基于重写技术的
嵌入式系统建模与验证

（申请清华大学工学博士学位论文）

培 养 单 位：计 算 机 科 学 与 技 术 系

学　　科：计 算 机 科 学 与 技 术

研　究　生：刘　嘉　祥

指 导 教 师：顾　　明　教　授

二〇一七年四月

Rewriting-Based Modeling and Verification of Embedded Systems

Dissertation Submitted to

Tsinghua University

in partial fulfillment of the requirement

for the degree of

Doctor of Philosophy

in

Computer Science and Technology

by

Liu Jiaxiang

Dissertation Supervisor : Professor Gu Ming

April, 2017

关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定，即：

清华大学拥有在著作权法规定范围内学位论文的使用权，其中包括：（1）已获学位的研究生必须按学校规定提交学位论文，学校可以采用影印、缩印或其他复制手段保存研究生上交的学位论文；（2）为教学和科研目的，学校可以将公开的学位论文作为资料在图书馆、资料室等场所供校内师生阅读，或在校园网上供校内师生浏览部分内容；（3）根据《中华人民共和国学位条例暂行实施办法》，向国家图书馆报送可以公开的学位论文。

本人保证遵守上述规定。

（保密的论文在解密后应遵守此规定）

作者签名：_____　　导师签名：_____

日　　期：_____　　日　　期：_____

摘 要

随着嵌入式系统与现代社会生产、生活越来越深度的结合，其可靠性和安全性也变得与人们的生命财产安全息息相关，利用形式化方法对嵌入式系统进行验证以保证其正确性的需求日益迫切。形式化模型作为形式化验证方法的核心，是影响它在嵌入式系统中进行应用的关键因素。现代嵌入式系统中多线程技术的应用以及系统与外界环境复杂的交互，对形式化模型的建模能力和验证能力都提出了更高的要求。

重写模型适用于对多线程行为进行建模，且支持模型检测、定理证明等多种形式化验证技术，近年来受到形式化验证领域的关注。将重写模型应用于嵌入式系统的实际验证项目中，目前面临两个问题：(1) 如何在行为具有不确定性的重写模型中描述嵌入式软件的顺序行为；(2) 如何提高易用性，降低重写模型的建模成本。围绕这两个问题，本文提出了一个形式化模型——规范化条件重写模型，且基于该模型设计开发了一套针对嵌入式系统的建模方法以及一个针对 C 语言程序的终止性验证工具 Ceagle-Term：

1. 针对重写模型对顺序行为表达能力的局限性，本文提出能够支持确定性行为描述的规范化条件重写模型。规范化条件重写模型支持自定义数据类型，具备描述状态等价、条件控制的表达能力，也能对以硬件并发行为为代表的不确定性行为、以及以软件顺序行为为代表的确定性行为进行描述和语义区分。
2. 针对模型易用性问题，以建模方法论为切入点，本文基于规范化条件重写模型，提出一套对嵌入式系统结构层次性、行为异构性、结构动态性和实时性等特征的建模方法，旨在对建模过程进行指导。基于语义映射的方式，本文对该建模框架予以实现，并通过对两个真实嵌入式系统进行应用，验证了该方法在嵌入式系统中实际应用的可行性。
3. 针对模型易用性问题，以嵌入式系统软件的自动建模为另一切入点，本文开发了一套基于规范化条件重写模型的 C 程序终止性自动验证工具 Ceagle-Term，为系统的完全正确性提供了必要的工具支持。

作为本文的应用案例之一，机车优化控制系统目前运行稳定，并在沈阳铁路局通过了实车运用考核；而经过本文建模验证的速率单调调度系统目前在某工业级航天控制器中在线运行。

关键词：形式化方法；嵌入式系统；规范化条件重写模型；建模；验证

Abstract

Embedded systems are increasingly used in the modern society, becoming embedded in our lives. Due to the importance of their reliability and safety, formal methods for verifying the correctness of embedded systems are in great demand. As the heart of formal verification methods, formal models are the key to their application on embedded systems. Multithreading techniques are deployed in the modern systems, and there exists complicated interaction between embedded systems and their environment. Both will raise new challenges to the formal models about their abilities to model and verify.

