有限状态迁移系统

程序设计上机实习报告

陈奕行 1700010780

陈贵显 1700010785

吴雨杭 1700010698

目录

PART Ⅰ
线性时序公式（LTL）的检验

1.前言

当今社会越来越依赖专用的计算机和软件系统来为我们生活的方方面面提供帮助。面对计算机系统逐渐增加的复杂性，模型检验（*Model Checking*）已成为计算机科学发展的当务之急。根据著名的停机问题（*Halting Problem*），一般的检验算法并不存在。在本部分中，我们对有限状态迁移系统（*Finite Transition System*）上的时序逻辑（*Linear Temporal Logic*），并给出相应算法的证明及复杂度分析。

2.问题分析和系统整体设计

建立好数据结构之后，先构造 *LTL* 公式对应的 *Generalized Büchi automaton*（下简称 *GBA*），再依赖此 *GBA* 构建 *Nondeterministic Büchi automaton*（下简称 *NBA*）。利用 *TS* 及 *NBA* 构造 *Product transition system*，转化为此 *TS* 上的 *LTL* 公式检验。这里由于新构造 *LTL* 公式的特殊性，可以利用 *DFS* 最终解决并输出反例。大体可以使用下图概括。

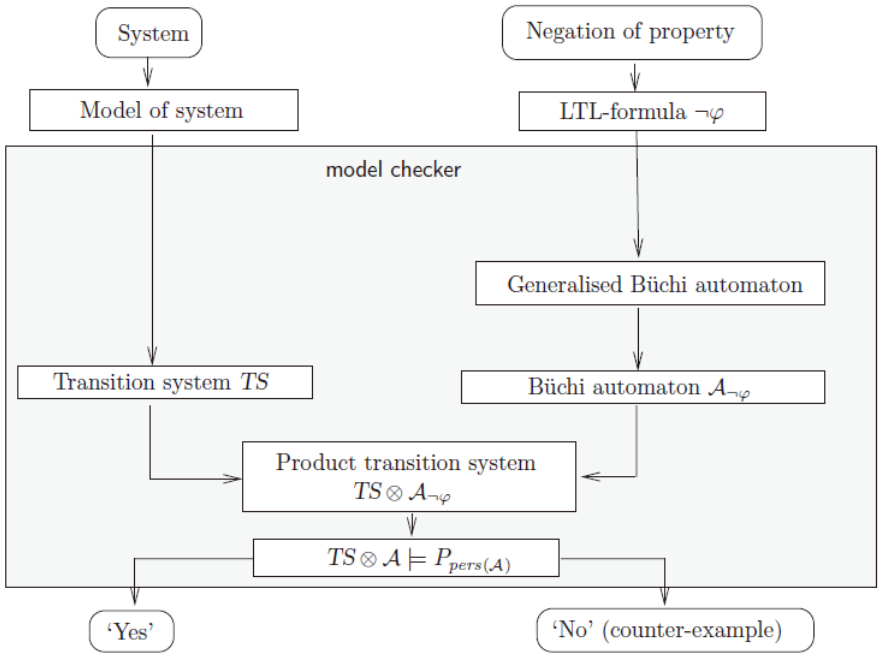


Figure 5.16: Overview of LTL model checking.

3.数据结构设计

3.1 有限状态迁移系统

有限状态迁移系统的定义如下：

A *transition system* TS is a tuple $(S, Act, \rightarrow, I, AP, L)$ where

- S is a set of states,
- Act is a set of actions,
- $\longrightarrow \subseteq S \times Act \times S$ is a transition relation,
- $I \subseteq S$ is a set of initial states,
- AP is a set of atomic propositions, and
- $L : S \rightarrow 2^{AP}$ is a labeling function.

TS is called *finite* if S, Act, and AP are finite.

其中 *AP* 表示对于状态 *S* 的一种描述，而 *L* 则代表每个状态到其描述的映射。因而这里的论断是建立在状态的性质之上，进行了一层抽象。而本题中并未给状态规定泛性质，故可取 *L* 为典范映射，即 *AP* 在不考虑具体数据结构的时候认为与 *S* 相同。

此外，*TS* 在遗忘其上附加结构时可以看成一个有向图。即状态为顶点，箭头表示有向边。这实际可以视为范畴 *TS* 至有向图范畴的遗忘函子，记为 *Forget*。

由于 *TS* 在此后不会发生变化，我们把所有的集合用元组储存。具体而言，初始化函数如下：

```
class TransitionSystem:
    def __init__(self, S, Act, Arrow, I, AP, L):
        self._S = tuple(S)
        self._Act = tuple(Act)
        self._Arrow = tuple(Arrow)
        self._I = tuple(I)
        self._AP = tuple(AP)
        self._L = L
```

定义方法 *input* 表示从文件读取信息来构建此类。

我们不妨可假设 *TS* 没有终状态。若不然，可加入一状态 *Terminal*，在一切原来终状态与 *Terminal* 之间构造一个指向 *Terminal* 的边，以及一条由 *Terminal* 指向自己的边。并且，可以假设这一个新状态不满足任何论断，避免对之前的判断造成影

响。在程序中使用 $complete$ 方法完成，且最后新添加的状态为 $None$，新加入边为‘ ’。这样子不会增加 AP。

为以后论述方便，特定义以下几个记号：

（1） $Post(s)$表示状态 s下一步可能达到的状态。

（2） $\sigma \in (2^{AP})^{\omega}$表示一条无限长状态转移路径。σ[j]表示此路径从第 j+1 项开始的部分。

（3） 定义状态$s \vDash \phi \Leftrightarrow L(s) \vDash \phi, \phi \in LTL$.后面⊨的定义与 LTL 中相同。

（4） 称 TS 中一条由起点出发，且不能继续延长(即达到终状态或无限长)的状态转移链为路径。TS 中一切路径记为$Paths(TS)$.

（5） 路径$\sigma = s_0 s_1 \ldots$为一条路径，$trace(\sigma) = L(s_0)L(s_1)\ldots$为其迹。于是类似扩展至$Paths(TS)$上，定义出$Traces(TS)$.

（6） 将2^{AP}视为字母表，字母表中科取出若干字母构成单词。按照此定义，迹必定为单词。

3.2 *LTL* 公式的设计

3.2.1 *LTL* 公式的定义

LTL 公式定义在$AP \cup \{true\}$之上，并由∧,∪, 0, ¬四个运算生成。其中前两者为二元运算符，后两者为一元运算符。

在$(2^{AP})^{\omega} \times LTL$上定义二元关系⊨，满足：

$$
\begin{array}{lll}
\sigma & \models & \text{true} \\
\sigma & \models & a \qquad \text{iff} \quad a \in A_0 \quad (\text{i.e., } A_0 \models a) \\
\sigma & \models & \varphi_1 \wedge \varphi_2 \quad \text{iff} \quad \sigma \models \varphi_1 \text{ and } \sigma \models \varphi_2 \\
\sigma & \models & \neg \varphi \qquad \text{iff} \quad \sigma \not\models \varphi \\
\sigma & \models & \bigcirc \varphi \qquad \text{iff} \quad \sigma[1\ldots] = A_1 A_2 A_3 \ldots \models \varphi \\
\sigma & \models & \varphi_1 \mathsf{U} \varphi_2 \quad \text{iff} \quad \exists j \geqslant 0.\ \sigma[j\ldots] \models \varphi_2 \ \text{ and } \ \sigma[i\ldots] \models \varphi_1, \text{ for all } 0 \leqslant i < j
\end{array}
$$

且有限状态迁移系统 TS，TS 满足φ当且仅当 TS 中一切路径的迹满足φ。

引入以下符号

$$Words(\varphi) = \left\{\sigma \in (2^{AP})^{\omega} \mid \sigma \models \varphi\right\}$$

即$Traces(TS) \subset Words(\varphi)$.

