不变式约束, 当其处于 Working 位置时由控制器在 t<=3 时间范围内, 控制可控迁移发生, 而(c)Start 位置的不变式约束则保证了系统运行过程中只能停留于此 5 个时间单位, 则必须经由不可控迁移至 Working 位置, 在迁移的同时将时钟变量置 0, 此时可由控制器确保进入 Good 状态. 而(d)以 Start 为源位置状态的迁移其发生是不可控的, 即若在 3<t<=5 时此迁移发生, 则不能保证会到达 Good 位置, 所以(d)不满足获胜目标.

对于图 5 中的三个例子, (f)在处于 Working 位置时, 因不可控迁移优先发生, 所以不满足<>Good 的获胜目标. (h)Working 位置因没有不变式约束, 则即使控制通往 Bad 的迁移不发生, 也可能会导致系统一直停留在 Working 位置而不能到达 Good, 故而也不满足获胜目标. (g)在处于 Working 位置时, 因不变式约束, 至多可以停留三个时间单位, 我们令控制器控制系统通往 Bad 的迁移不发生, 则不可控迁移必须发生, 到达 Good.

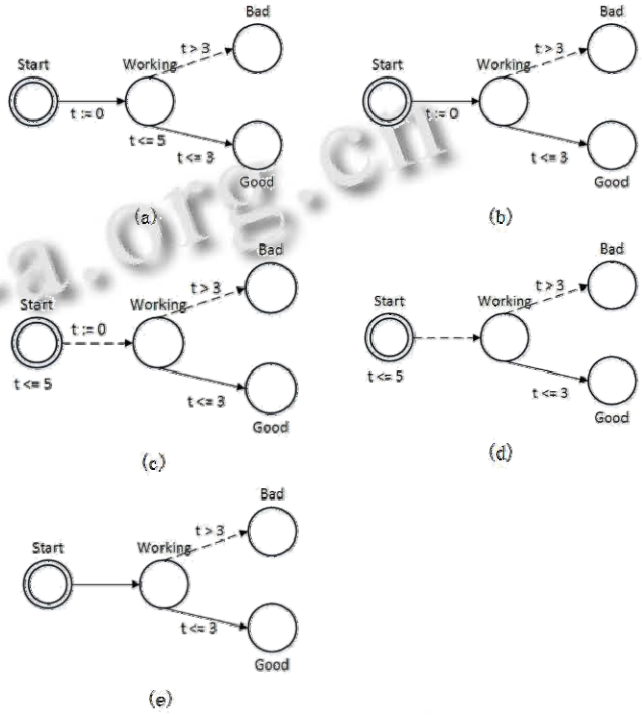

图 4 几个简单的时间博弈自动机的例子

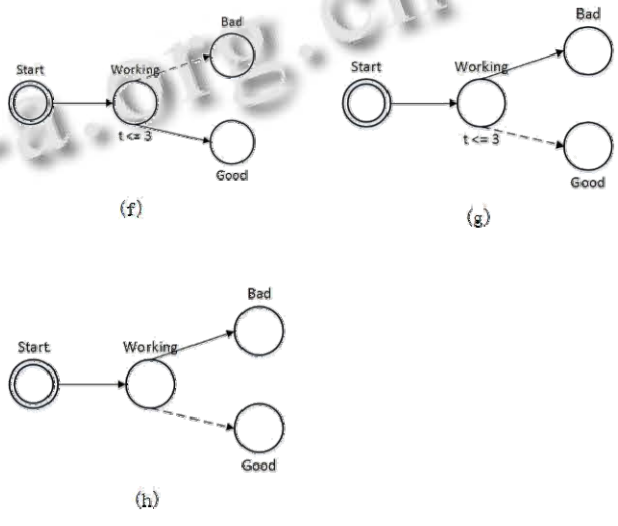

图 5 几个简单的时间博弈自动机的例子(续)

