2015 届研究生硕士学位论文　　学校代码：10269
学　　号：51121500033

華東師範大學

East China Normal University

硕士学位论文

MASTER'S　DISSERTATION

论文题目：

线性时态逻辑可满足性工具的设计与实现

院　　系：软件学院
专　　业：软件工程
研 究 方 向：程序分析与验证
指 导 教 师：何积丰　教授
学位申请人：姚银波

Dissertation for Master Degree of 2015 University Code：10269

Student Code: 51121500033

East China Normal University

Title: The Design and Implementation of Linear Temporal Logic Satisfiability Tool

Department： SEI

Major： Software Engineering

Research direction: Program Analysis and Verification

Supervisor： Prof. Jifeng He

Applicant： Yinbo Yao

摘 要

线性时态逻辑(LTL)自从上个世纪七十年代被引入计算机科学领域之后,得到了很大的发展。现如今,LTL 作为一种形式化的性质规范描述,已在程序验证与分析、程序综合、数据库和人工智能领域被广泛使用。

对于逻辑语言 LTL 来讲，可满足性问题是一个亟待解决的关键性问题。与此同时,对于工业界而言,该问题也是十分重要的。因为对于程序验证、综合或是人工智能领域而言,不可满足的公式是没有意的。

本文就 LTL 的可满足性问题展开,提出了一种新的算法 OFOA(第三章)。传统方法中,对 LTL 的可满足性性判定需要先将其转化为等价的 Buchi 自动机,然后再进行判断;而我们的算法提供了一套全新的推理框架,可以在公式展开的同时对其可满足性进行判断,这就是一个明显的优化。按照“on the fly”的思想,我们为 LTL 公式定义了“义务集合”的概念, 从而极大加速了公式可满足性的判定过程。

以该算法为基础,我们设计实现了 LTL 可满足性判定工具 Aalta, 文章的后半部分主要介绍该工具的设计框架、相应的数据结构和核心算法。为了验证算法的效率,我们选取目前公认的最出色的可满足性检查工具(PANDA、Cadence SMV 和 SPOT)进行比对,并设计了一 系列的实验。我们选取了不同类型的公式进行测试,结果表明了我们 工具的正确性和性能的优越性。

关键词：线性时态逻辑，LTL 可满足性检查，义务集合，程序验证，迁移系统，Buchi 自动机

ABSTRACT

Linear Temporal Logic(LTL) has been developed a lot since it was introduced to computer science in 1970s. Nowadays, It is widely used in various fields such as program verification and analysis, program synthesis，database and artificial intelligence.

LTL is a kind of logic language, of which the main issue is satisfiability problem. Meanwhile, satisfiability problem is also important in industry because in terms of program verification，program synthesis and artificial intelligence, an unsatisfiable formula is meaningless.

In this paper, a new algorithm named OFOA(which in chapter 3) aimed at satisfiability problem is proposed. This new algorithm provides a new set of reasoning framework that satisfiability judgement and formula expansion proceed at the same time, which is a significant optimization regarding to the traditional method that satisfiability judgement can't be conducted until an equivalent Buchi automata has been converted. According to the "on the fly" idea, we defined a new concept for LTL formula, "obligation set", which greatly accelerates the satisfiability judgement process.

Aalta, a new satisfiability tool based on OFOA algorithm was implemented which will be discussed later in the latter part of this paper about its design framework, data structure and core algorithm. To verify the efficiency of this algorithm, we designed a set of comparison experiments. We choose three most excellent satisfiability checking tools so far (PANDA, Cadence SMV

and SPOT) to do these comparison experiments and the results eventually demonstrated our tool's correctness and good performance.

Keywords: Linear Temporal Logic, LTL Satisfiability Checking, Obligation Set, Transition System, Program Verification, Buchi Automata

第一章 ： 绪论

1.1 线性时态逻辑的发展历程

对于响应式系统来说，其正确性不仅仅取决于运算的输入和输出，关键在于系统的执行序列。 时态逻辑(Temporal Logic)可以刻画逻辑中的时态性，通过对命题逻辑和谓词逻辑的扩展来描述响应式系统的无限执行序列。 通俗来讲，时态逻辑是一种包含时间因子的逻辑，即在逻辑中加入时间的因素。20 世纪 50 年代后期，Prior 提出了两种时态算子，分别用“F”和“P”来表示“将来的某个时间”及“过去的某个时间” [3]。

时态逻辑有两个分支，分别是线性时态逻辑(Linear Temporal Logic)和分支时态逻辑(Computation Tree Logic)。 线性观点认为，同一时刻，一个状态只有一个后继； 但分之观点认为是有一个树形的分支结构存在的，一个状态有多个可能存在的后继。 目前来看，两种时态逻辑的分支都已经取得了巨大进展，我们不探讨时间的本质究竟是线性的还是分支的， 在本文中，我们只关注线性时态逻辑。

20 世纪 70 年代末，Pnueli 将时态逻辑引入了计算机领域[4]， 并将其作为形式化的规范语言在计算机科学中用于并发工具的规范和验证，并因此在 1996 年获得了计算机界最高奖项—图灵奖。 1977 年，Pnueli 提出了“将来线性时态逻辑”，即现在的标准线性时态逻辑，其中包括了两个新的时态算子：X(Next)和 U(Until)。

20 世纪 70 年代末，Pnueli 将时态逻辑引入了计算机领域[4]， 并将其作为形式化的规范语言在计算机科学中用于并发工具的规范和验证，并因此在 1996 年获得了计算机界最高奖项—图灵奖。1977 年，Pnueli 提出了“将来线性时态逻辑”，即现在的标准线性时态逻辑，其中包括了两个新的时态算子：X(Next)和 U(Until)。

1.2 线性时态逻辑的研究现状

线性时态逻辑 LTL 应用在计算机领域的多个方面。

1.2.1 程序验证

线性时态逻辑首次被引入计算机科学领域就是将其作为一种行为规范语言来描述程序的性质。 许多模型检查工具都使用 LTL 来作为规范语言，工具 SPIN 使用 LTL 来描述性质， 通过判断一个给定的程序模型 M 是否能满足给定的规范 φ 验证程序的正确性。 使用 LTL 来作为行为规范语言的一个原因是 LTL 对公平性约束方便。

程序验证的方法有两种， 一种是定理证明，使用一些数学手段将模型 M 与性质 φ 形式化处理，在构建的推理系统上进行推到证明，从而判定模型 M 是否满足性质 φ。 还有一种方法就是模型检查，其验证原理与定理证明的方法不同，模型检查首先将规范 φ 取非得到¬φ，构造出与¬φ 等价的自动机 A， 将模型 M 作为所有状态均为接收状态的自动机 AM，AM 与 A 做交运算得到自动机 Ap， 在 Ap 上进行穷举搜索遍历，检查其接收的语言是否为空，若为空则判定 M 满足规范 φ，若不为空即找到了判定 M 不满足规范 φ 的一个反例，由此可说明模型 M 不满足规范 φ。

1.2.2 程序综合

程序综合问题可以描述为：给定一个性质规范 φ，能否生成一个模型 M 使得 M 满足规范 φ[5]，该问题是 1957 年 Church 在一篇文章中提出的， 同时他还提出了另外一个问题就是判定问题，给出一个程序 M 和其性质规范 φ，如何判定 M 是否满足规范 φ。 在上面介绍的程序验证中已经给出了这个问题的回答。 程序综合问题目前也已经有了解决方案，也是利用性质 φ 与自动机的转化来解决。 但是

其难度比程序验证大，其复杂程度已经达到了双指数级。

1.3 本文的选题与主要工作

本文就 LTL 的可满足性问题展开，并对其进行了深入的研究。

我们都知道，LTL 现在被广泛用于程序验证、模型检查、人工智能等领域。拿模型检查举例，人们往往使用 LTL 公式来书写性质规范，但假如所书写的规范恒真或者恒假，其实是没有做检查的意义的，因此，在做模型检查之前，事先判断性质的可满足性是非常有必要的。而在计算机理论领域，可满足性问题是一个最基本的问题，它是第一个 NP-完全问题，计算机理论中的很多算法问题终都能归约到可满足性问题，所以也是个非常重要的问题。

本文首先介绍线性时态逻辑相关的理论知识（第二章），然后在前人的基础上提出一种高效的 LTL 可满足性算法（第三章），以此为基础，我们设计开发可满足性工具 Aalta 并介绍相关算法和数据结构（第四章），最后给出实验结果，以证明我们算法的可行性和高效性（第五章）

第二章 背景知识

2.1 LTL 线性时态逻辑

2.1.1 LTL 语法

本节主要讲述 LTL 的基本语法规则。LTL 的基本组成有：原子命题，布尔运算符与或非，以及时态算子 X，F，G，U，R。其中：

- X 表示 Next 下一个状态
- F 表示某个 Future 状态
- G 表示所有将来的状态(Globally)
- U 表示 Until 直到
- R 表示 Release 释放

LTL 的语法可以如下递归定义：

$\phi ::= true \mid false \mid a \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid X\phi \mid \phi_1 U \phi_2 \mid \phi_1 R \phi_2$

其中，X φ为真表示下一状态φ要成立； φ1 U φ2 为真表示直到φ2 成立之前φ1 是成立的； φ1 R φ2 可以等价为¬(¬φ1 U ¬φ2)； G φ等价于 false R φ； F φ等价于 true U φ； F ¬φ 成立当且仅当¬(G φ)成立。

另外，我们使用|φ|来表示 LTL 公式的长度，长度计算根据公式φ中操作符的个数为标准。 公式 true、false 和原子命题公式 a 的长度为 0。 举例，公式 F(a∧b)的公式长度为 2。