此外，还可以规定运算符如下：

$$\Diamond\varphi \stackrel{\text{def}}{=} \text{true}\,\mathsf{U}\,\varphi \qquad \Box\varphi \stackrel{\text{def}}{=} \neg\Diamond\neg\varphi$$

分别表示在未来某一时刻（*eventually*）和总是（*always*）。

以及：

$\Box\Diamond\varphi$ “infinitely often φ”
$\Diamond\Box\varphi$ “eventually forever φ”

供以后使用。注意到以上四个均在 *LTL* 无终状态时有意义。

最后，还可以定义 *weak until* 二元运算符。此运算符在给出 *LTL* 标准形式时用处颇多。定义为：$\varphi\,\mathsf{W}\,\psi \stackrel{\text{def}}{=} (\varphi\,\mathsf{U}\,\psi) \vee \Box\varphi.$

3.2.2 *LTL* 公式的的实现

LTL 采用二叉树结构，且一元运算符的左子树为 *None*，运算对象于有子树之中。二元运算符左右子树分别为参与运算的两个运算对象。

```
class LTL:
    def __init__(self, data, left=None, right=None):
        self.data = data
        self.left = left
        self.right = right
```

输入 *LTL* 时，采用以上算符的定义可以将“*and*”，“*not*”，“*until*”，“*next*”，“*weak until*”，“*eventually*”，“*always*”，“*or*”这 8 种运算符转化为前四种存入 *LTL* 结构中，以实现相关运算。称前四种运算符为基本运算符。

3.3.3 *LTL* 公式中的方法

LTL 中定义了以下几个方法：

（1） *sub* 求子公式，利用树形结构十分容易。

（2） 对某一公式求否，并将$\neg\neg a$ 与a等同。

（3） *closure* 给出 *LTL* 公式f的闭包，其定义为满足如下条件的最小集合：

- **true** $\in cl(f)$
- $f \in cl(f)$
- if $f_1 \in cl(f)$ then $neg(f_1) \in cl(f)$
- if $\mathbf{X}\, f_1 \in cl(f)$ then $f_1 \in cl(f)$
- if $f_1 \wedge f_2 \in cl(f)$ then $f_1, f_2 \in cl(f)$
- if $f_1 \vee f_2 \in cl(f)$ then $f_1, f_2 \in cl(f)$
- if $f_1\ \mathbf{U}\ f_2 \in cl(f)$ then $f_1, f_2 \in cl(f)$
- if $f_1\ \mathbf{R}\ f_2 \in cl(f)$ then $f_1, f_2 \in cl(f)$

这里f不妨设仅仅只有基本的四个运算符。并且以上关系中将$\neg\neg f$与f等价。注意这是唯一的等价关系，并且我们不对运算的括号进行展开。

（4） *elem_sub* 求基本子集，基本子集 *B* 为f闭包的子集，且满足：

1. B is *consistent* with respect to propositional logic, i.e., for all $\varphi_1 \wedge \varphi_2, \psi \in closure(\varphi)$:
   - $\varphi_1 \wedge \varphi_2 \in B \Leftrightarrow \varphi_1 \in B$ and $\varphi_2 \in B$
   - $\psi \in B \Rightarrow \neg\psi \notin B$
   - true $\in closure(\varphi) \Rightarrow$ true $\in B$.
2. B is *locally consistent* with respect to the until operator, i.e., for all $\varphi_1 \mathsf{U} \varphi_2 \in closure(\varphi)$:
   - $\varphi_2 \in B \Rightarrow \varphi_1 \mathsf{U} \varphi_2 \in B$
   - $\varphi_1 \mathsf{U} \varphi_2 \in B$ and $\varphi_2 \notin B \Rightarrow \varphi_1 \in B$.
3. B is *maximal*, i.e., for all $\psi \in closure(\varphi)$:
   - $\psi \notin B \Rightarrow \neg\psi \in B$.

分别使用三个静态方法 *consistent, locally_consistent, maximal* 来实现。

（5） 由于二叉树可由先根和中根序唯一确定，因而可以利用这个性质写出判断两个 *LTL* 公式相等的方法。因为 *Python* 中集合的元素必须为 *hashable*，故也可以定义 *LTL* 的 *hash* 值为其先根序和中根序构成元组的 *hash* 值。由于显然的原因，我们这里不会考虑碰撞的问题。

4.输入数据处理

4.1. *LTL* 公式转换成二叉树的算法

整个数据输入部分仅仅这一步较为困难，故在此详述。

首先，将文件里保存的迁移系统的信息读入程序待处理，得到一个存有所有线性时序逻辑公式的表。

LTL 类的初始化方法如下：

```
def __init__(self, data, left=None, right=None):
    self.data = data
    self.left = left
    self.right = right
```

这样定义出来的 *LTL* 实际上表示二叉树的一个节点，在每个节点里，*self.data* 存储算符，*self.left, self.right* 存储算符的左边和右边的子表达式。对于一元运算符，我们在其左边存入一个 *None*。基本命题，即各个 *action* 也存入 *self.data*，其左右存入 *None*。要得到 *LTL* 公式的二叉树，还需要定义一个函数来处理字符串。对于每个由字符串表示的公式，可以参考将四则运算算式转换为表达式树的算法。我们需要两个栈，记为 *st*，*exl*，*st* 用于存储运算符，*exl* 用于存储已得到的子表达式树。每次不断打开左括号直到遇到表达式，将其运算符存入 *st*，将其左右基本命题转化为表达式树存入 *exl* 中。遇到右括号，则弹出 *st* 中最后一个运算符和 *exl* 中最后两个表达式树，将其合并为一个表达式树，存入 *exl* 中，这样可以保证 *exl* 中的最后两个表达式树始终是此次弹出的运算符所对应的左右子表达树。直到 *exp* 处理完，取出 *exl* 中第一个也是唯一一个（在表达式符合规范的情况下）表达式树，即是所求的表达式树。

具体处理中，记表达式为 *exp*，首先处理整个表达式由一个基本命题构成的情况。然后，为方便处理，将表达式中的 *weakuntil* 全部替换为 *weak*。之后，函数的主要部分由一个循环完成，循环条件为“*while exp*：”。分四种情况处理：i：*exp* 的开头为‘（’；ii：开头为‘）’；iii：一元运算符；iiii：二元运算符。

i: 遇到‘（’，将其去掉，继续循环。这里不需要把‘（’压入栈 *st* 中，因为题目要求仅处理所有算式由括号括起的情况，即所有表达式都是符合规范的，故不需要判断括号是否配对、表达式是否合法。

ii：遇到‘）’，弹出 *st* 中运算符，取出 *exl* 中最后两个表达式树，以运算符为节点数据，两个表达式树为左右子节点，合并为一个表达式树，存入 *exl* 中，继续循环，如图所示：

```
while exp:
    if exp[0] == '(':
        exp = exp[1:].strip()
    elif exp[0] == ')':
        data = st.pop()
        r, l = exl.pop(), exl.pop()
        exl.append(LTL(data, l, r))
        exp = exp[1:].strip()
    elif exp[:3] == 'not':
```