Recently, rewrite systems have won attention for formal verification, given the facts that they are suitable for modeling the behaviors of multithreads and they support multiple verification techniques such as model checking and theorem proving. To apply rewrite systems on realistic verification projects for embedded systems, we need to answer two questions: (1) How do we capture sequential behaviors via rewrite systems inheriting non-determinism? (2) How do we enhance the usability of rewrite systems to ease the modeling process? To answer these questions, this dissertation introduces a new formal model named *normalization conditional rewrite systems*. Based on the new model, we develop a methodology for modeling embedded systems and a termination prover CeagleTerm for C programs:

1. Aimed at the limitation of rewrite systems for expressing sequential behaviors, this dissertation introduces normalization conditional rewrite systems, which are able to describe deterministic behaviors. Normalization conditional rewrite systems support user-defined data types and the expressivity for state equivalence and conditional branching. They can also model and distinguish the semantics of non-deterministic behaviors, such as those concurrently in hardware, and of deterministic behaviors, such as those sequentially in software.
2. Aimed at the usability of the model, focusing on modeling methodologies, this dissertation proposes a methodology which captures the hierarchy, the heterogeneous behaviors, the dynamic structures and the real-timeness of embedded systems. We implement the methodology based on semantics mapping. Then we demonstrate its feasibility for application, by applying it on two realistic embedded systems.
3. Aimed at the usability of the model, focusing on the automatic construction of

models of embedded software, this dissertation develops an automatic termination prover Ceagle-TERM for C programs based on normalization conditional rewrite systems. The tool provides the possibility to prove the total correctness of systems.

As one of the case studies, the Train Optimized Control System is currently in smooth operation, and passes the on-board test by Shenyang Railway Administration; the formally verified Rate-Monotonic Scheduling System by this dissertation has been serving in some industrial avionic control system.

Key words: formal methods; embedded systems; normalization conditional rewrite systems; modeling; verification

目 录

主要符号对照表

FSM	有限状态机 (Finite State Machine)
HCFSM	层次化并发有限状态机 (Hierarchical Concurrent FSM)
CPN	着色 Petri 网 (Colored Petri Net)
LTL	线性时序逻辑 (Linear Temporal Logic)
CIC	归纳构造演算 (Calculus of Inductive Constructions)
HOL	高阶逻辑 (Higher-Order Logic)
RTL	寄存器转换语言 (Register Transfer Language)
$\mathcal{F}$	词汇表
$\mathcal{X}$	可数的变量符号集合
$\mathcal{T}(\mathcal{F},\mathcal{X})$	项表达式集合
$\mathcal{R}$	重写模型
$\mathcal{E}$	等价模型
$\mathcal{R}_{\mathcal{E}}$	模重写模型
$\mathcal{R}_{\mathcal{SE}}$	规范化条件重写模型
$\mathcal{R}^{\mathcal{L}}$	重写逻辑模型
RMS	速率单调调度 (Rate-Monotonic Scheduling)
PC	程序计数器 (Program Counter)
CTL	分支时态逻辑 (Computation Tree Logic)
$\mathcal{R}_{\mathcal{I}}$	整数重写模型
TPDB	终止性问题数据库 (Termination Problems Data Base)
DSL	领域特定语言 (Domain Specific Language)

第 1 章　绪论

1.1　研究背景

嵌入式系统[1] 最早出现在六十年代，其产生是为了减少搭载系统的体积重量以及降低成本。作为计算资源的成本高昂的计算机逐步被成本相对低廉的微处理器系统替换，导致设备不仅变得小型化、低价格，同时处理能力以及功能也不断提高。这些变化使得七八十年代时期，民用电子、消费类电子等行业得到了快速发展并大规模兴起。从工业 2.0 时代开始的自动化、电力驱动的大规模工业生产中，早期的嵌入式系统开始大量投入使用。到现在的工业 3.0 时代以及 2013 年德国提出的工业 4.0，以信息化、智能化、自动化、网络化作为核心，数字化产品以及产品生产制造过程，其全生命周期都将依赖于电子信息而产生的新工业模式。在这中间，嵌入式系统的不断强大将为工业的进程提供技术支撑，以加快工业 4.0 在全领域的普及。