2.1.2 LTL 语义

模型$M = (S, \rightarrow, L)$，其中$\pi = s_1 \rightarrow ...$为 M 中的路径，$\pi$与 LTL 公式的满足性关系⊨如下定义：

- $\pi \vDash \top$

- $\pi \nvDash \bot$
- $\pi \vDash a$ 当且仅当 $a \in L(s_1)$
- $\pi \vDash \neg\phi$ 当且仅当 $\pi \nvDash \phi$
- $\pi \vDash \phi_1 \wedge \phi_2$ 当且仅当 $\pi \vDash \phi_1$ 且 $\pi \vDash \phi_2$
- $\pi \vDash \phi_1 \vee \phi_2$ 当且仅当 $\pi \vDash \phi_1$ 或 $\pi \vDash \phi_2$
- $\pi \vDash X\phi$ 当且仅当 $\pi^2 \vDash \phi$
- $\pi \vDash \phi_1 U \phi_2$ 当且仅当存在 $i \geq 1$ 使得$\phi^i \vDash \phi_2$ 并且对于所有$j = 1, ..., i-1$都有$\pi^j \vDash \phi$
- $\pi \vDash \phi_1 U \phi_2$ 当且仅当存在 $i \geq 1$ 使得$\phi^i \vDash \phi_1$ 并且对于所有$j = 1, ..., i$都有$\pi^j \vDash \phi$

其中，π^i表示从状态s_i开始的路。

2.2 迁移系统

迁移系统在计算机科学中常用来描述系统的行为，一般用有向图来表示，其中结点表示状态，描述系统信息，有向边表示状态的迁移。 描述顺序程序的迁移系统中，结点信息一般为内存信息、PC 等； 而描述硬件电路的迁移系统中，结点信息中则包括寄存器值，当前输入等[1]。

迁移系统的定义如下： 一个迁移系统 TS 是一个六元组$(S, Act, \rightarrow, I, AP, L)$，其中

- S 为一组状态集合
- Act 为一组迁移条件集合
- $\rightarrow \subseteq S \times Act \times S$ 为迁移关系
- $I \subset S$ 是一组初始状态
- AP 是一组原子命题
- $L : S \rightarrow 2^{AP}$ 是标记函数

如果 S、Act 和 AP 均为有限集合，则该迁移系统为有限迁移系统。

2.3 $Büchi$自动机

一个$Büchi$自动机 A 是一个五元组$(S, \Sigma, \delta, S_0, F)$，其中

- S 为 A 中状态的集合
- Σ为自动机 A 的字母表
- $\delta : S \times \Sigma \to 2^S$为自动机 A 中的迁移关系
- $S_0 \subseteq S$ 为自动机 A 的初始状态集合
- $F \subseteq S$ 为自动机 A 的接收状态集合

$Büchi$自动机为确定的$Büchi$自动机当且仅当对于任意的$a \in \Sigma$ 和$s \in S$，有$|\delta(s, a)| = 1$成立；否则，该自动机为非确定的$Büchi$自动机。对于$Büchi$自动机来说，其接收序列为无限序列。下面是$Büchi$自动机接收序列的定义。

给定无限序列$\sigma = A_0A_1A_2... \in \Sigma^w$，其在自动机 A 上的运行轨迹为一串无限状态序列$q_0q_1q_2..$，其中$q_0 \in Q_0$，对于每一个$i \geq 0$都有$q_i \to q_{i+1}$。

如果执行序列可以无限次经过自动机 A 的接收状态，则该执行序列可被自动机 A 接收。

2.4 LTL 可满足性检查算法

LTL 作为一种描述程序性质的逻辑语言，其基本问题便是可满足性，不可满足的公式是没有价值的。LTL 可满足性问题如下描述，给出一个 LTL 公式ϕ，是否存在一个模型 M 使得$M \vDash \phi$，若存在这样一个模型 M，则该公式ϕ可满足，若不存在模型 M，则公式ϕ是不可满足的 LTL 公式。LTL 可满足性检查问题的输入是一个 LTL 公式，输出为满足/不可满足。

当前主流的 LTL 可满足性检查方法主要分为以下几种：

2.4.1 基于 tableau 方法的 LTL 可满足性检查

Wolper 在 1985 年的一篇文章里提出了 tableau 方法， 该方法首先根据 LTL 的基本语法将 LTL 公式转化为 Negation Normal Form(NNF)形式，然后按照时态对转化后的公式进行分解， 每一步分为当前状态和下一状态，最终可以得到一个该 LTL 公式所对应的迁移系统， 由于 LTL 公式的长度是有限的，所以该迁移系统不会无限扩大。 Schwendimann 以该方法为基础，提出了一种新的方法来进行 LTL 可满足性检查。 该方法的基本原理与 Wolper 的 tableau 方法一致，是以一种树形结构来进行信息的存储。 随后，Gore 实现了该方法并完成工具 pltl，该工具分为两种模式：一种是 pltl-tree，一种是 pltl-graph。 Gore 在实现的过程中对该方法进行了优化与改进。 我们在后面的实验环节会将自己的工具与 pltl 进行比较。

2.4.2 转化为模型检查问题的 LTL 可满足性检查

将 LTL 可满足性检查问题转化为模型检查问题的好处是可以利用现有的模型检查技术来对 LTL 可满足性检查问题进行求解。 其基本思想为：给定一个定义在原子集合 S 上的公式 ϕ，如果一个模型 M 是完全的，那么该模型包含 S 上所有的可能无限序列， 即 ϕ 是可满足的当且仅当 M 不满足 $\neg\phi$。 由此，LTL 可满足性检查问题可以利用 LTL 模型检查工具来实现。

LTL 模型检查工具主要分为两种，具体的和符号化的。

上文中所提到的工具 SPIN 是具体的模型检查工具，其会构造出完整的状态空间，利用穷举法遍历所有状态从而找到反例， 工具 SPOT 也是具体的模型检查工具，SPOT 是当前工业界非常流行的模型检查工具。 使用该类方法的模型检查工具有很多，但是他们都是用 Perl 或 Python 语言，对工具的性能有所影响，因此我们的工具使用 C 语言来实现。 符号化的模型检查工具如 CadenceSMV、NuSMV 和 VIS，则利用 BDD 的方法符号化地来表示和分析模型。

LTL 模型检查工具遵循自动机理论，其实现方式均为将给定的 LTL 公式以具体的或符号化的方式转化为等价的 Buchi 自动机， 随后模型检查工具对该自动机进行遍历以找到可接收的序列。 若可接收序列为空，则该公式不可满足。

Rozier 和 Moshe 比较了符号化的和具体的 LTL 可满足性检查工具，其结果表明符号化工具性能由于具体化工具[8]。但是他们没有将该类工具与基于 tableau 的工具和下文中将要提到的基于 SAT 的可满足性检查工具进行比较。 我们在后期的实验中对这几类工具的性能都进行了比较。

第三章 OFOA 算法框架设计

OFOA 算法是一种基于义务加速的 LTL 可满足性检查算法，对于义务这一概念的定义会在后面给出。 本算法基于前文中 Wolper 提出的 tableau 方法进行 LTL 公式的基本展开，但是核心的可满足性检查与 Moshe 的方法不同。 Moshe 在 20 世纪 90 年代后期的文章中提出的方法是将 LTL 公式展开为 GBA(Generalized Buchi Automata)， 然后将 GBA 转化为 BA，该 BA 与给定的 LTL 等价，通过检查 BA 接收语言是否为空来判断 LTL 公式的可满足性。 如果 BA 接收语言为空，则该 LTL 公式不可满足，否则，该公式可满足。

OFOA 算法直接将 LTL 公式转为Büchi Automata，相较于 Moshe 的方法减少了最终自动机的状态个数， 并且在判定可满足的过程中采用义务加速的方式加快判定， 对于无法通过义务加速方式判定的公式通过查找 SCC(最强连通图)的方式来确保该算法的完备性。 下面来详细介绍下该算法。

3.1 从 LTL 到 Buchi 自动机

首先介绍义务和义务集合的概念。

义务集合：给定一个 LTL 公式ϕ，定义其义务集合为$olg\{\phi\}$，如下：

1. $olg\{true\} = \{\emptyset\}$，并且$olg\{false\} = \{\{false\}\}$；
2. 如果$\phi = p$，那么$olg\{\phi\} = \{p\}$；
3. 如果$\phi = X\psi$，那么$olg\{\phi\} = olg\{\psi\}$；
4. 如果$\phi = \phi_1 \vee \phi_2$，那么$olg\{\phi\} = olg\{\phi_1\} \cup olg\{\phi_2\}$；
5. 如果$\phi = \phi_1 \wedge \phi_2$，那么$olg\{\phi\} = \{S_1 \cup S_2 | S_1 \in olg\{\phi\} \wedge S_2 \in olg\{\phi_2\}\}$；
6. 如果$\phi = \phi_1 U \phi_2$或者$\phi = \phi_1 R \phi_2$，那么$olg\{\phi\} = olg\{\phi_2\}$。

$olg\{\phi\}$中的每个元素$O \in olg\{\phi\}$为公式ϕ的一个义务。义务标识了公式需要被无限次满足的属性性质。

一个序列ξ若能无限次满足ϕ的义务，那么ξ就能满足公式ϕ，即$\xi \models \phi$。

每个 LTL 公式都有一个和它语义等价的析取范式 DNF，LTL 公式所描述的是一条无限序列，DNF 格式的公式抽象化地展现了当前状态与下一状态的关系。

如公式aUb，其 DNF 展开式为$b \vee (a \wedge X(aUb))$，若一个无限序列ξ满足公式aUb，即该序列ξ满足公式$b \vee (a \wedge X(aUb))$，则$\xi^0 \models b$或$(\xi^0 \models a) \wedge (\xi^1 \models (aUb))$。