iii：对于一元运算符，以 *not* 为例，首先判断前三位子字符串是否是‘*not*’，是，在 *exl* 中存入 *None*，在 *st* 中存入‘*not*’，去掉 *exp* 的前三位，这样又面临两种情况，运算符后面是基本命题或括号括起来的表达式。如果是后者，则 *exp* 的首位是‘(’，归为情况 i，继续循环即可。如果是前者，需要取出该基本命题。由于所有运算都由括号括起来，只需找到第一个右括号所在位置，在这之前的子字符串即为所需的基本命题，提取出来，存入 *exl*，继续循环。如图所示：

```
elif exp[:3] == 'not':
    exl.append(None)
    st.append('not')
    exp = exp[3:].strip()
    if exp[0] == '(':
        continue
    else:
        c = exp.find(')')
        exl.append(LTL(exp[:c].strip(), None, None))
        exp = exp[c:]
        continue
```

iiii：对于二元运算符‘*and*’，‘*until*’和‘*or*’。由于运算符不一定在开头，不好判断，故将其放在最后一种情况处理，这样，当其他情况都不符合，就只剩下这一情况。以 *a*，*b*，*d*，*w* 分别记‘*and*’，‘*until*’，‘*or*’和‘*weak*’首次出现的位置，以处理 *until* 为例，

```
if b != -1 and (a == -1 or b < a) and (d == -1 or b < d) and (w == -1 or b < w):    # until
    if b == 0:
        st.append('until')
        exp = exp[b+5:].strip()
        if exp[0] == '(':
            continue
        else:
            c = exp.find(')')
            exl.append(LTL(exp[:c].strip(), None, None))
            exp = exp[c:]
            continue
    else:
        exl.append(LTL(exp[: b].strip(), None, None))
        st.append('until')
        exp = exp[b+5:].strip()
        if exp[0] == '(':
            continue
        else:
            c = exp.find(')')
            exl.append(LTL(exp[:c].strip(), None, None))
            exp = exp[c:]
            continue
```

其中 If b != -1 and (a == -1 or b < a) and (d == -1 or b < d) and (w == -1 or b < w): # until

b != -1, 判断‘*until*’在 *exp* 内。*(a == -1 or b < a)*，*(d == -1 or b < d)*，*(w == -1 or b < w)*分别判断‘*until*’在‘*and*’，‘*or*’，‘*weak*’之前。如果 *b* == 0，则其处理方式与一元运算符类似。如果 *b* != 0，则取出 *b* 前一段字符，存入 *exl* 中，再将‘*until*’存入 *st* 中，保留 *until* 之后的字符串，剩下的处理与一元运算符处理后类似。

当 *exp* 截取完，取出 *exl* 中的第一个元素，即是字符串转化得到的表达式二叉树。此时的二叉树还需要进一步处理，将运算符‘*weakuntil*’，‘*or*’，‘*always*’，‘*eventually*’转换成由更基本的运算符。等价关系如下：

ϕ weakuntil ψ = (ϕ until ψ) or（always ϕ）；

ϕ1 or ϕ2 = not ((not ϕ1) and（not ϕ2)）；

always ϕ = not（eventually（not ϕ））；

eventually ϕ = true until ϕ。

Weakuntil 转换得到 *or* 和 *always*，*always* 转换得到 *eventually*，因此 *weakuntil* 的转换应该在 *or* 和 *always* 之前，*always* 的转换应该在 *eventually* 之前。

具体的实现由四个函数 *trans_weakuntil*，*trans_or*，*trans_always*，*trans_eventually* 达到。四个函数的定义类似，按宽度优先序遍历二叉树，遇到待处理的运算符，按相应规则转换即可。

5.算法设计

5.1 构建 *NBA*

A *nondeterministic Büchi automaton* (NBA) $\mathcal{A}$ is a tuple $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ where

- Q is a finite set of states,
- Σ is an alphabet,
- $\delta : Q \times \Sigma \to 2^Q$ is a transition function,
- $Q_0 \subseteq Q$ is a set of initial states, and
- $F \subseteq Q$ is a set of *accept* (or: final) states, called the *acceptance set*.

其中$\sigma = A_0A_1A_2 \ldots \in \Sigma^\omega$表示无限序列 $q_0q_1q_2 \ldots \in Q$，其中$q_0 \in Q_0, q_i \in \delta(q_{i-1}, A_i)$。且若$q_0q_1q_2 \ldots$中有无限多项在 F中，则称之为可接受的。并且定义：

$$\mathcal{L}_\omega(\mathcal{A}) = \{\sigma \in \Sigma^\omega \mid \text{there exists an accepting run for } \sigma \text{ in } \mathcal{A}\}.$$

GNBA 的定义则将 $\mathcal{F}$ 取为2^Q子集族。

5.2 由 *LTL* 生成 *NBA*

5.2.1 由 *LTL* 生成 *GNBA*

设 *LTL* 公式为f，定义在AP上，下试图在AP幂集上定义 *GNBA*

（1） Q为f闭包内一切基本子集构成的子集族。

（2） Q_0为Q中包含f的子集构成的子集族。

（3） 设$f_1 \cup f_2 \in closure(f)$, 定义$F_{f_1 \cup f_2} = \{A \in Q \mid f_1 \cup f_2 \notin A \vee f_2 \in B\}$。

则$F = \{F_{f_1 \cup f_2} \mid f_1 \cup f_2 \in closure(f)$。

（4） 定义$\delta: Q \times 2^{AP} \to 2^Q$为：

若$A \cap closure(f) \neq B \cap AP$，则令$\delta(B, A) = \emptyset$.

若不然，为满足以下两个条件的集合B'构成的集族：

① 对任意$\bigcirc g \in closure(f)$: $\bigcirc g \in B \Leftrightarrow g \in B'$.

② 对任意$g_1 \cup g_2 \in closure(f)$:

$$g_1 \cup g_2 \in B \Leftrightarrow (g_2 \in B \vee (g_1 \in B \wedge g_1 \cup g_2 \in B'))$$

且当F为空集族时，设F包含Q为其唯一元素。

5.2.2 由 *GNBA* 生成 *NBA*

设$G = (Q, \Sigma, \delta, Q_0, F)$，且$F = \{F1, \ldots, Fk\}$，则按照如下方式构造 NBA：

$$Q' = Q \times \{1, \ldots, k\},$$
$$Q'_0 = Q_0 \times \{1\} \models \{\langle q_0, 1\rangle \mid q_0 \in Q_0\},$$
$$F' = F_1 \times \{1\} = \{\langle q_F, 1\rangle \mid q_F \in F_1\}.$$

$$\delta'(\langle q, i\rangle, A) = \begin{cases} \{\langle q', i\rangle \mid q' \in \delta(q, A)\} & \text{if } q \notin F_i \\ \{\langle q', i{+}1\rangle \mid q' \in \delta(q, A)\} & \text{otherwise.} \end{cases}$$

且将$k+1$ 与1 等同。

则此算法的复杂度为$O(|G| \times |F|)$，其中$|G|$为 $GNBA$ 中的状态与迁移数目之和。

5.3 *TS* 与 *NBA* 的乘积

设$TS = (S, Act, \rightarrow, I, AP, L)$，$NBA = (Q, 2^{AP}, \delta, Q_0, F)$。且 TS 无终状态，NBA 满足$\delta(q, A) \neq \emptyset, \forall q \in Q, A \in 2^{AP}$. 定义$TS \otimes NBA = (S \times Q, Act, \rightarrow', I', AP', L')$，且满足：

(1) $\rightarrow'$满足$s \rightarrow_\alpha t \wedge q \rightarrow_{L(t)} p \Leftrightarrow (s, q) \rightarrow'_\alpha (t, p)$.