最早，不同的嵌入式系统承载不同功能，其设计是为特定系统、特定任务而定制的，大多数嵌入式系统都是功能单一的系统。在发展过程中，嵌入式系统的处理单元逐渐变得强大（如现在的 ARM、PowerPC、MIPS 等），外围设备、硬件资源越来越丰富，使其可以承担更多复杂的处理任务，并具有了通用性、架构可移植的特点。在嵌入式系统硬件能力提升的基础上，除了为硬件系统开发固件外，Linux、MS-DOS，以及专为嵌入式系统开发的实时操作系统 VxWorks 等操作系统的加入，可以完成系统设备的协调调度、资源监管等任务，并且使得开发运行针对应用需求的用户控制界面和具有更多功能的应用软件成为可能。

除消费类电子产品外，嵌入式系统在民用、军用大型设备中的作用同样重要。在航空领域中，民用航空飞机的机载电子设备功能越发强大，系统架构也越发复杂。飞行控制系统、导航系统、通信系统、雷达系统、环境控制系统，以及各终端传感器系统都搭载了嵌入式系统来对其进行控制，并完成计算、存储、传输等以支持飞控、通信、导航等高级任务。其中系统异步、并发、调度、实时等问题引起的不确定性都是影响系统正常工作、影响飞行安全的重要因素。安全性是民用飞机的重要属性，系统安全性的设计、验证以及管理贯穿飞行器整个生命周期。有 ARP4754[2]、ARP4761[3] 来指导飞行器系统的安全性设计，以及 DO-178C、DO-254 来指导符合安全性等级的软件、硬件设计。其中嵌入式系统的硬件要满足安全性设计中对应的可靠性需求，而固件及软件系统则要满足对应的软件验证需求。如飞控系统的安全性等级为 A 级，其涉及的嵌入式系统硬件设备要满足安全性保障

等级为 A 的设计、测试、验证流程；对于其系统软件，则要求其设计及验证过程中必须使用严格的形式化方法对其安全等级进行保障。其它安全攸关的系统，如轨道运输、海洋运输都有相应的安全性保障需求。嵌入式系统硬件设计验证方法早已比较成熟，但是软件的形式化设计验证方法相对于硬件起步较晚。随着嵌入式系统软件发展的加快，针对嵌入式系统软件的形式化设计与验证存在迫切需求。在嵌入式系统的设计与开发过程中，硬件与软件往往是难以分开单独设计的，因为其软件及固件开发的专用性，互相影响程度很深。所以一种基于形式化方法的、可以同时对硬件及软件系统进行验证的方法亟待研究。

形式化建模与验证方法的基本流程如图 1.1 所示。建模人员需要先对目标系统进行手工或自动建模，得到系统的行为模型。针对系统的行为模型，验证人员再根据其关心的系统属性，利用相应的验证技术，如模型检测[4]（model checking）、定理证明[5]（theorem proving）等，及其相应的工具对系统模型进行形式化验证，从而得到针对属性的验证结果。根据验证结果，开发人员可以对目标系统进行修改或调整。在整个流程中，形式化的系统模型作为连接建模过程和验证过程的桥梁，对建模的难易程度、精确程度及验证的效果都起着至关重要的作用，是整个形式化建模验证方法的基础。