由此，每个 LTL 公式都可以通过产生 DNF 公式的方式得到其对应的迁移系统。这是 LTL 专为Büchi自动机的基础步骤，最后一步为接受状态的定义。对于 R-自由的公式，将 true 定为接受状态，对于 U-自由的公式，将所有状态定义为接受状态。对于非特殊的公式，其接受状态的确定是靠义务集合来驱动的，当义务集合被满足，则当前状态为接收状态。

下面选取一个例子来介绍 LTL 到Büchi自动机的转化。

选取公式$\phi_1 = G(aUb \wedge cUd)$为例。$\phi_1$的析取范式为如下：

$\phi_1 = (b \wedge d \wedge X\phi_1) \vee (a \wedge d \wedge X\phi_2) \vee (b \wedge c \wedge X\phi_3) \vee (a \wedge c \wedge X\phi_4)$

其中

$\phi_2 = aUb \wedge G(aUb \wedge cUd), \phi_3 = cUd \wedge G(aUb \wedge cUd), \phi_4 = aUb \wedge cUd \wedge G(aUb \wedge cUd)$

$\phi_1, \phi_2, \phi_3, \phi_4$都是语义等价的且具有同样的析取范式。

与公式$G(aUb \wedge cUd)$等价的Büchi自动机如图 3-1 所示。

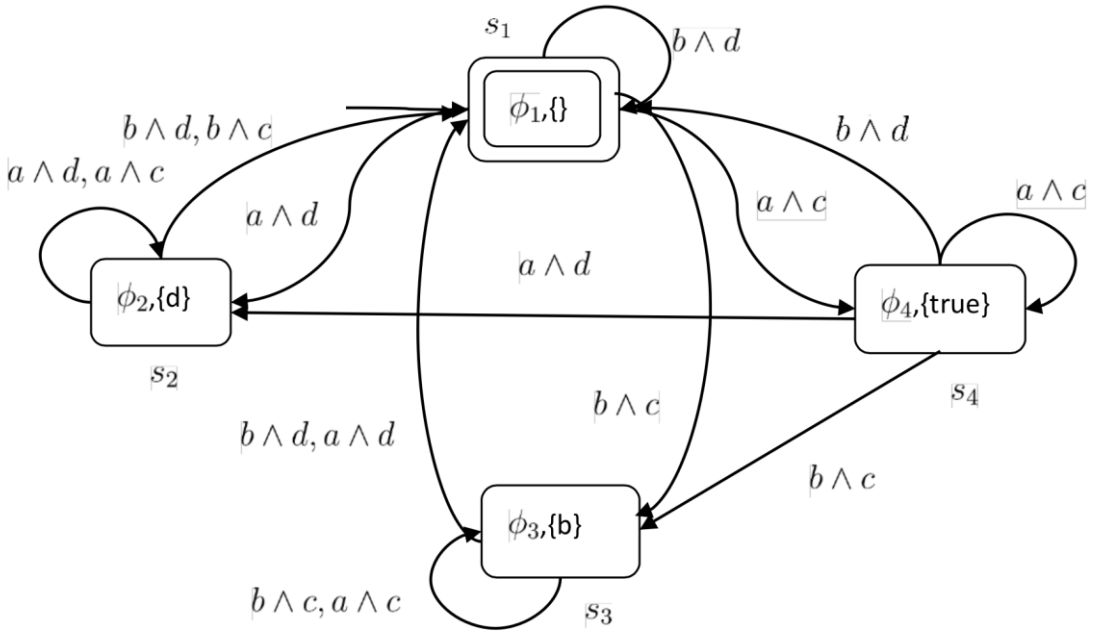

图 3-1 与公式 G(a U b ∧ c U d)等价的 Buchi 自动机

图中生成的自动机有 4 个状态，分别对应四个公式$\phi_1, \phi_2, \phi_3, \phi_4$。其中，公式$\phi$对应的状态$s_1$为初始状态。在$\phi_1$对应的析取范式中有一项为$(b \wedge d \wedge X\phi_1)$，其中$X$操作符后面的$\phi_1$代表状态$s_1$，因此状态$s_1$有一个自回路，边上标注为$b \wedge d$，。根据前面提到的义务集合计算方法，我们可以计算得出四个公式$\phi_1, \phi_2, \phi_3, \phi_4$的义务集合，其义务集合是相同的，均为 b,d，可以表示为$olg\{\phi_i\} = \{b, d\}$，$\phi_i(i = 1, , 2, 3, 4)$。义务标识了公式需要被无限次满足的属性性质，那么若公式$\phi_i(i = 1, , 2, 3, 4)$是可满足的，当且仅当义务集合{b,d} 中的每一个义务都可以被无限次满足，即无限次出现。如图所示，自动机的每个状态包含一个公式和一个过程集合，过程集合用来记录已经被满足的性质，初始状态的过程集合被置为空。图中，状态s_1的过程集合P_1为空集，状态s_2的过程集合{d}是由状态s_1的过程集合和s_1到s_2的边 a,d 的合取计算得到的，因为 a 不存在于ϕ_2的义务集合{b,d}中，因此状态s_2的过程集合P_2为{d}，状态s_3的过程集合由状态s_1的过程集合和s_1到s_3的边 b,c 的合取计算得到，其中 c 不属于义务集合{b,d}中，所以因此状态s_3的过程集合P_3为 b，同理可计算

出状态s_4对应的状态集合P_4。若目前为止义务集合中没有一个属性被满足，那么过程集合表示为$\{true\}$，如状态集合P_4，指向状态s_4的边上，公式都不包含在义务集合{b, d}中，因此状态集合P_4置为$\{true\}$。若某状态对应的过程集合是公式中某一个义务的超集，那么就将该过程集合置为空， $P_1' = P_4 \cup \{b\} = \{b, d\}$得来，因为$olg\{\phi_1\}$是过程集合$P_1' = \{b, d\}$的超集。因此$P_1'$被重新置为了空集。

3.2 义务加速检查

在将 LTL 转为Büchi自动机后，可以检查该自动机接受语言是否为空来判定给定 LTL 公式的可满足性。 通过分析自动机产生过程可以看出该自动机与 Moshe 所提出的自动机生成方法的区别，该算法直接得到一个Büchi自动机，而没有 GBA 的中间过程，因此减少了时间。 并且在 GBA 转为 BA 的过程中会产生许多的冗余状态，该算法也在一定程度上避免了冗余状态的产生。

下面介绍 OFOA 算法的义务加速检查部分。该过程以下的定理作为依靠。

义务加速检查：假定ϕ中存在一个义务$O \in Olg(\phi)$时一致，那么我们有$O^w \models \phi$成立。

例：公式$G(aRb)$，根据义务集合的计算方法，该公式的义务为$\{b\}$，无限序列b^ω能够无限次消除义务$\{b\}$，因此$b^\omega \models G(aRb)$。

但是该方法是不完备的，如果公式不能通过义务加速检查的方式判定其可满足性，不能直接认定该公式是不可满足的。

例：公式$Fb \wedge G(X\neg b)$，通过上一小节中义务集合的计算方法我们可以得到该公式对应的义务集合为$\{b, \neg b\}$，其中包含有不一致的义务。因此我们不能通过义务加速的方法快速判定该公式的可满足性，同时也不能简单判定该公式不可满足，无限序列$\{b\}\{\neg b\}^w$可以满足公式$Fb \wedge G(X\neg b)$。

通过义务集合的一致性来判定公式可满足性只是一种加速方式，这样的可满足性检查时不完备的。

3.3 基于义务集合的可满足性检查

在无法通过义务加速方式来判定的时候，同样有另外一套方法来进行 LTL 公式的可满足性检查。工具 Aalta_v1.0 实现了该方法，通过实验证明了 OFOA 算法的有效性。定理如下：

LTL 可满足性检查的中心定理：令 λ 为输入公式。那么$SAT(\lambda)$成立当且仅当在T_λ中存在一个 SCC（最强连通图）B，并且在 B 中存在一个状态ψ使得$L(B)$是ψ中一个义务的超集，即存在$O \in olg\{\psi\}$使得$O \subseteq L(B)$成立。

下方的图形化描述可以更好地理解 LTL 可满足性检查的中心定理。

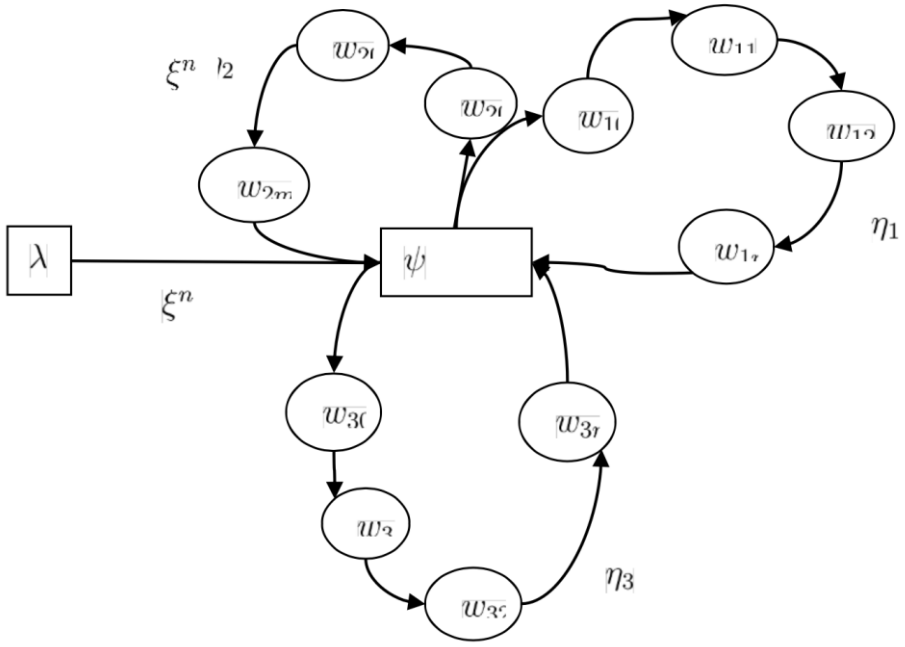

图 3-2 中心定理的图形化描述

3.4 基于 SAT 技术的 LTL 可满足性检查

为了利用 SAT 求解器的性能进行加速，有一种新的 LTL 可满足性检查框架，将上文中的 OFOA 可满足性检查算法进行优化与改写，在新的框架下实现得到了工具 Aalta_v2.0。