(2) $AP' = Q$.

(3) L':$S \times Q \rightarrow 2^Q$满足$L'\big((s, q)\big) = \{q\}$.

然后在AP上定义命题逻辑公式$\phi = \bigwedge_{a \in F} \neg a$，使用两次深度搜索在其上完成检验，具体过程将在下面给出。

6.关键问题和具体分析

由定义，有限状态迁移 TS 满足公式$\varphi \Leftrightarrow Traces(TS) \subset Words(\varphi) \Leftrightarrow Traces(TS) \cap Words(\neg\varphi) = \emptyset$.下面构造 $GNBA$，满足:$L_\omega(GNBA) = Words(\neg\varphi)$。

构造过程已经在上面给出，下面说一下其大致思路，为明显计，设对φ进行构造：

设$\sigma = A_0A_1A_2 \ldots \in Words(\varphi)$，试图构造$B_i$满足：

$$\psi \in B_i \Leftrightarrow A_iA_{i+1}A_{i+2} \ldots \vDash \psi$$

ψ为φ的子式，由于完备性不妨设ψ在φ的闭包中。

GNBA 将上述B_i作为其状态，并且构造要使得$\sigma' = B_0B_1B_2 \ldots$为 *GNBA* 中的一条状态迁移链。于是我们需要将逻辑运算符表示为 *GNBA* 里的状态、迁移关系及终状态。其中“*true*”论断以及命题逻辑关系（包括运算符“*not*”，“*and*”）被 *GNBA* 的状态记录；“*next*”运算符被 *GNBA* 的迁移关系记录；“*until*”运算符根据展开律$f_1 \cup f_2 = f_2 \vee (f_1 \wedge \bigcirc(f_1 \cup f_2))$可以将其用状态和迁移关系分别记录。

其中上面定义的基本子集三条性质中 *consistent* 用来处理命题逻辑关系，使之不至于出现矛盾，意义较为明显。*locally consistent* 是用来描述“*until*”在展开律下的一致性，第一条意味着$A_iA_{i+1}A_{i+2} \ldots \vDash \varphi_2$,自然有$A_iA_{i+1}A_{i+2} \ldots \vDash \varphi_1 \cup \varphi_2$;第二条意味着$f_1 \wedge \bigcirc(f_1 \cup f_2) \in B_i$，自然有$f_1 \in B_i$。*maximal* 定义是由于一条路径要么满足一个公式，要么不满足，于是一个命题或其否必在B_i中。 Q即为把一切上述定义出的基本子集为状态。由上述B_i的定义，φ必定在B_0中，于是Q_0取为一切包含φ的基本子集。由B定义当$A \cap closure(f) = B \cap AP$时$B$存在下一个状态，于是我们设其他情况没有下一个状态，否则会在 *GNBA* 中添加一些没有必要的迁移关系。δ第一条是由于希望$\bigcirc f \in B_i \Leftrightarrow f \in B_{i+1}$，第二条是由于“*until*”算符的展开律与$B_i$定义配合。

设由$A_0A_1A_2 \ldots \in Words(\varphi)$定义出来的$B_0B_1B_2 \ldots$显然均为基本子集，且$B_0 \in Q_0$。$B_0B_1B_2 \ldots$为 *GNBA* 中一系列状态，根据$\delta$的定义，可以验证为 *GNBA* 中的一条状态迁移链。设已经添加 $\mathcal{F}$ 如前，若$B_i \notin F_{\varphi_{1,j} \cup \varphi_{2,j}}$，则$\exists k > i, s.t. \varphi_{2,j} \in B_k$，于是$B_k \in F_{\varphi_{1,j} \cup \varphi_{2,j}}$，从而对无限多$i$有$B_i \in F_{\varphi_{1,j} \cup \varphi_{2,j}}$，从而$B_0B_1B_2 \ldots$的可接受的。于是以上证明了$L_\omega(GNBA) \supset Words(\varphi)$。

反之，注意到展开律并不能完全描述“*until*”算符。“*until*”还具有某种最小性。援引以下引理，其中最小性自然定义，引理证明略去。

For LTL formulae φ and ψ, $Words(\varphi \,\mathsf{U}\, \psi)$ is the least LT property $P \subseteq (2^{AP})^\omega$ such that:

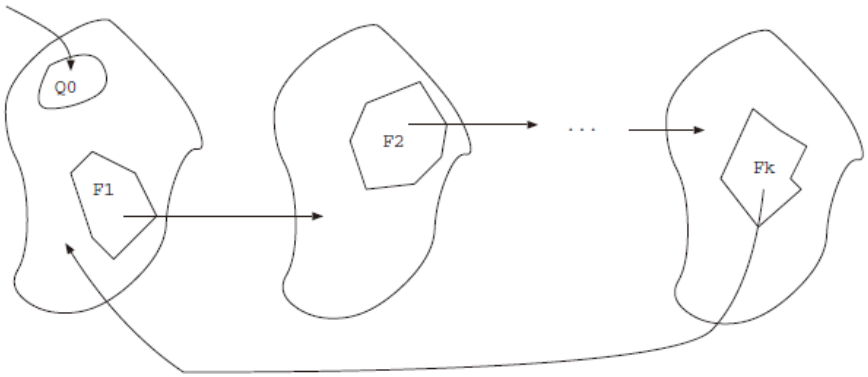
$$Words(\psi) \cup \{A_0 A_1 A_2 \ldots \in Words(\varphi) \mid A_1 A_2 \ldots \in P\} \subseteq P \qquad (*)$$

Moreover, $Words(\varphi \,\mathsf{U}\, \psi)$ agrees with the set

$$Words(\psi) \cup \{A_0 A_1 A_2 \ldots \in Words(\varphi) \mid A_1 A_2 \ldots \in Words(\varphi \,\mathsf{U}\, \psi)\}.$$

使用这一个引理，可以证明$L_\omega(GNBA) \subset Words(\varphi)$，由于较为复杂且启发性不强，故在此处略去。

由 $GNBA$ 构造 NBA 的过程实质上是把集族 $\mathcal{F}$ 换为其第一个分量F。注意到若 $\mathcal{F}$ 为空，则可以在 $\mathcal{F}$ 中加入元素$Q \in GNBA$，不难验证此时并不影响 $GNBA$ 的结构。设 $\mathcal{F} = \{F_1, F_2, \dots, F_k\}$，取$k$份$Q$，并将第一份中的$Q_0$视为初始子集。定义迁移函数将第$i$份$Q$里的$F_i$内元素迁移至第$i+1$份之中，余下元素仍然在第$i$份里，具体定义如上。这样定义的迁移函数可以保证 NBA 每一个可接受迁移必定经过了k份Q里的所有F_i。因为一旦进入F_1则下一步进入第二份Q，然后要想回到F_1必须顺次经过所有F_i，具体如下图所示。从而由 $GNBA$ 构造出的 NBA 的可接受路径，可以与原 $GNBA$ 可接受路径建立典范同构。特别的，由迁移函数定义，$L_\omega(GNBA) = L_\omega(NBA)$。（注意路径为同构而迁移单词为相同）