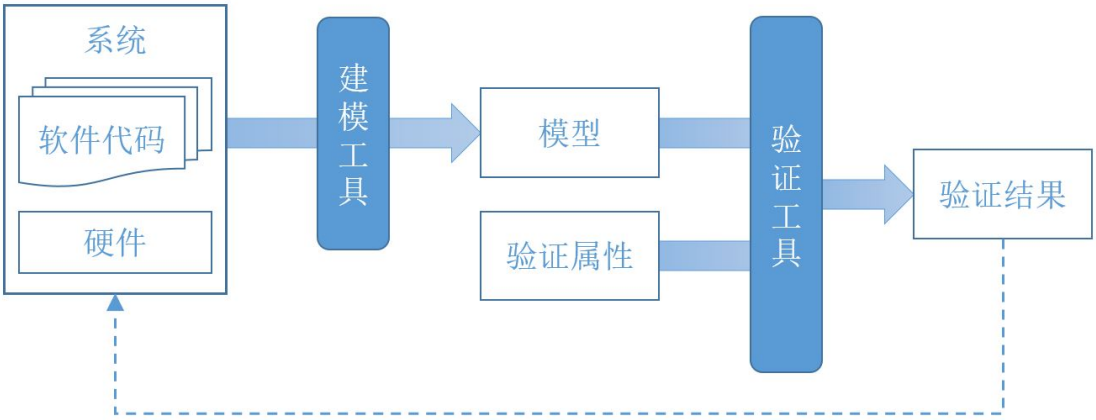


图 1.1　形式化建模与验证基本流程

在种类繁多的嵌入式系统中，存在一类系统，它们与环境存在频繁的交互：它们通过输入设备（如传感器等）从环境接收周期性或非周期性的输入，系统对这些输入进行处理，然后通过输出设备（如执行器等）将处理结果反馈到环境中。这类系统被称作反应式系统[6,7]（reactive system）。大部分嵌入式系统都是反应式的：小到我们身边的消费类电子产品（如智能穿戴设备），大到交通运输系统（如第 3.5 小节讨论的机车控制系统）。由于反应式系统的输入不是单一的数值或事件，而是

一个可能无穷的输入序列，这一显著特征从两个方面对形式化建模验证所使用的形式化模型提出了要求。

1. 表达能力，即建模能力。随着现代嵌入式系统处理单元计算能力的提高，在硬件条件允许的情况下，多线程技术越来越多地被应用到反应式系统中[8–10]。这一方面给系统引入了更多的并发性，要求形式化模型可以对并发行为进行描述；另一方面也要求形式化模型可以对线程的创建、释放等行为进行描述。出于对可组合性和层次化的考虑，线程一般被看作系统中的组件进行建模，于是线程的动态创建与释放行为要求形式化模型具有对系统结构动态变化的描述能力。
2. 验证能力。无穷的输入序列以及多线程带来的并发性使传统的模型检测技术可能面临状态空间爆炸的问题[11]。因此，对反应式系统的形式化验证要求所使用的模型支持能解决状态空间爆炸的形式化验证技术，如定理证明。

在下一小节中，我们将从上述两个方面对主流建模方法及形式化模型进行介绍和对比。

1.2　研究现状

嵌入式系统的建模方法不胜枚举，根据其描述对象的不同，可以建立架构模型、状态模型、数据模型等。它们分别以不同角度去描述嵌入式系统，用仿真、测试、形式化验证等方法保证系统的正确性、有效性、可靠性等多方面重要属性，特别是对安全攸关系统的设计、验证与运行提供有力支持。如上一节提到，嵌入式系统在众多安全攸关系统（如航空航天飞行器、铁路、核电站、医疗、保密系统等）中扮演着重要的角色。自 1990 年起，已有多份统计以研究报告与论文的形式指出形式化方法在工业领域中所取得的突出成果。2009 年英国约克大学的 Jim Woodcock 统计得到[12]：在工业应用中，对比只依赖于测试验证的工程开发，基于形式化建模与验证的工程项目有 92% 得到了产品品质的提升，体现在更少的故障、正确性的提升、设计能力提升等多方面。

在这些形式化方法中，有些用抽象的形式化规约（specification）避免模型中的二义性，从而保证设计质量，如 VDM[13]、Z[14] 等。有些用状态模型描述系统行为特性，模型本身能够采用静态分析与模型执行来进行验证，如 Petri 网[15]、自动机[16] 等。

有限状态机[17]（FSM）是面向系统状态最基本的模型，被控制系统广泛地应用。但对于功能、架构复杂的嵌入式系统，有限状态机缺乏对其并发性、实时性等方面的支持。所以有限状态机有多种扩展形式：支持并发性的层次化并发有限状