在上述的例子中, 若不可控迁移的源状态没有不变式限制, 将可能导致系统运行停滞在此状态, 当有不变式约束存在时, 可以通过合成策略令系统控制器

在这一时刻始终处于等待的状态, 一旦条件满足, 则不得不发生迁移, 从而强制不可控迁移发生, 进而到达获胜目标. 这一点在实现上与 UPPAAL-TIGA 的思路有所不同, 我们补充了在回溯过程中对前驱的计算内容, 补充了 Predt 算子的第一个参数值的内容. 文献[1]中有关于这个算子的介绍, 对于 Predt(X, Y), 其运算的结果可以描述为: 求得在 δ 时间范围内可以迁移到集合 X 中所有状态的前驱集合, 并在这些前驱中删除掉可以在 δ 时间范围内可以迁移到集合 Y 所包含的状态的那些状态, 经过这些运算后所剩余的状态, 即为结果状态集.

以下, 将具体说明算法中补充的内容. 对状态 S, 所有可以经由不可控迁移到达的状态 T, 将计算与之相对应的 Preu(T_reach_dbm). 其中, T_reach_dbm 指的是当系统处于在状态 T 所对应的位置向量时, 若时间变量满足此约束时, 系统可以由当前位置到达获胜目标. 这样我们可以通过控制可控迁移始终不发生来强迫不可控迁移发生. 但并非所有的通过不可控迁移计算而来的 Preu(T_reach_dbm)都有效, 必须满足下面的两个条件, 才可以补充至 Predt 中:

(1) 对应于这个不可控迁移的源位置状态 S 上, 必须具有不变式约束, 否则系统存在停滞在这个位置的可能性;

(2) 经过计算后的 Preu(T_reach_dbm)其时钟范围需包含这个不可控迁移的源位置状态 S 不变式约束的临界值.

实际上, 大多系统并不如图 4 和图 5 所示进程中只包含一个进程的运行, 所以这里这里所提到的每一个可控和不可控迁移都对应于所有进程中的某一个位置出发的一条迁移.

为了清晰地展示扩展后的算法过程, 除了上面提到的 State 外, 我们还将使用到如下的数据结构:

(1) VisitedStates: 已经被遍历过的状态集合;

(2) SymbolTrans: 这是一个存储符号化迁移的边, SymbolTransStack 表示所有待遍历的边, 以栈的方式存储, 也是借助这个栈不断地入栈和出栈, 从而实现状态空间的遍历. SymbolTransSet 为待遍历的边的集合, 它是后文中 EnterTrans 的辅助数据结构.

(3) Hash: 其元素为(State,reach_dbm)的哈希表, 这里 State 所对应的 reach_dbm, 事实上是一个 dbm 类型的联合, 在计算过程中, 它可能是多个 dbm 的并集.

(4) EnterTrans: 是元素为(State,SymbolTransSet)的哈希表, 对应地记录了所有以 State 状态为目标节点的符号化迁移, 为集合形式, 在遍历的过程中, 这个集合被不断的扩大.

另外, 算法中我们使用了 DBM 库的一些函数, 下面对其中最主要是 Prec(S)和 Preu(S), 分别表示能够通过可控迁移到达状态 S 的前驱状态的 DBM 集合和通过不可控迁移到达状态 S 的前驱状态的 DBM 集合. Predt(X,Y)的含义在上文已经解释过, 它在 DBM 库里也有对应的函数实现, 可以直接使用.

算法的伪代码列在如下的表 2 和表 3 中, 其中, 前者为整个合成算法的初始化阶段, 后者是合成算法的整个执行过程. 表 4 是算法过程终止后的处理方案.