设已经根据¬φ构造出对应的$NBA_{\neg\varphi}$现在我们希望计算$Traces(TS) \cap Words(\neg\varphi) = Traces(TS) \cap L_\omega(NBA)$。定义留存性质（$persistence\ property$）为形如“$eventually\ always\ \phi$”的线性时序性质，其中$\phi$为命题逻辑公式，换言之，$\phi$从某一状态起便一直成立。下面证明:

$$Traces(TS) \cap L_\omega(NBA) = \emptyset \Leftrightarrow TS \otimes NBA \vDash "eventually\ always\ \phi"（*）$$

“⇐”：反之，取一条不满足论断的路径$\sigma' = (s_0, q_1)(s_1, q_2)$ ….，投射至TS上得到其上路径σ，且存在无穷多i使得q_i不满足ϕ，即$q_i \in F$。又由于$q_i \to_{L(s_i)} q_{i+1}$，从而$q_0 q_1$ …为NBA中可接受状态转移序列。因而$trace(\sigma) \in Traces(TS) \cap L_\omega(NBA)$。

“⇒”:反之，TS中取状态链$\sigma = s_0 s_1 s_2$ …满足其迹在$L_\omega(NBA)$中，则可设$q_0 q_1 q_2$ …满足$q_0 \in Q_0$，$q_i \to_{L(s_i)} q_{i+1}$，由于对无穷多的i，有$q_i \in F$成立。从而可以在$TS \otimes A$中找到$\sigma' = (s_0, q_1)(s_1, q_2) \dots$，于是$TS \otimes A \vDash "not\ eventually\ always\ \phi"$

于是我们仅仅需要在$TS \otimes NBA$检验"$eventually\ always\ \phi$"，由ϕ的定义，这等价于在有向图$Forget(TS \otimes NBA)$上寻找是否存在一条路径，与点集$Forget(F)$相交无穷

多次。（回忆之前关于遗忘函子的定义）这样的路径仅仅可能有一种形式，即从初状态出发，到达F中的一个状态，并且此状态在一个环上。采取两部深度优先搜索来解决这个问题。

具体而言，*persistence_checking* 函数内部定义两个函数 *cycle_check* 和 *reachable_cycle*，以及四个变量 R，U，T，V，及一个布尔量 *cycle_found*。其中 T 和 V 是用于 *cycle_check* 的变量，每次运行 *cycle_check* 均初始化为空。V 为记录路径的栈，T 为记录搜索中已到达的点的集合。*reachable_cycle* 则是用来寻找从初始状态到F中状态的路径，旦找到了一个在F中状态，调用 *cycle_check* 检验是否有圈。其中两个子程序都涉及布尔量 *cycle_found* 的修改，一旦由 *cycle_check* 返回的 *cycle_found* 为 *True*，则 *reachable_cycle* 便跳出循环，同时 *persistence_checking* 也跳出循环，表明已经找到反例。

主程序 *persistence_checking* 中，每次在未被访问的初始状态中找一个状态，对其作 *reachable_cycle*，如果能够找到一条与$Forget(F)$相交无穷多次的路径，则说明 $TS \otimes A$不满足"*eventually always* ϕ"，根据以上等价关系可知道原来的TS不满足φ，由（*）式的证明可见，$trace(\sigma) \in L_\omega(NBA)$，于是$TS \otimes NBA$一条不满足的路径投射到$TS$上也不满足$\varphi$，于是可以生成反例。即把 U 与 V 接起来。反之，说明TS满足φ。

于是以上的算法可以实现 *LTL* 的模型检验及反例生成。

$$
\begin{aligned}
TS \models \varphi \quad &\text{iff} \quad Traces(TS) \subseteq Words(\varphi) \\
&\text{iff} \quad Traces(TS) \subseteq (2^{AP})^\omega \setminus Words(\neg\varphi) \\
&\text{iff} \quad Traces(TS) \cap \underbrace{Words(\neg\varphi)}_{\mathcal{L}_\omega(\mathcal{A}_{\neg\varphi})} = \varnothing \\
&\text{iff} \quad TS \otimes \mathcal{A}_{\neg\varphi} \models \Diamond\Box\neg F.
\end{aligned}
$$

（附：等价关系总结）

7.复杂度分析

设状态个数为 n，则 *AP* 的原子公式个数也为 n。公式φ中原子论断为 m。*Act* 的大小为 l。

对于一个公式φ，验证它一般需要$O(m)$的时间复杂度。

7.1 *NBA* 的构造

构造基本子集的时候需要对φ的闭包取子集，由φ的二叉树结构，闭包元素为不会超过$4m$，从而后面构造基本子集过程中，由于要取闭包的一切子集，从而复杂度为$O(2^m)$，$GNBA$ 的Σ为AP的幂集，从而复杂度也为$O(2^m)$。后面构造 NBA 的过程中，构造了Q的k个副本，由于k为 $\mathcal{F}$ 元素，于是k为$O(m)$量级；此外定义 NBA 上迁移函数调用了 $GNBA$ 中定义的迁移函数，这一步复杂度至多也为$O(m \times 2^m)$。综上可以看出构造 NBA 的时间和空间复杂度为$O(m \times 2^m)$量级。

7.2 $TS \otimes NBA$的构造

构造$S \times Q$大致需要的时间复杂度为$O(n \times 2^m)$，而构造其上迁移关系需要考虑 NBA 上的迁移与原来 TS 上迁移，由于 NBA 上迁移仅仅调用了定义在$Q \times L(t)$的δ（t 定义与算法处一致），因而复杂度为$O(l \times 2^m)$。L'仅仅形式定义以后不会用到命题逻辑的定义为$O(|F|) \leq O(2^m)$，因而这一步复杂度为$O(|TS| \times 2^m)$

7.3 *Persistence property* 的检验

显然这一个步骤的空间复杂度为$O(n' + l')$, n', l'为新构造出 TS 的对应数量。不妨设所有状态都由初始状态可达到，那么由于 T 在操作过程中不断增加，于是在内部深度搜索中（即 $cycle\_check$），TS 中每一个状态至多被经过一次，而调用 $cycle\_check$ 一般而言需要m'次,检验一个状态是否在 F 中也需要$O(m')$的时间复杂度。在外部深度搜索中，同样由于 R 不断增加，每一个状态至多也只是出现一次。同时，在计算一个状态的下一个状态时，会计算其发出的边；易见这一些边也只会被计算一次。综上可知，最坏情况的时间复杂度为$O(n' + l' + n'm')$

7.4 整个过程的复杂度分析

由于构造 NBA 复杂度为$O(|TS| \times 2^m)$，$persistence\ property$ 的检验的复杂度不会超过这一个。因为$l' = |Q| < 2^m$, $n' = |S \times Q| < n \times 2^m$，$m' = |F| < m$，故总复杂度为$O(|TS| \times 2^m)$。

PART Ⅱ
计算树逻辑（CTL）的检验

1. 问题描述和需求分析

1.1 问题要求简述：

设计并实现一个 *CTL* 的模型检查工具，使得对给定的状态迁移系统 *TS* 和 *CTL* 公式 *L*，可以判断 *TS* 是否满足 *L*，若满足，返回 *True*，若不满足，返回 *False* 并给出 *TS* 中一条不满足 *L* 的路径作为反例。并对所给出的实现进行性能分析。