SAT 求解器的输入为一个布尔公式，因此需要用义务公式来替代义务集合。

义务公式：给定一个 LTL 公式ϕ，其对应的义务公式标记为$of(\phi)$，递归定义如下：

1．$of(true) = true$，并且$of(false) = false$；

2．如果$\phi = p$，那么$of(\phi) = p$；

3．如果$\phi = X\psi$，那么$of(\phi) = of(\psi)$；

4．如果$\phi = \phi_1 \vee \phi_2$，那么$of(\phi) = of(\phi_1) \vee of(\phi_2)$；

5．如果$\phi = \phi_1 \wedge \phi_2$，那么$of(\phi) = of(\phi_1) \wedge of(\phi_2)$；

6．如果$\phi = \phi_1 U \phi_2$或者$\phi = \phi_1 R \phi_2$，那么$of(\phi) = of(\phi_2)$。

根据以上定义计算得到的义务公式本质上是一个布尔公式。

在进行基于 SAT 的可满足公式加速时，可以在每个新状态产生时计算其义务公式，将义务公式作为 SAT 求解器的输入，若该义务公式可满足，则表明在$Olg(\phi)$中存在一致的义务。

对于无法通过加速检查方式判定可满足性的 LTL 公式，同样利用上一章节的中心定理进行判定，最终圈的判定问题也可归约为 SAT 求解问题。

第四章 工具详细设计与实现

4.1 数据结构

4.1.1 公式结构

我们使用二叉树结构来存储公式结构，其中分支节点存储操作符，叶子节点存储变量值。 假设有公式(a ∧ b ∨ ¬b ∧ c) U d，用二叉树来表示见下图：

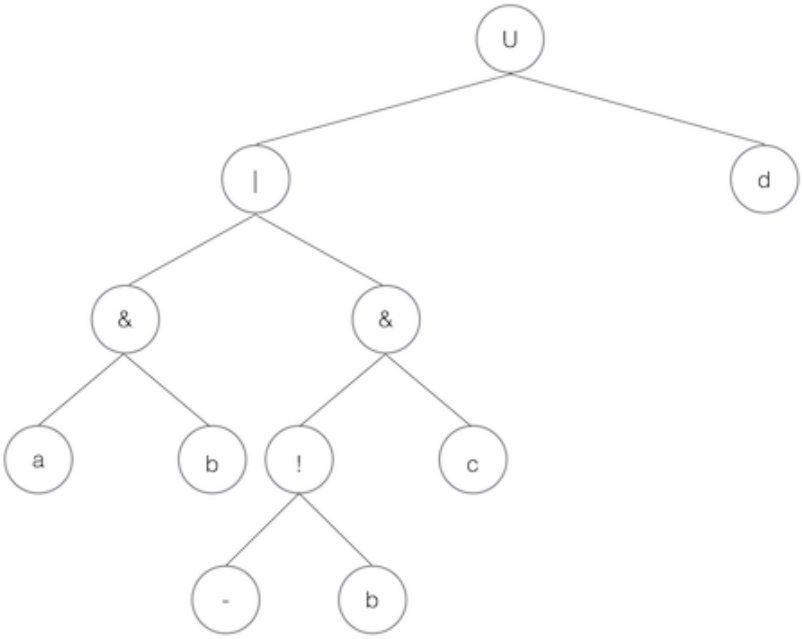

图 4-1 公式结构

该图的根节点为 Until 算子，右子节点为一个叶子节点，存放变量 b，该节点我们也称之为原子节点； 左子节点为一棵以 Or 算子为根的子树，儿子节点为两棵以 And 算子为根的子树， 其中右侧子树的左节点是一棵 Not 算子的子树，该子树左节点为空，用-表示，有节点是一个原子节点。

由上可知，要还原公式，只需要先序遍历该二叉树即可。 与此同时，叶子节点为-的节点表示空节点，用于表示一元运算符（¬、X）的表达式

4.1.2 DNF 结构

公式展开方面，我们使用图结构来表示，进过分析，这是一个带边信息的有向图，且含有重边。 依旧用上面的那个例子，该公式所对应的 DNF 如下图：

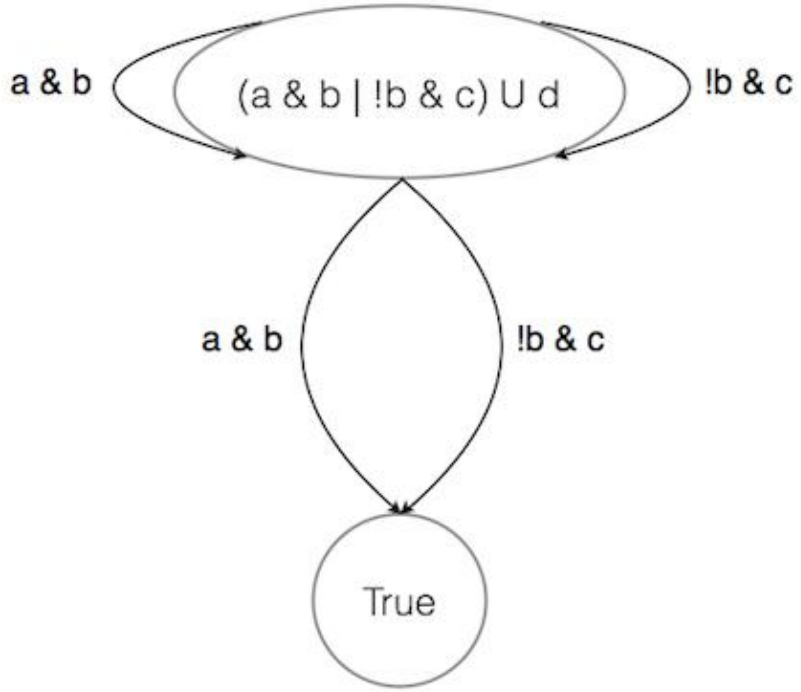

图 4-2 DNF 结构

从上图可知，节点(a ∧ b ∨ ¬b ∧ c) U d 有两条自环，自环上的信息依次是 a ∧ b 和¬b ∧ c， 同时有两条路径指向节点 True，而边上的信息也是 a ∧ b 和¬b ∧ c。

然后，该图中的重边是可以去除的，去除重边后等价于下图：

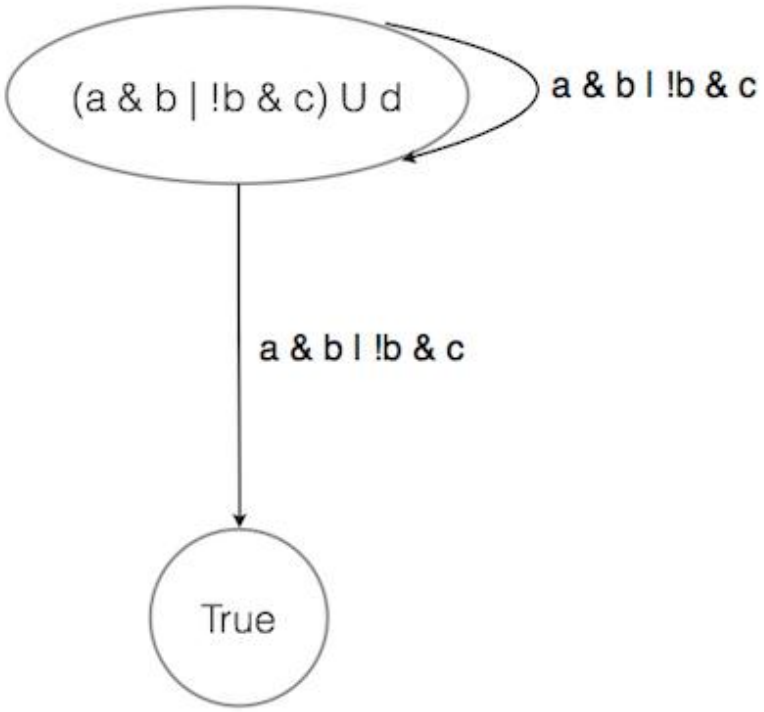

图 4-3 去重边的 DNF 结构

因此，我们可以用邻接链表来存储整张图，而为了加速查找邻接节点用户合并边，可以使用 hash 表来替换链表结构。

4.1.3 存储结构

在工具运行过程中，我们需要将公式信息保存下来，这里有几点因素的考虑：

1. DNF 公式展开的时候需要判定公式的唯一性（关于公式唯一性的判定将在下一小节深入展开，这里不做详述。）
2. 若公式足够复杂，将会产生极其庞大的中间公式，有可能是内存所无法承受的，这时我们需要借用数据库等外存设备，这也是引入存储层的重要原

因。

存储结构方面，我们借用存储公式的树结构，设计以下表：

字段名	数据类型	字段描述
id	Number	公式 id，唯一，自增，从 1 开始
leftId	Number	左节点 id，0 表示空节点
rightId	Number	右节点 id，0 表示空节点
oper	Number	若子节点 id 都为 0，表示变量编号；否则，表示运算符