态机[18]（Hierarchical Concurrent FSM，HCFSM）以及支持时间的时间自动机[19,20]（timed automata）。HCFSM 可将一个状态组视为一个状态，状态组之间通过全局变量进行通信，可以用于描述多组件的、并发的控制系统。时间自动机作为有限状态机的另一个变种，增加对状态转移过程中时间因素的描述，变迁上标记的是该状态转移发生的时间约束，从而可以满足对实时系统时间分析的需求。然而，自动机及其扩展模型的模型结构是静态的，因此无法对系统结构的动态变化进行表达。

Uppaal2k[21]，作为 Uppaal[22] 的后继工具，提供了对时间系统的建模、仿真以及验证功能，适用于对具有共享变量、通道通信的系统进行描述。Uppaal2k 支持对时间自动机图形化的建模、动态仿真以及模型检测等功能，可以进行有界活性检测、死锁检测、验证使用 TCTL[23]（Timed Computation Tree Logic）表达式描述的多种性质，不支持定理证明技术。

Petri 网[15] 的研究与应用相当广泛。由于对异步并发系统描述上的优势，Petri 网在实时系统、协议验证、硬件设计、制造过程、商业管理等众多领域中均发挥了作用[24–27]。在实时嵌入式系统的应用中，Petri 网为描述待验证系统的异步、并发性、不确定性等多种特性提供了有效工具。虽然 Petri 网的网络结构是静态的，即它的三要素——库所、变迁以及有向弧不会发生改变，但由于 Petri 网的令牌（token）数量动态可变，且 Petri 网的状态由令牌分布决定，因此它可用于描述对象系统的结构变化。为了扩展标准 Petri 网的适应性，Petri 网衍生出多种变种，如着色 Petri 网[28]（Colored Petri Net，CPN）、对象 Petri 网[29]（object Petri net）、混合 Petri 网[30]（hybrid Petri net），以及多种时间相关的 Petri 网，如 Time Petri Net[31]（TPN）、Timed Petri Net[32]（TdPN）、Timing Constraint Petri Net[33]（TCPN），以适用于不同的应用需求。

在工具方面，一共有多达 40 多种工具提供了对时间相关的 Petri 网的支持。TINA[34]（TIme petri Net Analyzer）是其中应用最广泛的一种，可以对 TPN 和 TdPN 进行线性时序逻辑[35]（Linear Temporal Logic，LTL）性质验证、可达性验证等。Romeo[36] 是支持 TPN 验证的工具，并且预定义了多种从 TPN 到时间自动机的转换方法。这些工具基本都支持 Petri 网的建模、仿真及多种验证技术。

对于前面提到的时间自动机以及几种时间 Petri 网，由于它们的语法、以及其工具的局限性，不支持对系统中复杂的数据类型进行建模。由于语言表达能力不足，使得这些建模方法在应用于涉及复杂数据结构（如结构式数据类型或自定义数据类型）的系统时，模型的描述能力具有局限性。

在 Petri 网的众多变种中，着色 *Petri* 网可以对自身的令牌对象进行具体的特征描述，即着色。并且它支持利用面向表达式的函数式程序设计语言对系统进行

描述，可以支持如布尔、整数、列表、字符、结构体等多种数据类型及自定义数据类型，丰富了模型的表达能力。通过 CPN Tools[37] 可以对使用 CPN 的项目进行建模、仿真，以及基于模型检测的形式化验证，同时 CPN Tools 具有对函数式程序设计语言标准 ML[38]（SML）的支持。有了以上特点，CPN 可以用来对具有复杂对象的系统进行建模及验证。

类型论[39]（type theory）是数学、逻辑学和计算机科学的一个理论分支，它是大多数定理证明工具的理论基础。与自动机和 Petri 网不同，当把类型论看作一个形式模型用于建模与验证时，系统的状态一般用代数表达式进行基于语法的建模，而系统的行为则被建模成函数演算或逻辑关系。其中在建模验证领域应用较为广泛的类型论有*归纳构造演算*[40]（Calculus of Inductive Constructions，CIC）和*高阶逻辑*[41]（Higher-Order Logic，HOL）。由于类型论的表达式类型是利用归纳方法进行定义的，因此基于类型论的形式化模型可以支持用户自定义类型，也可以通过表达式及类型的定义来描述目标系统的动态结构变化。