1.2 问题背景与分析：

计算树逻辑——*CTL*（*Computation Tree Logical*），是检验系统属性的一个重要时态逻辑结构分支。计算树逻辑的提出是由于，在 *LTL*（*Linear Temporal Logical*）中，一些过于复杂的模型检验问题难以得到解决，例如检验从状态 *s* 出发是否所有的道路π都满足公式φ。因此为了克服这样的困难，上世纪八十年代，*Clarke and Emerson* 提出了 *CTL*。不同于 *LTL* 的是，CTL 并非是基于线性的时间概念——一连串线性的状态，而是基于树形的时间概念——树形的状态。树形的状态决定了在每一个时刻（状态），检验都分裂进入了不同的可能分支之中，这也解释了为何 *CTL* 能解决 *LTL* 所不能解决的复杂问题。

CTL 主要考虑两类检验问题，一种是道路的存在性问题，即从初始状态是否存在一条道路满足公式φ；另一种是道路的普适性问题，即从初始状态出发的任何一条道路是否都能满足公式φ。

1.3 设计思路：

首先，对于输入的 *CTL* 公式φ，我们将其转化为基本的一些命题，并将其存储在一棵二叉树中。其中，每个结点仅存储一个符号或者一个最基本的命题（不可再分的），使得按照中根序遍历这棵二叉树，可以得到最初输入的公式φ。其次，我们需要将存储在二叉树的 *CTL* 公式φ转化为 *ENF* 形式（*Existential Normal Form*），这一转化过程在二叉树中以中根序遍历的方式进行。最后，我们引入 *Sat*（*Satisfaction Set*），通过递归的方式计算出公式φ的所有子结构的 *Sat* 并最终得到φ的子结构。这样一来，我们只需要验证 *TS*（*Transitional System*）的初始状态 *I* 是否包含于 *Sat*（φ），就能够判断该系统是否满足这一 *CTL* 公式。

2. 程序设计（数据结构与算法）

2.1 输入算法设计：

对于文件读取函数 *read_TS*()，在之前的报告中已经进行了介绍，这里不予赘述。

2.2 存储算法设计：

对于迁移系统 *TS*，它的设计和思想在 *LTL* 部分中已经完整给出，值得注意的是，这里添加了函数 *pre*()和函数 *post*()，用于这里主要介绍对于 *CTL* 公式φ的存储。在类 *CTL* 中，我们定义了四个主要的静态方法函数，分别为 *tokens*，*infix_order*，*trans_infix_suffix*，*construct_tree*，下面作细致介绍。

在函数 *tokens*，*infix_order* 和 *trans_infix_suffix* 中，参数均为(*Propositions*, *Operators*, *line*)，其中 *Propositions* 表示所有的迁移状态（即 *TS* 中的所有状态 *States*），*Operators* 表示所有的运算符号（包括一元运算符、二元运算符以及括号，集合形式），*line* 则表示一个表达式（即 *CTL* 公式φ，字符串形式）。

首先我们需要将表达式 *line* 里面的所有最基本元素提取出来，这些最基本的元素既包括 *Propositions* 中的元素，也包括 *Operators* 中的元素。为此我们实现了一个生成器函数 *tokens*，它按公式φ从左到右的输入顺序将其中属于 *Propositions* 和 *Operators* 的元素一一生成（理论上在公式合法的情况下，*tokens* 作扫描的过程中只是丢弃了字符串 *line* 的一些空格）。生成器函数 *tokens* 将表达式 *line* 分块，方便了接下来的操作。

其次我们自然而然地可以确定我们需要构造的二叉树的中根序（因为之前我们已经提到，以中根序遍历我们的目标二叉树即是对 *line* 从左到右遍历所有基本元素），只需要将 *tokens* 生成出的元素（排除掉括号）直接按顺序排成一个表即可，这一点由函数 *infix_order* 实现。接下来我们只需要确定目标二叉树的后根序，这样一来通过中根序和后跟序我们就可以确定这棵二叉树。

从中根序到后根序的转化由函数 *trans_infix_suffix* 实现。这一过程基于所有运算均由括号确定优先顺序，因此不需要比较各个运算符的优先级，同时保证一组括号内只会出现一个运算符以及至多两个运算元。首先引入了一个栈 *st* 和一个空表 *exp*（这个空表用于存储最终输出的后根序），注意到对于一个括号内的运算符以及运算元，无论该运算符是一元运算符还是二元运算符，它在这个括号内的所有运算所构建的子树中都处于根节点的位置，这意味着它在后根序中应该处于这个括号内最后的位置，而对于其它的运算元，它们的相对次序则对应于后根序中的相对次序。因此对于

tokens 按顺序生成的元素，如果它属于 *Propositions*，那么就直接将其加入表 *exp*，如果它属于 *Operators* 且不是反括号，那么就将其压入栈 *st*，如果它是反括号‘)’，那么我们需要将栈 *st* 中的元素一一出栈，直到遇到与之对应的正括号‘(’就停止（即遇到的第一个正括号），这其中出栈的元素（根据前面的分析有且仅有一个），则加入表 *exp*。当所有 *tokens* 中的元素全部生成完毕后，再将栈 *st* 中剩余的元素加入表 *exp* 中（如果有的话也仅有一个）。最终我们返回表 *exp* 表示由中根序转换得到的后根序。

最后，我们通过函数 *construct_tree* 来由中根序和后根序生成目标二叉树。这一算法由递归的方法实现，每次我们由后根序总能确定一棵树的根节点（后根序的最后一位），然后我们在中根序中寻找根节点的位置（从前往后一一比较并最终确定下标为 *i*）。从而我们能将这棵树按左右子树分开，并赋予当前根节点值。（左子树的中根序为原中根序的前 *i* 个，后根序为原后根序的前 *i* 个；右子树的中根序为原中根序的第 *i+2* 到最后一个，后根序为原后根序的第 *i+1* 到倒数第二个）再对左右子树分别递归调用 *construct_tree* 即可。

综上所述，我们就将 *CTL* 公式φ存储到了一棵二叉树中。对于类 *CTL*，我们只需要给出待检验的迁移系统 *TS* 和公式φ（*expression*），我们将 *TS* 的状态集合 *S* 作为 *Propositions*，并在外部取出所有的运算符号 *operators* 作为 *Operators*，就能生成出我们需要的目标二叉树。特别的，对于所生成的二叉树的任一结点，如果它存储的是一元运算符，那么它的左子树为空。

2.3 形式转换算法设计

对于已经存入二叉树中的 *CTL* 公式φ，我们在二叉树内对其进行形式转化，使之成为 *ENF* 形式。

首先我们给出 *ENF* 的定义：

Definition 6.13. Existential Normal Form (for CTL)

For $a \in AP$, the set of CTL state formulae in *existential normal form* (ENF, for short) is given by

$$\Phi \quad ::= \quad \text{true} \;\Big|\; a \;\Big|\; \Phi_1 \wedge \Phi_2 \;\Big|\; \neg\Phi \;\Big|\; \exists\bigcirc\Phi \;\Big|\; \exists(\Phi_1 \,\mathsf{U}\, \Phi_2) \;\Big|\; \exists\Box\Phi.$$

同时我们也给出一般的 *CTL* 公式的句法形式：

Definition 6.1. Syntax of CTL

CTL *state formulae* over the set AP of atomic proposition are formed according to the following grammar:

$$\Phi ::= \text{true} \mid a \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \exists\varphi \mid \forall\varphi$$

where $a \in AP$ and φ is a path formula. CTL *path formulae* are formed according to the following grammar:

$$\varphi ::= \bigcirc \Phi \mid \Phi_1 \,\mathsf{U}\, \Phi_2$$

where Φ, Φ_1 and Φ_2 are state formulae. ■

我们可以发现对于一般的形式，它只使用∃*O*，∃*U*，∀*O*，∀*U*这四种形式来构建一个基本的 *CTL* 公式（也即是定义中的 *CTL state formulae*），我们只需要将其中的∀O，∀U进行转化，并允许∃的存在。