表 4-1 存储结构

其中运算符是事先规定好的枚举； 变量编号从 2 开始，0 表示 False，1 表示 True，此后以遍历顺序编号，若最先遇到变量 a，则 a 的编号为 2，而后遇到变量 b，则 b 的编号为 3，依次类推。

具体实现上，我们可以使用 hash 表存储，并使用单例模式

4.2 核心算法

4.2.1 规范化

在做公式的其他操作之前，先要对公式进行规范化操作，以方便其他操作。 公式的规划化操作包含以下两类操作：

1. 去除↔、→、G 和 F 算子，我们通过以下规则进行改写：
   a) $a \rightarrow b \equiv \neg a \vee b$ (N1.1)
   b) $a \leftrightarrow b \equiv (\neg a \vee b) \wedge (\neg b \vee a)$ (N1.2)
   c) $G\ a \equiv False\ R\ a$ (N1.3)
   d) $F\ a \equiv True\ U\ a$ (N1.4)
2. 保证¬算子只出现在原子前，我们通过以下规则进行改写：

a) ¬True ≡ False (N2.1)

b) ¬False ≡ True (N2.2)

c) ¬¬a ≡ a (N2.3)

d) ¬(Xa) ≡ X(¬a) (N2.4)

e) ¬(G a) ≡ True U ¬a (N2.5)

f) ¬(F a) ≡ False R ¬a (N2.6)

g) ¬(a U b) ≡ ¬a R ¬b (N2.7)

h) ¬(a R b) ≡ ¬a U ¬b (N2.8)

i) ¬(a ∧ b) ≡ ¬a ∨ ¬b (N2.9)

j) ¬(a ∨ b) ≡ ¬a ∧ ¬b (N2.10)

k) ¬(a->b) ≡ a ∧ ¬b (N2.11)

l) ¬(a<->b) ≡ (a ∧ ¬b) ∨ (¬a ∧ b) (N2.12)

举例 考虑公式¬((a -> Xb) U (Ga))， 运用规则 N1.1 和 N1.3 ，可以将公式转化为¬((¬a ∨ Xb) U (False R a))； 继续运用规则 N2.7，可以得到¬(¬a ∨ Xb) R ¬(False R a)； 再使用规则 N2.10 和 N2.8，得到(¬¬a ∧ ¬Xb) R (¬False U ¬a)；最后使用规则 N2.2 、N2.3 和 N2.4，公式最终规范化为(a ∧ X(¬b)) R (True U ¬a)。

4.2.2 化简

化简有助于缩短公式长度，最大程度的加速公式判定，因此在做完公式规范化以后，我们使用以下规则对公式进行化简：

1. And 算子的化简

a) True ∧ a ∧ ... ≡ a ∧ ... (S1.1)

b) False ∧ a ∧ ... ≡ False (S1.2)

c) a ∧ a ∧ ... ≡ a ∧ ... (S1.3)

d) 若 a 和 b 是冲突的，则 a ∧ b ∧ ... ≡ False（由于冲突算法过于冗余，算法过程见附）

2. Or 算子的化简

a) False ∨ a ∨ ... ≡ a ∨ ... (S2.1)

b) True ∨ a ∨ ... ≡ True (S2.2)

c) a ∨ ¬a ∨ ... ≡ True (S2.3)

d) a ∨ a ∨ ... ≡ a ∨ ... (S2.4)

e) a ∨ b U (¬a ∨ ...) ≡ True (S2.5)

f) b U (a ∨ ...) ∨ c U (¬a ∨ ...) ≡ True (S2.6)

g) a ∨ b R a ∨ ... ≡ a ∨ ... (S2.7)

h) a ∨ b U a ∨ ... ≡ b U a ∨ ... (S2.8)

3. Next 算子的化简

a) X True ≡ True (S3.1)

b) X False ≡ False (S3.2)

4. Until 算子的化简

a) False U a ≡ a (S4.1)

b) a U False ≡ False (S4.2)

c) a U True ≡ True (S4.3)

d) a U (a ∨ ...) ≡ a ∨ ... (S4.4)

e) a U (a U b) ≡ a U b (S4.5)

f) a U (b U a) ≡ b U a (S4.6)

g) a U (b R a) ≡ b R a (S4.7)

h) (b R a) U a ≡ a (S4.8)

i) (a U b) U a ≡ b U a (S4.9)

j) (b U a) U a ≡ b U a (S4.10)

k) X a U a ≡ X a ∨ a (S4.11)

l) X a U X b ≡ X (a U b) (S4.12)

5. Release 算子的化简

a) True R a ≡ a (S5.1)

b) a R False ≡ False (S5.2)

c) a R True ≡ True (S5.3)

d) a R (a ∧ ...) ≡ a ∧ ... (S5.4)

e) (a ∨ ...) R a ≡ a (S5.5)

f) a R (a R b) ≡ a R b (S5.6)

g) a R (b R a) ≡ b R a (S5.7)

h) a R (b U a) ≡ b U a (S5.8)

i) (b U a ∨ ...) R a ≡ a (S5.9)

j) (a R b) R a ≡ b R a (S5.10)

k) (b R a) R a ≡ b R a (S5.11)

l) X a R X b ≡ X (a R b) (S5.12)

m) ¬a R a ≡ False R a (S5.13)

n) (b R (¬a ∧ ...) ∧ ...) R a ≡ False R a (S5.14)

举例 考虑公式(((a ∨ b) R (c R (a ∨ b)) U (a ∨ b)) R a) ∧ a， 运用规则 S5.7，可以将公式转化为((c R (a ∨ b)) U (a ∨ b) R a) ∧ a； 继续运用规则 S4.8，可以得到((a ∨ b) R a) ∧ a； 再使用规则 S2.7 和 N2.8，得到 a ∧ a； 最后使用规则 S1.3，公式最终简化为 a。

由上面的例子我们会发现，一个看起来很长很复杂的式子通过简化之后，有可能只是一个很简单的公式， 这也是判 LTL 可满足性之前需要对公式进行简化的重要性。

4.2.3 一致性判断

在工具运行过程中，我们需要判断两个公式是否是同一个公式， 假如已字符串的方式或者树同构的方式比较两个公式的一致性，其算法复杂度将达到 O(n)，当公式一长，所要比较的中间公式一多，其效率将是无法接受的。 所以，我们需要一个可以用来唯一标识公式的方法，并且该标识的计算复杂度小于 O(n)。

观察我们的公式结构我们会发现，假如左右子公式存在一个唯一的标识来标记， 那么，对于 Not 算子、Next 算子、Until 算子以及 Release 算子，我们只需要对比操作符和左右子公式的唯一标识就能判定两个公式的一致性了。 但是，对已 And 算子和 Or 算子，由于这两个算子符合交换率，即便两个公式树结构不一致，仍然有可能表示同一个公式， 如下图，这两棵公式树都表示公式 a ∧ b ∧ c ∧ d。

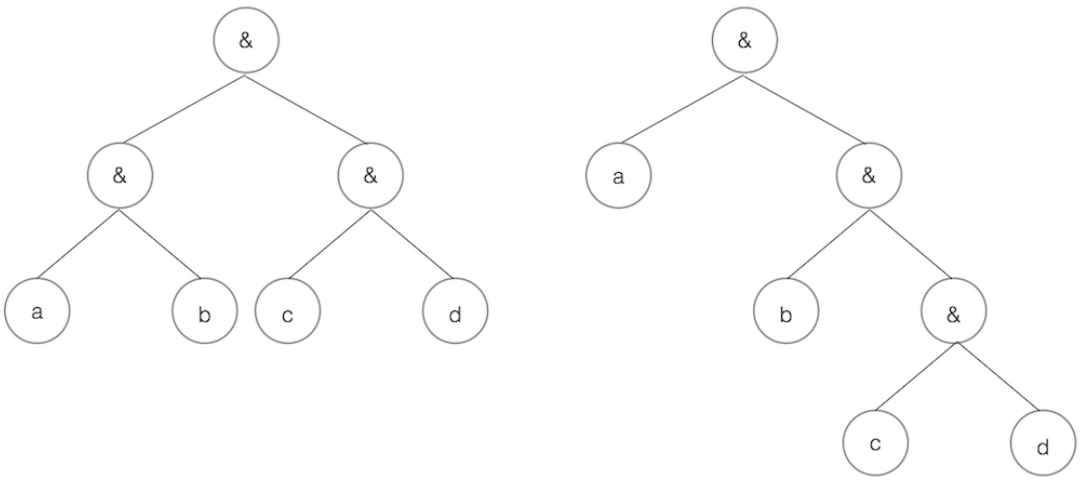

图 4-4 相同公式的不同树结构

因此，我们需要为 And 算子和 Or 算子规定一种形态，以防止其多态化。 以 And 算子为例，假如我们有公式 (a1 ∧ a2 ∧ ⋯ ∧ an)， 我们给出构造算法的伪代码：

```
Tree build_and_tree(a1, a2, ..., an){
  sort(a1, a2, ..., an);
  tree := make_tree(an);
  for (n = n-1; i > 0; --i)
    tree := make_tree(AND, ai, tree);
  return tree;
}
```