表 2 合成算法的初始化阶段

```
VisitedStates.add(S0);
对所有由 S0 出发的符号化迁移(SymbolTrans   st){
    SymbolTransStack.push_back(st);
    }
EnterTrans(S').setEmpty();
if(S0 ∈ Goal)
    return; //找到策略
else
     Hash(S0) = dbm.setEmpty();
    //将初始状态的 reach_dbm 置空
```

表 3 算法向前探索状态空间和向后回溯的过程

```
while(!SymbolTransStack.isEmpty()
&& !Hash(S0).intersect(dbm.setZero())){
    st = SymbolTransStack.pop_back();
    //弹出符号化迁移, S 为 st 源节点, S'为目标节点
    if(S'∈Visited){
    //目标节点尚未被访问
    VisitedStates.add(S');
    EnterTrans(S').add(st);
    if(S'Ï Goal){
       Hash(S') = dbm.setEmpty; //reach_dbm 置空
    }
    else{
    Hash(S') = S'.ClockInfo;
    SymbolTransStack.push_back(st);//入栈,引发回溯
    }
    对 S'出发的符号化迁移(SymbolTrans new_st){
```

```
    SymbolTransStack.push_back(new_st);
    }
    }
    else{
    //目标节点已经被访问
    for(对 S 可以到达的每一个后继 T){
    result_fed_t_c |= Pre_c(Hash(T));
    if(状态 S 上的不变式约束不为空){
    result_fed_t_i |=Pre_u(Hash(T));
    Result_fed_t_i =result_fed_t_i∩S 的不变式;
    }
    else{
    //状态 S 的不变式约束为空
    result_fed_t_i = dbm.setZero(); 置空
    }
    result_fed_t_u |=Pre_u(T.ClockInfo - Hash(T));
    }//end of for
  result_fed_t_c |= Hash(S); //和 S_reach_dbm 取并
  result_fed_t_c|=result_fed_t_i; //result_fed_t_c 被扩大
  S_reach_dbm_tmp = Pred_t ( result_fed_t_c, result_fed_t_u) &
S.ClockInfo;
    //使用 Pred_t 得出的结果需满足 S 状态本身的时钟约束才是有效的,
因而此处取二者的交集
  if(Hash(S) ⊂ S_reach_dbm_tmp){
  Hash(S) = S_reach_dbm_tmp;
  //若 S_reach_dbm 为新集合的子集, 此处更新
将 EnterTrans(S)中所有的迁移加入 SymbolTransStack
    //持续回溯, 将此次更新的结果向前传递
  }
}
```

最后, 该合成算法将在遇到以下的条件后终止, (1)初始状态的reach_dbm包含了初始时刻;(2)所有待检测的边已经穷尽.

伪代码如表 4 中所示.

表 4 合成算法结束的条件

```
if(Hash(S0).intersect(dbm.setZero())){
    return; //获胜策略存在
    }
if(SymbolTransStack.isEmpty() ){
    return; //遍历完毕, 未到达获胜目标
    }
```

4 实验研究

本文中实现了一个基于 LTL 性质的可达性控制器合成工具 CTAV/TGA. 可达性作为 LTL 语言所能描述的基础性质, 是实现以 LTL 性质为获胜目标的控制器合成工具的基础. 在模型描述语言的选择上, 为了更加方便合成算法和系统方案的实现, 本工具选取了 UPPAAL 的模型描述语言, 并使用 UTAP 库对系统模型予以解析.

以下将使用文章开头的工厂的例子来予以测试和比较. 其中 factory01 即为开头的例子,factory02 将工人模型从 Sleeping 到 Working 的迁移和加工零件模型的 Working 到 Finish 的迁移变更为不可控的迁移类型. 下划线后的数字表示当前进程中加工零件模型的实例. 如: factory01_5 表明当前的实验结果针对的是 factory01 这个例子, 当前有 5 个加工零件的实例.

表 5 实验运行结果比较

例子	CTAV/TGA		Uppaal-Tiga	
	检验结果	运行时间	检验结果	运行时间
factory01_1.xml	Yes	0.003s	Yes	0.004s
factory01_2.xml	Yes	0.005s	Yes	0.006s
factory01_3.xml	Yes	0.64s	Yes	0.014s
factory02_1.xml	No	0.002s	No	0.008s
factory02_2.xml	No	0.336s	No	0.018s
factory02_3.xml	No	0.83s	No	0.034s