但我们需要考虑到对于一个道路公式（即定义中的 *CTL path formulae*）它的内部可能存在各种各样的符号（例如∨，♦，W 等），它们会给我们下一步计算 *Sat*（）时带来麻烦。为了避免这种情形的发生，我们需要对这些特殊的符号同时进行转换，将其转化为我们熟悉的符号。

因此我们在 *suf_order_tran* 函数中采用后根序遍历公式存入的二叉树，并在遍历过程中对所到的根节点进行操作 *proc*，具体的操作只需要遵照逻辑运算的原理即可。下面给出所有的转化公式：

$$\forall\bigcirc\Phi \equiv \neg\exists\bigcirc\neg\Phi,$$
$$\forall(\Phi \,\mathsf{U}\, \Psi) \equiv \neg\exists(\neg\Psi \,\mathsf{U}\, (\neg\Phi \wedge \neg\Psi)) \wedge \neg\exists\Box\neg\Psi$$

$$\forall\Diamond\Phi \equiv \neg\exists\Box\neg\Phi,$$
$$\forall\Box\Phi \equiv \neg\exists\Diamond\neg\Phi = \neg\exists(\text{true}\,\mathsf{U}\,\Phi)$$

$$\Diamond\psi = \text{true}\,\mathsf{U}\,\psi$$

$$\varphi\,\mathsf{W}\,\psi \equiv \neg((\varphi \wedge \neg\psi)\,\mathsf{U}\,(\neg\varphi \wedge \neg\psi))$$

$$\varphi_1 \vee \varphi_2 \overset{\text{def}}{=} \neg(\neg\varphi_1 \wedge \neg\varphi_2)$$

这样一来我们就能成功地实现转化过程，从而将存储在二叉树中的 *CTL* 公式转化为 *ENF*。

2.4 *CTL* 模型检验算法设计

CTL 的模型检验十分直接，我们只需要计算 *Sat*（*φ*）即可，然后验证 *TS* 的初始状态集合 *I* 是否包含于 *Sat*（*φ*）即可。下面先给出 *Sat* 的定义：

Given a transition system TS as before, the *satisfaction set* $Sat_{TS}(\Phi)$, or briefly $Sat(\Phi)$, for CTL-state formula Φ is defined by:

$$Sat(\Phi) \;=\; \{\, s \in S \mid s \models \Phi \,\}.$$

结合迁移系统 *TS* 满足一个 *CTL* 公式*φ*的定义，我们得出以下结论：

The transition system TS satisfies CTL formula Φ if and only if Φ holds in all initial states of TS:

$$TS \models \Phi \quad \text{if and only if} \quad \forall s_0 \in I.\, s_0 \models \Phi \quad .$$

This is equivalent to $I \subseteq Sat(\Phi)$. ■

这解释了为何我们只需要计算 *Sat*（*φ*）并验证 *i* 与它的包含关系。

接下来我们采用递归的方式计算 *Sat*（*φ*），即对于整棵二叉树，求它的 *Sat* 可以转化为求它的左右子树的 *Sat*，并根据根节点处的符号决定具体的转化运算，这相当于我们在递归的过程中计算了整棵二叉树的所有子树的 *Sat*。

基于前面的工作，我们已经将*φ*转化为了 *ENF*，所以我们只需要考虑几种递归调用时根节点的情形，并采用相应的递归，这大大简化了我们的运算部分。我们所需讨论的情形如下所示：

switch(Φ):

true	:	**return** S;
a	:	**return** $\{\, s \in S \mid a \in L(s) \,\}$;
$\Phi_1 \wedge \Phi_2$	:	**return** $Sat(\Phi_1) \cap Sat(\Phi_2)$;
$\neg\Psi$	:	**return** $S \setminus Sat(\Psi)$;
$\exists\bigcirc\Psi$	:	**return** $\{\, s \in S \mid Post(s) \cap Sat(\Psi) \neq \varnothing \,\}$;
$\exists(\Phi_1 \,\mathsf{U}\, \Phi_2)$	:	$T := Sat(\Phi_2)$; (* compute the smallest fixed point *) **while** $\{\, s \in Sat(\Phi_1) \setminus T \mid Post(s) \cap T \neq \varnothing \,\} \neq \varnothing$ **do** **let** $s \in \{\, s \in Sat(\Phi_1) \setminus T \mid Post(s) \cap T \neq \varnothing \,\}$; $T := T \cup \{\, s \,\}$; **od**; **return** T;
$\exists\Box\Phi$	:	$T := Sat(\Phi)$; (* compute the greatest fixed point *) **while** $\{\, s \in T \mid Post(s) \cap T = \varnothing \,\} \neq \varnothing$ **do** **let** $s \in \{\, s \in T \mid Post(s) \cap T = \varnothing \,\}$; $T := T \setminus \{\, s \,\}$; **od**; **return** T;

end switch

通过运算我们可以得出 *Sat*（φ），具体的算法和程序设计在后面的部分细致介绍。最后通过 *FinalAnswer* 函数来判断 *I* 是否包含于 *Sat*（φ），如果成立，则返回 *True*，代表这个迁移系统 *TS* 满足 *CTL* 公式φ；否则就返回 *False*，代表这个迁移系统 *TS* 不满足 *CTL* 公式φ。这就完成了 *CTL* 模型检验工作。

3. 关键问题和具体分析

3.1 存储结构类型的选择

我们对于输入的 *CTL* 公式φ，选择了二叉树来存储它，每个结点仅存储一个运算符或者运算元，并且使得最终遍历中根序能够得到最初φ的顺序。这样的存储方式使得后续的操作更加容易完成。在形式转换算法中，我们只需要遍历所有结点并依次进行转换操作即可。而在计算 *Sat*()的部分，我们只需要能够借助树形结构进行递归计算，大大提高了程序的效率。

相较于其它的存储结构，如线性结构的栈或者队列，不能很好地将φ的形式作出分解储存，故不能对后续操作进行简化（原本输入的φ基本上就是线性的）。而对于网状结构的图，不能方便地进行遍历操作，递归的性质也远不如树形结构，因此我们选择树形结构。又由于公式φ中只存在一元运算和二元运算，因此我们只需要使用二叉树便能完成目标。至于每个结点只存储一个运算符或者运算元，则是为了方便转换和递归，减少讨论的情形，所增加的部分存储空间是可以接受的。

3.2 形式转换算法设计中的细节

在形式转换算法中，我们采用后根序对存储 *CTL* 公式φ的二叉树进行遍历转换，这是由于如果采用中根序或者先根序，那么很可能会出现漏过修改部分的情形，例如$\forall(\phi \vee \psi)$,对于中根序或者先根序，它将先到达$\forall$，但是它的右子树的根节点存储的是$\vee$，而我们的程序为了简洁起见，并没有定义这样的情形，因此程序不会进行转换操作，这样一来整个转换算法就失败了。

但是采用后根序就不会出现这样的纰漏，因为后根序总是最后遍历到根节点的位置，因此在达到根节点位置时之前其子结构中的运算符已经完成了转换，就不会出现转换失败的情形。

3.3 *CTL* 模型检验算法的具体分析

我们具体分析 *Sat*()递归运算中的情形。所有的可能情形如下所示：

(a) $Sat(true) = S$,

(b) $Sat(a) = \{ s \in S \mid a \in L(s) \}$, *for any* $a \in AP$,

(c) $Sat(\Phi \wedge \Psi) = Sat(\Phi) \cap Sat(\Psi)$,

(d) $Sat(\neg\Phi) = S \setminus Sat(\Phi)$,

(e) $Sat(\exists \bigcirc \Phi) = \{ s \in S \mid Post(s) \cap Sat(\Phi) \neq \varnothing \}$,

(f) $Sat(\exists(\Phi \,\mathsf{U}\, \Psi))$ *is the smallest subset* T *of* S, *such that*

(1) $Sat(\Psi) \subseteq T$ *and* (2) $s \in Sat(\Phi)$ *and* $Post(s) \cap T \neq \varnothing$ *implies* $s \in T$,

(g) $Sat(\exists \Box \Phi)$ *is the largest subset* T *of* S, *such that*

(3) $T \subseteq Sat(\Phi)$ *and* (4) $s \in T$ *implies* $Post(s) \cap T \neq \varnothing$.