举个例子，还是公式 a ∧ b ∧ c ∧ d，且假如 a、b、c、d 的唯一标识分别为 2、4、1、3 时， 通过上述算法进行构造后，树的形态如下图：

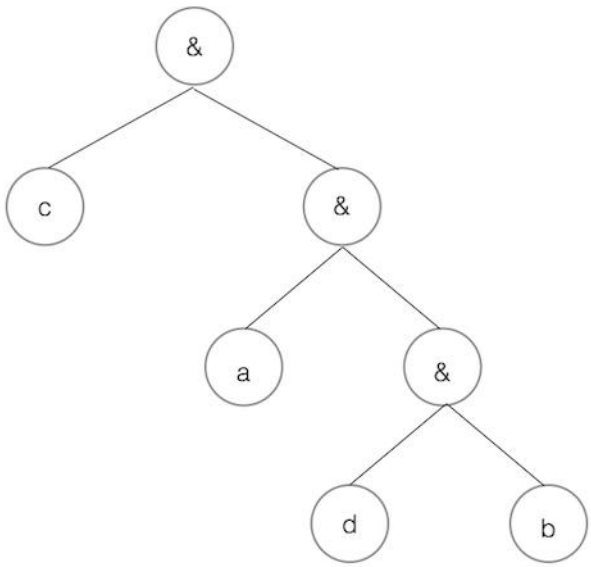

图 4-5 通过算法固定结构

Or 算子的构造算法同 And 算子。

由上述方式，我们可以保证公式的唯一性， 与此同时，由于公式树的叶子节点就是原子变量，我们可以通过变量名来对叶子节点进行唯一标识， 因此，通过数学归纳法可知，我们可以为所有公式做唯一标识，且每次标识，其计算复杂度为 O(1)。

这其实是一种以空间换时间的方法，尽管我们的时间复杂度降低了，但是，我们需要将所有的子公式记录下来。 这里，我们只需要记录当前公式的 id，左右子公式的 id 以及操作符 oper 就可以了，这也是上一小节存储结构中那张存储表的由来。

4.2.4 寻找强连通分量

我们使用 Tarjan 算法来寻找强连通分量[9]，以下简单介绍下 Tarjan 算法的流程。

Tarjan 算法基于深度优先搜索算法（Deep First Search），而每棵搜索树上的子树就是一个强连通分量。 搜索时，我们使用一个堆栈来存储当前搜索树上未处理的节点，回溯时，可以判断栈顶到栈中的节点是否为一个强连通分量。

为此，我们定义 DFN(u)为节点 u 搜索的次序编号(即时间戳)，Low(u)为 u 或 u 的子树能够追溯到的最早的栈中节点的次序号。 而当 DFN(u)=Low(u)时，以 u 为根的搜索子树上所有节点是一个强连通分量。

算法伪代码如下：

```
tarjan(u)
{
    DFN[u]=Low[u]=++Index
    Stack.push(u)
    for each (u, v) in E
        if (v is not visted)
            tarjan(v)              // 递归
            Low[u] = min(Low[u], Low[v])
        else if (v in S)
            Low[u] = min(Low[u], DFN[v])
```

```
    if (DFN[u] == Low[u])   // 如果节点 u 是强连通分量的根
        repeat
            v = S.pop   // 将 v 退栈，为该强连通分量中一个顶点
        until (u == v)
}
```

当遍历到节点 u 时，我们初始化该节点的 DFN 和 LOW 为当前时间戳，然后将 u 入栈； 然后遍历 u 的邻接节点，若该邻接节点 v 未被遍历过，对其进行递归调用该算法， 回溯时，将 u 节点的 LOW 更新为 DFN(u)、Low(v)、DFN(v)的最小值； 遍历完之后，如果 u 的 DFN 和 Low 相等，说明我们找到了一组强连通分量， 我们对堆栈进行出栈操作，直到出栈的节点为 u，此时，拿到的这些点就是一组强连通分量。

我们可以发现，在运行该算法的过程中，每个节点都被访问了一次，并且只进出堆栈一次，每条边页只被访问了一次，所以 Tarjan 算法的时间复杂度为 O(V+E)。

求有向图的强连通分量还有另一个算法，叫 Kosaraju 算法。 Kosaraju 是通过对有向图及其逆图两次 DFS 的来求强连通分量的，其时间复杂度也是 O(N+M)。与 Trajan 算法相比，Kosaraju 算法可能更方便理解， 但是 Tarjan 只需对原图进行一次 DFS，不需要建立逆图，代码实现上更为简洁。 而在实际的测试中，Tarjan 算法的运行效率一般也比 Kosaraju 算法高 30%左右。 因此这里我们选择 Trajan 算法来作强连通分量的实现。

4.3 Aalta 的实现

有了前两个小节对数据结构和核心算法的介绍之后，这一小节我们来介绍 Aalta 工具的具体实现。 下图展示了工具 Aalta 的整体框架：

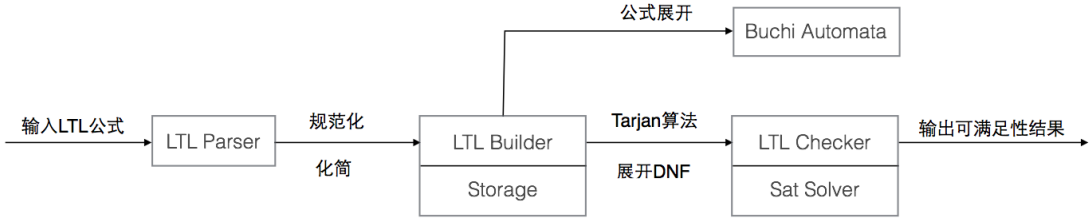

图 4-6 工具框架

由框架图中我们可以看出，整个工具分成输入、输出、语法解析器、公式构建器、可满足性检查器这几部分。

当我们向 Aalta 工具输入公式时，工具首先会用 LTL 语法解析器对输入进行语法解析，得到一棵初步的抽象语法树结构。 然后对该结构进行规范化和简化处理得到精化的 LTL 公式结构。 最后，拓展 Tarjan 算法对该 LTL 公式进行展开，展开的同时， 对遍历到的公式节点进行义务加速检查，若可以判定，给出判定结果，结束程序，否则继续展开； 与此同时，对找到的强连通分量，判定其可满足性，若不可判定，继续展开。 若全部展开完毕仍无法判定，表示该公式不可满足。

另一方面，当我们对公式进行展开而不做判定时，最后得到的就是一个 Buchi 自动机。

4.3.1 输入

Aalta 的输入是线性时态逻辑公式的字符串，并且可以处理命题逻辑算子（与、或、非、条件、双条件）和时态逻辑算子（Next、Until、Release、Global、Future），下表列出了工具可以接受的操作符以及在公式中对应的符号。

操作符名称	操作符符号
NOT	!, ~
AND	& , &&
OR	|, ||
Next	X
Until	U
Release	R,V
Global	G, []
Future	F, <>
→	->
↔	<->
True	true, TRUE
False	false, FALSE

表 4-2 运算符对应表

4.3.2 输出

Aalta 的输出是关于输入公式的可满足性结果(sat/unsat)。 如果公式是可满足的，将给出公式可满足的证据，并且使用小括号来表示无限序列， 比如说，假如输出的证据是“a(xy)”，这其实表示的是无限序列“$a(xy)^{\omega}$”。

4.3.3 语法解析器

语法解析方面，我们借用 FLEX 做文法分析、Bison 做语法分析。

Flex 的前身是 lex（Lexical Analyzer Generator）， 是 1975 年由 Mike Lesk 和当时尚在 AT&T 实习的 Eric Schmidt 共同完成的基于 UNIX 环境的词法分析器的生

成工具，是 UNIX 标准应用程序。 无奈由于自身效率低下且存在 bug，被人诟病。后来伯克利实验室的 Vern Paxson 用 C 重新写了 lex，并命名为 Flex（Fast Lexical Analyzer Generator）。 顾名思义，FLEX 由于其能高效的处理词法分析而被广泛使用。

Bison 的前身是 Yacc，是由贝尔实验室的 S.C.Johnson 基于 Knuth 的 LR 分析技术，于 1975～1978 年写成。 1987 年 UC Berkeley 的 Bob Corbett 在 BSD 下重写了 Yacc。 再后来 GNU Project 接管了项目，添加了很多特性，形成了今天的 GNU Bison。

借用这两大工具，我们能很方便的获取初步的 LTL 公式所对应的抽象语法树结构。

4.3.4 公式构建器

拿到 LTL 公式的初步语法树之后，我们使用上一小节中提到的规范化和简化算法，对该结构进行精化， 而此后所有生成的新公式都将通过 LTL Builder 来进行构造。 通过一致性判断可知，每个公式只需要保存其左右子树的唯一标识和操作符，便能判定其唯一性， 因此，我们需要记录所有公式和其子公式的结构和唯一标识。 所以，工具中的公式构建器搭建在一层存储之上，同时方便拓展。

对于存储层而言，我们无需创建多个实例，所以，我们使用单例模式来实现存储层，来保证单例类的实例都只存在一个，同时减少开销。

另一方面，鉴于我们可能替换整个存储结构，所以选择抽象工厂模式，来增强整个代码框架的可扩展性。

4.3.5 可满足性检查器

由于我们的算法是一个 on-the-fly 的算法，因此，我们每展开一次公式，都会对当前状态进行判定，判定结果通过可满足性检查器来得到。 从框架图中我们能

看到，可满足性检查器是基于 Sat 求解器的，在工业界，Sat 问题已经研究了很多年，拥有众多工具来实现， 在这里，我们选择一款开源 Sat 工具——MiniSat 求解器[10]，这也是我们整个工具的核心部分。

在公式展开的过程中，我们需要对以下两种情况进行可满足性检查：

1. 对于每一个访问到的节点公式，我们运用之前提到的算法，去判断义务集合的可满足性， 并转化为一个 sat 问题，借用成熟的 sat 求解工具来求解；
2. 对于搜索到的强连通分量，我们同样运用之前提到的算法，记录边上的信息，转化为圈的判定问题，并最终使用 sat 工具来判定。