基于归纳构造演算的 Coq[42] 以及基于高阶逻辑的 Isabelle[43] 是基于类型论的应用较为广泛的建模验证工具，其验证技术是定理证明。由于其底层逻辑的不可判定性，虽然 Coq 和 Isabelle 提供了大量判定过程（decision procedure）与半判定过程（semi-decision procedure）用于辅助证明过程，但其定理证明方法主要还是需要人工参与的*交互式定理证明*。另一方面，由于非确定性的系统行为需要利用逻辑关系进行建模，因此包含不确定性的系统模型无法进行仿真。基于类型论的建模验证方法及工具主要应用于数学定理证明[44,45]，近年来开始被应用于系统验证领域。由于其验证过程需要耗费较高的人力成本，且工具学习难度较大，因此主要应用于安全攸关系统的验证工作中，如编译器验证（CompCert 项目[46,47]）、操作系统验证（seL4 项目[48] 和 CertiKOS 项目[49,50]）等，目前针对嵌入式系统的应用较少。

另一种与类型论有关的形式化模型是*重写模型*[51,52]（rewrite system），它是基于代数表达式和规则的一种不确定性模型。与类型论类似，基于重写模型进行建模时，系统状态一般由表达式进行描述，而系统行为则由有向规则进行表达。因此重写模型也可支持自定义类型以及系统动态结构变化的描述。与类型论不同的是，重写模型本身是可执行的，即它支持模型仿真。重写模型的执行过程本身包含并发的特性。通过对重写模型的规则进行扩展，其衍生模型如*模重写模型*[53]（rewrite system modulo）及*条件重写模型*[54]（conditional rewrite system）等，可以支持系统中状态等价、条件控制等特性的描述。

近年来重写模型开始被应用于系统的建模与验证工作中。Bluespec[55] 是一种

基于重写模型的硬件描述语言，十分适用于 SoCs（System on Chips）的设计验证。Bluespec 使用重写的目的是利用重写模型固有的并发特点来描述并发系统，以便通过仿真来保障其设计正确性。它利用重写规则来描述并发，可以降低设计的复杂性。Bluespec 支持函数式程序设计语言 Haskell[56]，以便对嵌入式系统中的复杂类型、自定义参数等进行定义，增加其描述能力。其工具 Bluespec System Verilog[57] 可以将 Bluespec 写成的代码转换成 RTL（Register Transfer Language）。但工具本身并不支持形式化验证，需要通过外部工具进行[58]。重写逻辑[59,60]（rewriting logic）也是重写模型的一种扩展，它通过对重写规则以及规则应用策略的扩展，增加其对复杂数据结构及复杂系统结构的描述能力。基于重写逻辑的 Maude 语言及工具[61,62]，可支持利用重写逻辑对系统进行建模，并支持模型仿真、可达性验证、LTL 性质验证等功能。更重要的是，Maude 同时还提供了针对重写逻辑模型的交互式定理证明工具[63]。目前基于重写逻辑的建模验证工作主要针对协议验证及语言验证，还没有被广泛应用于嵌入式系统领域[60,64]。

表 1.1　形式化模型对比

形式化模型	自动机①	Petri 网②	类型论③	重写模型④
并发行为	✓	✓	✓	✓
自定义类型	☑	✓	✓	✓
动态结构建模	✗	✓	✓	✓
模型仿真	✓	✓	✗	✓
模型检测	✓	✓	✗	✓
定理证明	✗	✗	✓	✓

注：“✓”表示“支持”；“☑”表示“支持有限”，“✗”表示“不支持”。

① 代表模型：FSM、HCFSM、时间自动机；代表工具：SPIN、Uppaal2k
② 代表模型：CPN、TPN；代表工具：TINA、Romeo、CPN Tools
③ 代表模型：CIC、HOL；代表工具：Coq、Isabelle
④ 代表模型：模重写模型、条件重写模型、重写逻辑；代表工具：Bluespec、Maude