对于*(a)*、*(b)*、*(c)*、*(d)*、*(e)*，它们的运算情形都是显而易见的，可以直接通过定义得到。其中(a)可直接得到(即为全体状态)，(b)可由 answer 实现，(c)可由交集函数实现，(d)可由补集函数实现，(e)可由 answer1 实现。

对于*(f)*，我们采用枚举反向搜索（*Enumerative Backward Search*）的方法计算。以下给出算法思想：（其中括号内对于 ***E*** 和 ***T*** 的评价由于教材错误写反了）

Algorithm 15 Enumerative backward search for computing $Sat(\exists(\Phi \,\mathsf{U}\, \Psi))$

Input: finite transition system TS with state set S and CTL formula $\exists(\Phi \,\mathsf{U}\, \Psi)$
Output: $Sat(\exists(\Phi \,\mathsf{U}\, \Psi)) = \{ s \in S \mid s \models \exists(\Phi \,\mathsf{U}\, \Psi) \}$

$E := Sat(\Psi)$; (* E administers the states s with $s \models \exists(\Phi \,\mathsf{U}\, \Psi)$ *)
$T := E$; (* T contains the already visited states s with $s \models \exists(\Phi \,\mathsf{U}\, \Psi)$ *)
while $E \neq \varnothing$ **do**
  let $s' \in E$;
  $E := E \setminus \{ s' \}$;
  for all $s \in Pre(s')$ **do**
    if $s \in Sat(\Phi) \setminus T$ **then** $E := E \cup \{ s \}; T := T \cup \{ s \}$ **fi**
  od
od
return T

在程序中由函数 *answer2()*给出具体的程序实现。

对于*(g)*，我们采用相同的方法计算。以下给出算法思想：

Algorithm 16 Enumerative backward search to compute $Sat(\exists\Box\,\Phi)$

Input: finite transition system TS with state set S and CTL formula $\exists\Box\,\Phi$
Output: $Sat(\exists\Box\,\Phi) = \{\, s \in S \mid s \models \exists\Box\,\Phi \,\}$

$E := S \setminus Sat(\Phi)$; (* E contains any not visited s' with $s' \not\models \exists\Box\Phi$ *)

$T := Sat(\Phi)$; (* T contains any s for which $s \models \exists\Box\Phi$ has not yet been disproven *)

for all $s \in Sat(\Phi)$ **do** $count[s] := |\, Post(s) \,|$; **od** (* initialize array $count$ *)

while $E \neq \varnothing$ **do**
(* loop invariant: $count[s] = |\, Post(s) \cap (T \cup E) \,|$ *)
  let $s' \in E$; (* $s' \not\models \Phi$ *)
  $E := E \setminus \{\, s' \,\}$; (* s' has been considered *)
  for all $s \in Pre(s')$ **do**
(* update counters $count[s]$ for all predecessors s of s' *)
    if $s \in T$ **then**
      $count[s] := count[s] - 1$;
      if $count[s] = 0$ **then**
        $T := T \setminus \{\, s \,\}$; (* s does not have any successor in T *)
        $E := E \cup \{\, s \,\}$;
      fi
    fi
  od
od
return T

在程序中由函数 *answer3*()给出具体实现。

综上，所有的 *Sat*()计算就全部完成了，此外需要注意的是，*Sat*()函数中全部的输出均采用集合的形式，这样使得形式得到了统一。

4. 复杂度分析（略去读入部分）

设状态个数为 n，公式φ中所有运算符和运算元的个数总为为 m.

4.1 存储部分的复杂度分析

在存储 *CTL* 的二叉树的构建过程中，调用生成器函数 *tokens* 时，对 *line* 中每一个下标 i，均需要遍历其后的所有下标，因此时间复杂度为$O(m^2)$。中根序函数 *infix_order* 单纯调用 *tokens*，因此时间复杂度与之相同，为$O(m^2)$。中根序转为后根序函数 *trans_infix_suffix* 引入了栈和一个表进行临时存储，其空间复杂度不超过 $O(2m)$；由于表达式中所有的运算顺序已经通过括号确定，因此程序中的循环部分的实际时间复杂度是$O(1)$量级的，又由于其它的栈和表的操作的时间复杂都是$O(1)$，因此总的时间复杂度也是$O(m^2)$。根据根序构建二叉树的函数运用递归法，没调用一次需要进行一次循环寻找根节点的位置，这一时间复杂度不超过$O(m)$，而总共的调用次数不超过 m 次，因此调用时循环的复杂度不超过$O(m^2)$。

综上，在构建整棵二叉树时，其时间复杂度为$O(m^2)$，最终我们用一棵二叉树来保存公式φ，它的结点总个数为 m，因此空间复杂度不超过$O(3m)$。

4.2 形式转换部分的复杂度分析

形式转化函数 suf_order_tran 用后根序遍历了目标二叉树，遍历时间复杂度为$O(m)$，在遍历过程中的操作函数 $proc$ 可能会将目标二叉树的结点数量变多，但总体上我们认为它不影响遍历时间的复杂度的量级。因此形式转化部分的时间复杂度为$O(m)$量级。

4.3 *CTL* 模型检验部分的复杂度分析

CTL 模型检验部分也是由递归函数实现，由于二叉树共有 m 个子树（包括自身）因此递归次数不多于 m 次。对于每一次递归，若调用了交集函数 $intersection$ 或者补集函数 $complementary\_set$，则其时间复杂度与空间复杂度均不会超过$O(n)$（由于$Sat()$至多含有 n 个元素）；若调用了函数 $answer$ 或者 $answer1$，容易看出时间复杂度和空间复杂度也是$O(n)$；若调用了 $answer2$ 函数，其中存在一个二重循环，而在验证 s 是否在 $TS.pre(s1)$中时，求 $TS.pre(s1)$也需要进行一次循环，因而时间复杂度不会超过$O(n^3)$；若调用 $answer3$ 函数，其主要部分的时间复杂度分析与 $answer2$ 相同，不会超过$O(n^3)$，同时 $answer2$ 和 $answer3$ 函数的空间复杂度均为$O(n)$。

综上所述，Sat 函数的调用过程中，其总的时间复杂度不会超过$O(mn^3)$，而空间复杂度为$O(mn)$。

4.4 总体复杂度分析

总体而言，程序中时间和空间开销最大的地方在主函数 Sat 的计算中出现，因此总的时间复杂度为$O(mn^3)$，空间复杂度为$O(mn)$。