在检查过程中，如果能判定公式的可满足性或不可满足性，我们直接给出结果，否则，将继续对该节点进行展开，直到遍历完所有节点。

第五章 实验结果

为了验证我们所提出的动态的 LTL 可满足性检查方法的正确性和优越性，我们对比现有的最出色的可满足性检查工具设计了一系列的实验。

5.1 实验工具

首先，我们将该方法集成到自己的工具 Aalta 当中。 在集成后的 Aalta 工具当中包含两种可满足性检查的配置方案： 不包含义务加速的动态检查技术（OF）和包含义务加速的动态检查技术（OFOA）。 其中，后者为工具默认的配置方案。 然后，我们选取目前公认的最出色的可满足性检查工具进行比对。 这些工具分别是 PANDA＋Cadence SMV 和 SPOT。

5.2 实验平台

为了方便的计算实验时间以及更好的进行对比实验，我们的实验平台选用了 SUG@R 集群。 SUG@R 集群是美国莱斯大学开发的由一系列 Intel 赛扬处理器构建的微机系统。 该微机系统共包含 1024 个处理器。 我们实验当中的每一个测试用例都独自占有其中的有个处理器。 在实验时间的统计方面，我们使用 Unix 的“time”时间命令。 计时策略始于输入测试公式，终止于工具给出实验结果。 并且规定每个用例的运行时长不超过 10 分钟。

5.3 实验输入

为了很好的测试 LTL 可满足性检查工具，我们选取了不同类型的公式，包括：随机公式和模式公式。 其中，随机公式超过了 60,000 个，模式公式包含 8 种长度由 1 到 1000 的公式。 我们是从文献[11, 12, 13]得到的这些公式。 除此之外，我们也引入了一组新的测试集公式，我们将其命名为随机合取公式。 这种公式是 LTL

可满足性检查的难点：即一些小公式的合取。 假设随机合取公式有 n 个子公式。这些公式的获取过程分三步： 首先，文献[14]介绍了 44 种经常使用的模式，我们抽取出这 44 种模式； 然后，我们从这 44 种模式种随机选取 n 种模式； 最后，为这 n 种模式的原子变量随机赋值。 这样，我们就得到了随机合取公式。 在实验当中，我们为每一个 n 生成了 500 个随机合取公式。

5.4 实验结果

整体的实验结果表明，Aalta 不仅可以确保 LTL 可满足性检查的正确性，而且比 PANDA＋Cadence SMV 和 SPOT 的性能要优越。

有关 Aalta 的正确性，我们采用了这样的判断方法：假定 PANDA＋Cadence SMV 和 SPOT 对测试集公式的 LTL 可满足性检查的结果是正确的， 那么将 Aalta 的结果与其进行对比，如果结果一致，那么 Aalta 的检查结果也是正确的，反之不正确。 统计所有的实验结果表明，Aalta 的检查结果均与 PANDA＋Cadence SMV 和 SPOT 的结果一致， 亦即可以确保 Aalta 的 LTL 可满足性检查的正确性。

有关 Aalta 相较 PANDA＋Cadence SMV 和 SPOT 的优越性，我们将针对不同种类的公式进行以下的分类比对。

- **对于随机公式的检查，Aalta 的表现最优**

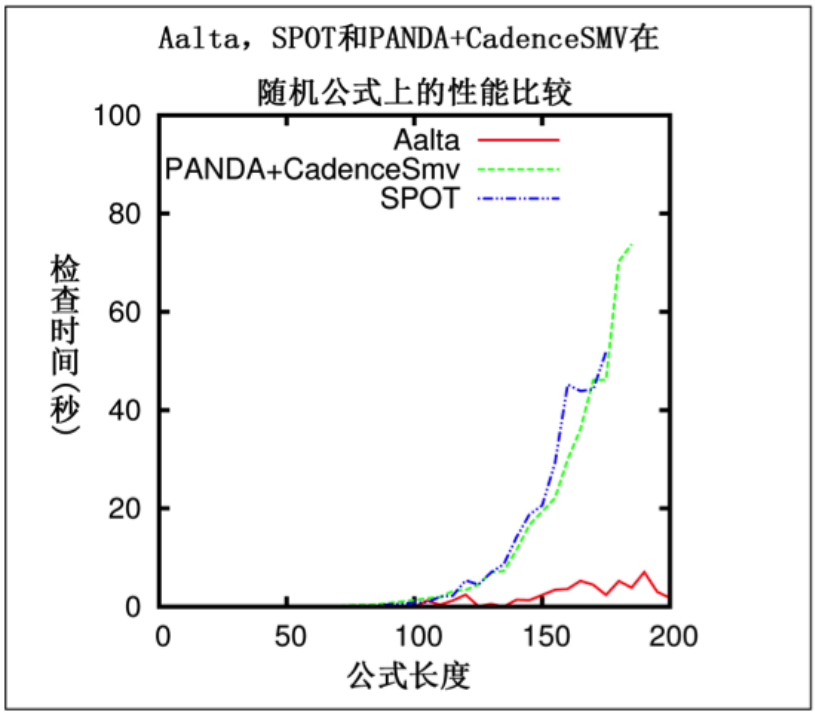

图 5-1 随机公式上的性能比较

我们对于三个工具在随机公式上的性能比较如图 5-1 所示。 在随机公式的测试中，我们选取了 20,000 个随机公式，每个公式设定了 3 个原子变量。在此基础上，改变公式的长度，由 5 到 200 以 5 的倍数递增。 图 1 中的曲线反映了针对不同的工具，LTL 可满足性检查的时间与公式长度的关系。 对于某个特定的公式长度，其对应的检查时间是取自这个长度公式的平均检查时间。 有图可知，Aalta 的时间性能明显优于其他两种工具。 Aalta 的检查时间对于 60%以上的测试公式可以达到毫秒级，而对于 PANDA＋Cadence SMV 和 SPOT 则需要十几秒。 检查完 20,000 个公式，Aalta 可以在一小时之内完成，而 PANDA＋Cadence SMV 和 SPOT 则要 40 个小时以上的时间。

Aalta 在随机公式上的时间性能之所以优于其他两个工具，是因为可满足性的公式达到 95%， 并且可以用义务加速技术来检查的 LTL 公式多达 80%左右。 虽然，Aalta 在可满足公式上存在优势，但是，PANDA＋Cadence SMV

在不可满足性公式上的性能是要优于 Aalta 的。 然而，综合这两方面的对比结果，Aalta 在整体上还是要优于 PANDA＋Cadence SMV 和 SPOT 的。

- **对于大部分的模式公式，Aalta 的表现最优**

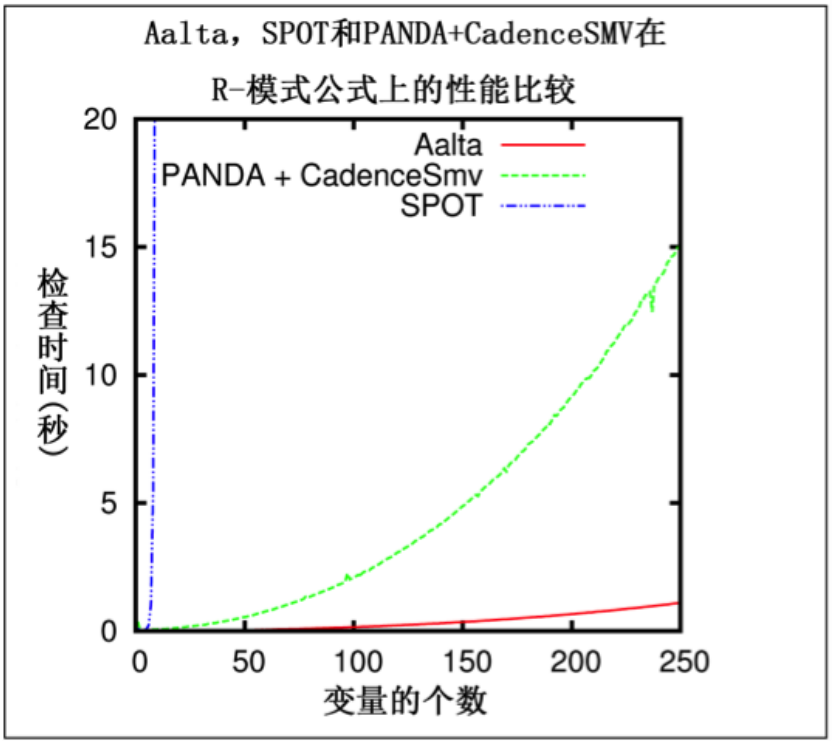

图 5-2 R-模式公式上的性能比较

如图 5-2 所示，三个工具的在 R-模式公式上的性能曲线表明，Aalta 的性能最优。 对于其他两个工具，PANDA＋Cadence SMV 的时间性能随着公式的长度呈现指数增长，SPOT 呈现线性增长， 因此 R-模式公式上 Aalta 的性能最优。

Aalta 之所以有这样的性能优势得益于义务加速技术。 否则，Aalta 的性能也是呈现指数增长的。

如图 5-3 所示，三个工具在 S-模式公式下的性能曲线表明，Aalta 的性能仅次于 SPOT，呈现线性增长。 SPOT 优于 Aalta 的原因在于它在构造自动机方面比 Aalta 快得多，而这样的优势是在长达十年的维护与优化中累积而来的。