总结以上四种形式化模型在表达能力及验证能力方面的特性，如表 1.1 所示。从该表可以看出，重写模型较符合反应式系统对形式化模型提出的要求。如果我们只考虑表达能力及状态空间爆炸问题，基于类型论的模型也可作为候选模型。但从验证成本的角度考虑，定理证明技术所耗费的人力成本较高，且只能验证“给定属性成立”：若验证人员经过大量尝试后无法成功证明某属性成立，此结果并不能说明该属性不成立。因此，在进行高成本的定理证明方法前，先使用自动化的模型仿真或模型检测技术对目标属性进行“反例”搜索，可以有效降低伪命题带来的无谓的定理证明人力成本。综合以上考量，本文选择重写模型作为对反应式

嵌入式系统建模验证的形式化模型进行研究。

1.3　研究思路

重写模型的良好特性虽然满足反应式系统的并发性及复杂性对形式化模型提出的要求，但若要将重写模型应用于对嵌入式系统的实际建模与验证工作中，目前的重写模型及已有扩展仍存在以下问题。

1. 重写模型对软件的顺序行为描述具有局限性。由于不确定性与并发性是重写模型的固有属性，因此重写模型及其扩展被广泛应用于硬件描述与协议验证的场景中。但由于嵌入式系统是硬件与软件共存的整体，软件部分含有大量顺序执行的代码。针对这些顺序行为的其中一种建模方式，是利用具有不确定性的规则对其进行编码。这种方案可能导致两个互相独立的系统线程之间产生大量与验证属性无关的非必要的交织（interleaving）行为，使模型的可达状态数量呈指数级增长。另外一种解决方案，是利用单条规则对大段的顺序代码进行抽象。这对建模人员的能力提出了更高的要求，且经过抽象的模型增加了理解模型的成本。因此，对重写模型进行扩展，使其在模型层面对顺序行为进行支持，对重写模型在嵌入式系统的建模应用具有重要意义。
2. 重写模型易用性低，建模成本较高。与自动机、Petri 网等可被图形化的形式化模型不同，重写模型是代数的（algebraic）模型。由于自动机被广泛用于系统规约的描述，且其图形化表示方式形象直观，因此对开发人员来说学习成本较低。而重写模型作为一种底层模型，不为开发人员所熟知，且其代数化的抽象表示方法，也给建模人员带来了额外的学习成本与理解成本。因此，提高重写模型的易用性，降低建模人员的使用成本，对重写模型在嵌入式系统的实际应用具有重要意义。

为推动形式化方法在嵌入式系统可靠性、安全性保障方面的应用，满足现代反应式系统给形式化建模验证方法提出的需求，本文将利用重写模型作为基本的形式化模型，针对上述问题从理论模型的角度及建模方法的角度，提高重写技术在嵌入式系统建模与验证中的实际应用能力。本文拟从以下三个角度进行研究：

1. 以嵌入式系统软件的顺序行为作为切入点，设计能够支持确定性行为的重写模型扩展——规范化条件重写模型。规范化重写模型对确定性行为的支持拟参考 Nipkow 对高阶重写 β 规则及 η 规则的规范化过程[65] 进行设计。通过加入模重写模型的等式规则、条件重写模型的条件规则，使规范化条件重写具备描述状态等价、条件控制的表达能力。本文将先对重写模型的规则进行扩展，给出规范化条件重写模型的形式化语法定义；再对重写模型的规则应

用策略进行扩展，给出规范化条件重写模型的形式化语义定义。

2. 针对重写模型易用性低的问题，以嵌入式系统建模方法论为切入点进行研究。针对嵌入式系统的特性，如多层次结构、高度并发、硬件的并发行为与软件的顺序行为并存、系统结构动态变化、实时性等，提出一套基于上述规范化条件重写模型的建模方法。基于语义映射的方式，本文拟将部分规范化条件重写模型映射为重写逻辑模型，从而将该建模方法在工具集 Maude 中进行实现。最后通过将该建模方法应用于两个真实的嵌入式系统案例，从而验证该建模方法在对嵌入式系统进行实际应用的可行性。
3. 针对重写模型易用性低的问题，以嵌入式系统软件——C 语言程序的自动建模作为切入点进行研究。针对程序终止性这一特定属性，开发一套基于规范化条件重写模型的 C 语言自动验证工具 Ceagle-Term。该工具接受 C 语言程序输入，通过对输入程序进行语义分析，自动建立其程序行为对应的规范化条件重写模型。最后利用重写领域已有的终止性求解工具，对生成的规范化条件重写模型进行求解，从而得到输入 C 程序的终止性验证结果。