从以上实验结果的对比中可以看出, CTAV/TGA 的对于文中的测试用例所给出的检验结果与 Uppaal-Tiga 的检验结果相吻合. 在对算法执行的过程进行分析时, 我们发现, 可达性合成算法主要的运行时间集中在空间展开和对已经遍历过的状态进行回溯这两部分. 算法过程中, 每增加一个状态, 其回溯所带来的时间耗费就相当可观, 可导致整个算法出现指数级别的计算量的增加. 因而如何避免减少无用状态数目的增加可以作为优化的一个思路. 可以看出, CTAV/TGA 的合成算法在进程数目到达 3 的时候, 运行时间有所增加, 相较于 Uppaal-Tiga, 我们在回溯过程中对算子的修改的确会带来一定程度的时间效率的

降低. 但这并不是导致算法时间增加的主要原因. 考虑到对于时间博弈自动机而言, 符号化状态是显示状态的集合, 这些集合之间蕴含着包含关系, 可以借鉴 Uppaal-Tiga 的思路在判断是否需要继续展开状态时, 进行一些比较, 从文献[1]的结果中可以看出, 这种优化的方法是很有效的, 这也将是后续要进行讨论和研究的地方.

5 总结

本文中描述了一个将 LTL 表达的可达性作为获胜目标的控制器合成工具 CTAV/TGA, 讨论并实现了可达性的控制器合成算法, 其中, 依据时间博弈自动机的概念, 对算法的一些情况给予补充. 通过使用借助 DBM 表示的符号化状态和 on-the-fly 的检测思路, 对工具的合成算法的效率提升有很大的帮助. 需要指出的是, 目前 CTAV/TGA 所能够合成的获胜目标性质种类还比较有限, 其中算法的改进仍有空间. 所以未来我们的工作还很多, 下一步将需要对更为复杂的 LTL 性质进行分析, 来进一步完善这个工具.

参考文献

1 Cassez F, David A, Fleury E, et al. Efficient on-the-fly algorithms for the analysis of timed games. CONCUR 2005-Concurrency Theory. Springer Berlin Heidelberg, 2005: 66–80.

2 Bulychev P, David A, Larsen KG, et al. Efficient controller synthesis for a fragment of MTL0,∞. Acta Informatica, 2013, 51(34): 165–192.

3 Peter H, Ehlers R, Mattmüller R. Synthia: Verification and synthesis for timed automata. Computer Aided Verification. Springer Berlin Heidelberg, 2011: 649–655.

4 David A, Behrmann G, Bulychev P, et al. Tools for model-checking timed systems. London, UK: ISTE. Hoboken, NJ: Wiley, 2013: 165–225.

5 Altisen K, Tripakis S. Tools for controller synthesis of timed systems. Proc. Work. on Real, 2002.

6 Alur R, Dill DL. A theory of timed automata. Theoretical Computer Science, 1994, 126(94): 183–235.

7 Maler O, Pnueli A, Sifakis J. On the synthesis of discrete controllers for timed systems. Lecture Notes in Computer Science, 1995, 900(6): 229–242.

8 Asarin E, Maler O. As soon as possible: Time optimal control for timed automata. Proc. of the Second International Workshop on Hybrid Systems: Computation and Control. Springer-Verlag. 1999. 19–30.

9 Alur R, Torre SL, Pappas GJ. Optimal paths in weighted timed automata. Theoretical Computer Science, 2004, 318(3): 297–322.

10 Asarin E, Maler O, Pnueli A. Symbolic controller synthesis for discrete and timed systems. Hybrid Systems II. Springer Berlin Heidelberg, 1999: 1–20.

11 David A, Hakansson J, Larsen K G, et al. Model checking timed automata with priorities using DBM subtraction. Lecture Notes in Computer Science, 2006, 4202: 128–142.

12 Jessen JJ, Rasmussen JI, Larsen KG, et al. Guided controller synthesis for climate controller using uppaal tiga. Lecture Notes in Computer Science, 2007, 4763: 227–240.

13 ltl2dstar-LTL to deterministic Streett and Rabin automata, http://www.ltl2dstar.de/.