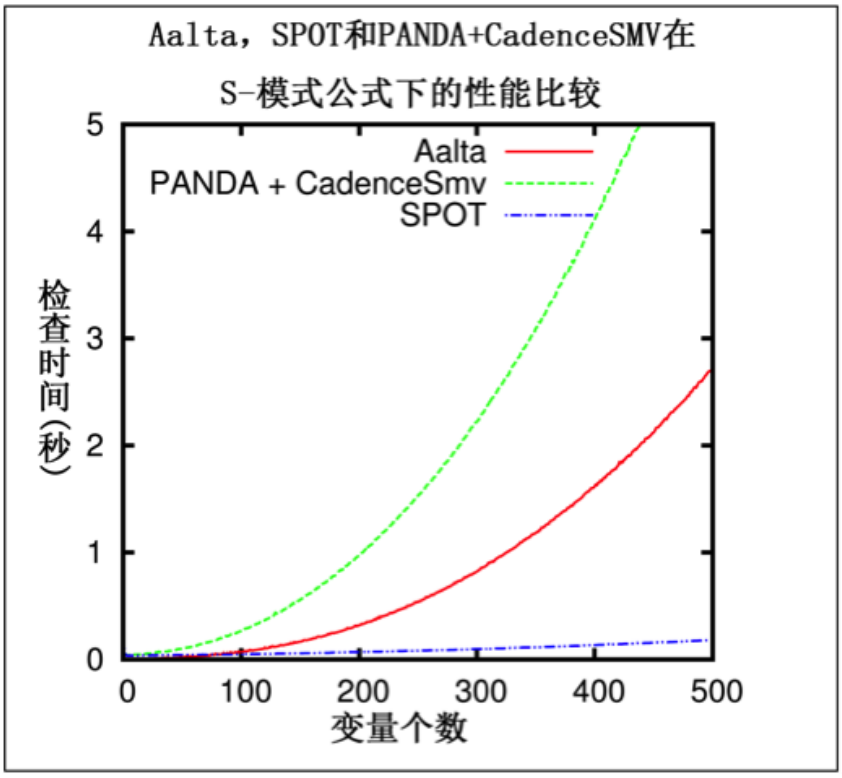

图 5-3 S-模式公式上的性能比较

因此，除了 S-模式公式之外，Aalta 对于大部分的模式公式的时间性能是最优的。

- **对于随机合取公式，Aalta 的表现最优**

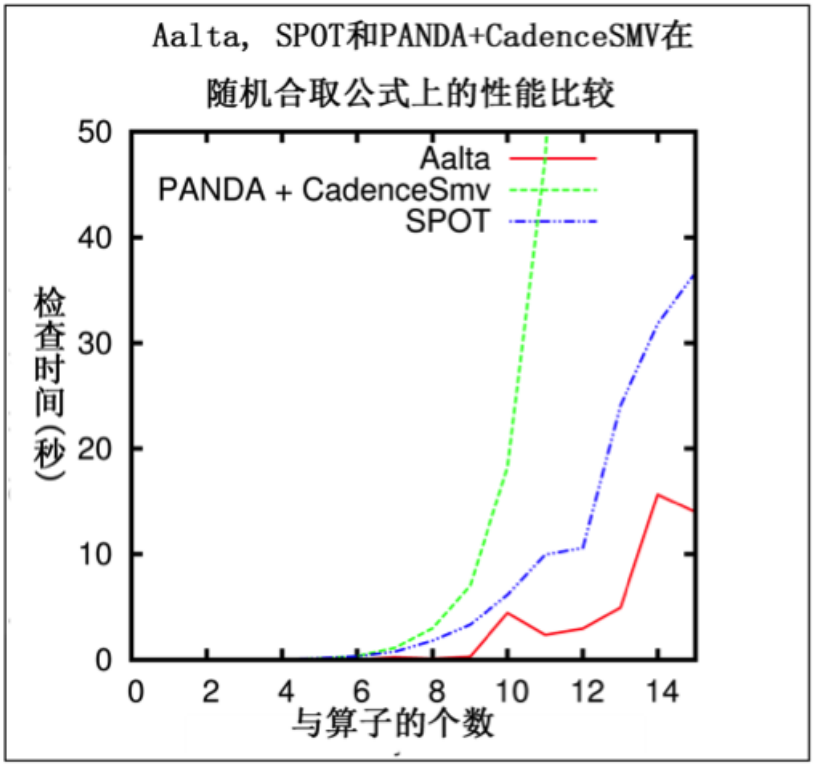

图 5-4 随机合取公式上的性能比较

如上图所示，三个工具在随 机合取公式上的性能对比曲线表明：Aalta 的表现仍然最优，SPOT 次之，PANDA＋Cadence SMV 最末。 在这部分公式的性能比对当中，考虑到长度太大的公式三个工具都将无法处理， 因此选定的公式的长度都在 100 左右，即，合取符号的数目在 1 到 15 之间。

虽然 Aalta 在随机合取公式上的性能相对最优，但是也仅仅是其他两个工具的两倍。 原因在于，Aalta 在处理可满足性公式上有很大优势，而对于不可满足公式则要差一些。 而实验中的随机合取公式中不可满足性公式的比例相对不较大，因此大大削弱了 Aalta 在可满足性公式上的优势。

参 考 文 献

[1] Principle of Model Checking.

[2] Logic in Computer Science.

[3] A. Prior. Time and Modality. Oxford University Press, 1957.

[4] A. Pnueli. The temporal logic of programs. In Proc. 18th IEEE Symp. on Foundations of Computer Science, pages 46–57, 1977.

[5] A. Church. Applicaton of recursive arithmetics to the problem of circuit synthesis.In Summaries of Talks Presented at The Summer Institute for Symbolic Logic,pages 3–50. Communications Research Division, Institute for Defense Analysis,1957.

[6] Wolper, P.: The Tableau Method for Temporal Logic: AnOverview. Logique et Analyse 28(110-111) (1985).

[7] Schwendimann, S.: A New One-Pass Tableau Calculus for PLTL. In: de Swart, H. (ed.) TABLEAUX 1998. LNCS (LNAI), vol. 1397, pp. 277–292. Springer, Heidelberg (1998).

[8] Rozier,K., Vardi, M.: LTL Satisfiability Checking. STTT12(2)(2010)

[9] Tarjan, R. E. Depth-first search and linear graph algorithms. SIAM Journal on Computing, 1 (2): 146–160, 1972

[10] G. De Giacomo and M.Y. Vardi. Automata-theoretic approach to planning for temporally extended goals. In Proc. European Conf. on Planning, pages 226–238. Springer, 1999.

[11] K.Y. Rozier and M.Y. Vardi. LTL satisfiability checking. In Proc. 14th International SPINWorkshop, volume 4595 of Lecture Notes in Computer Science, pages149–167. Springer, 2007.

[12] K.Y. Rozier and M.Y. Vardi. LTL satisfiability checking. Int'l J. on Software Tools for Technology Transfer, 12(2):1230–137, 2010.

[13] K.Y. Rozier and M.Y. Vardi. A multi-encoding approach for LTL symbolic satisfiability checking. In Proc. 17th Int'l Symp. on Formal Methods, volume 6664 of Lecture Notes in Computer Science, pages 417–431. Springer, 2011.

[14] M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. Property specification patterns for finite-state verification. In Proc. 2nd workshop on Formal methods in software practice, pages 7–15. ACM, 1998.

[15] L.J. Stockmeyer. The complexity of decision procedures in Automata Theory and Logic. PhD thesis, MIT, 1974. Project MAC Technical Report TR-133.

[16] A. R. Meyer. Weak monadic second order theory of successor is not elementary recursive. In Proc. Logic Colloquium, volume 453 of Lecture Notes in Mathematics, pages 132–154. Springer, 1975.

[17] J.Y. Halpern and J.H. Reif. The propositional dynamic logic of deterministic, wellstructured programs. Theor. Comput. Sci., 27:127–165, 1983.

[18] JY. Halpern and J.H. Reif. The propositional dynamic logic of deterministic, wellstructured programs (extended abstract). In Proc. 22nd IEEE Symp. on Foundations of Computer Science, pages 322–334, 1981.

[19] A.P. Sistla. Theoretical issues in the design of distributed and concurrent systems. PhD thesis, Harward University, 1983.

[20] A.P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logics. In Proc. 14th Annual ACM Symposium on Theory of Computing, pages 159–168, 1982.

[21] A.P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logic. Journal of the ACM, 32:733–749, 1985.

[22] P. Wolper. Temporal logic can be more expressive. In Proc. 22nd IEEE Symp. On Foundations of Computer Science, pages 340–348, 1981.

[23] P. Wolper. Temporal logic can be more expressive. Information and Control, 56(1–

2):72–99, 1983.

[24] M.Y. Vardi and P. Wolper. Reasoning about infinite computations. Information and Computation, 115(1):1–37, 1994.

[25] P.Wolper, M.Y. Vardi, and A.P. Sistla. Reasoning about infinite computation paths. In Proc. 24th IEEE Symp. on Foundations of Computer Science, pages 185–194, 1983.

附录

公式 a 与公式 b 冲突判定规则

1) 若 a 或 b 中至少有一个 为 False，则 a 和 b 冲突；
2) 若 a == ¬b，则 a 和 b 冲突；
3) 若 a 和 b 同时为 Next 算子或者 Release 算子，且 a 和 b 的右子公式冲突，则 a 和 b 冲突；
4) 若 a 为 Next 算子，b 为 Until 算子，且 a 的左子公式和 b 的右子公式冲突，则 a 和 b 冲突；
5) 若 a 和 b 同时为 Until 算子，且 a 的左子公式和 b 的右子公式冲突，则 a 和 b 冲突；
6) 若 a 和 b 中有一个算子为 Or，另一个算子为 Next、Until 或者 Or，且 a 的左子公式和 b 的右子公式冲突，则 a 和 b 冲突；
7) 若 a 和 b 中有一个算子为 Release，另一个算子不为 Release，且算子为 Release 的那个公式的右子公式与另一个公式冲突，则 a 和 b 冲突；
8) 若 a 和 b 中有一个算子为 And，另一个算子不为 Release，且不为 And 算子的公式与另一个公式的左子公式或右子公式冲突，则 a 和 b 冲突；