1.4　论文贡献

本文针对嵌入式系统的形式化建模与验证方法及其在实际系统中的应用，面向反应式系统具有的结构动态性及软硬件行为并存的异构性引发的问题进行研究。本文提出了具有针对性的形式化模型，并设计、实现了相关的建模方法及建模验证工具。本文具体贡献如下：

1. 提出了能描述系统结构动态性、且能在模型层面区分并发行为与顺序行为的形式化模型——规范化条件重写模型。与其它形式化模型相比，作为一种扩展的重写模型，规范化重写模型通过模型的代数性质和等价语义扩展，实现了对系统结构动态变化进行描述的能力。作为形式化验证过程的基础，规范化重写模型可以支持模型仿真、模型检测、定理证明等多种验证技术。与其它重写模型扩展相比，规范化重写模型能够对以硬件并发行为为代表的不确定性行为、以及以软件顺序行为为代表的确定性行为进行语义上的区分，使系统的行为模型具有更精确的语义，也给建模过程带来了便利，降低了验证过程可能出现的不必要的状态空间爆炸的风险。基于规范化条件重写模型，提出了对嵌入式系统的层次结构、动态结构变化等特征的具体建模方法并予以实现，降低了基于该模型进行建模的学习成本。该模型和建模方法在两个真实的嵌入式系统建模与验证案例中得以应用。
2. 设计开发了一套针对 C 语言程序的终止性验证工具 Ceagle-Term。该工具接

受 C 语言程序输入，其建模与验证过程自动进行，不需人工参与，有效降低了形式化方法应用于嵌入式系统建模与验证的成本，为系统的完全正确性提供了必要的工具支持。

1.5　论文结构

本文的组织结构如下：第 2 章先介绍重写模型的背景知识，然后对本文提出的规范化条件重写模型进行严格的形式化描述，包括它的语法及语义定义；第 3 章从模型应用的角度，重点阐述了基于规范化条件重写模型对嵌入式系统进行建模的方法，并对两个真实的应用案例进行了介绍；第 4 章描述了针对终止性对 C 语言程序进行自动建模验证的方法，并介绍了本文开发的相关工具 Ceagle-Term；第 5 章对本文的工作进行总结，并对未来的改进方向进行展望。

第 2 章　规范化条件重写模型

嵌入式系统的日益复杂，要求用于对其建模的形式化模型具有丰富的表达能力。重写模型作为一种基于语法、基于规则的计算模型，其表达能力具有很强的可塑性，目前已衍生出多种扩展形式。这些重写模型的扩展包括对等价关系、条件判断等行为的表达能力提升。在实际系统中，存在某类行为，其自身的运行结果是确定的，不会因系统中其它组件的交互而受到影响，也不会影响其它组件的运行结果，例如嵌入式软件中只涉及局部变量的顺序行为。我们称其为*确定性行为*。从验证的角度来说，系统状态空间的大小主要由*非确定性行为*（又称“*不确定性行为*”）决定，因此在模型中对确定性行为与非确定性行为进行区分，对于模型验证具有重要意义。目前已有的重写模型扩展，都没有针对这一问题给出解决方案。本章提出的规范化条件重写模型，是针对这一问题试图给出合理解决方案的一种新的重写模型扩展形式。

2.1　引言

重写（rewriting）是一种基于规则的、非确定性的（non-deterministic）抽象计算技术，它利用内涵（intensional）的方式来定义计算。重写模型[51,52]则可看作用于描述计算的计算模型。一个重写模型包含若干条形如 $l \to r$ 的规则。对象表达式根据这些规则进行语法上的变换，从而产生新的对象表达式，这个过程形式化地定义了“计算”的语义。

重写概念的提出，一开始是为了解决等式判断的问题。例如给定两个表达式 $2+2$ 和 $4+0$，如何判断它们是否相等。这需要一种技术将两个表达式各自计算成为最简形式（“范式”），然后判断它们的范式是否相等，从而得到原表达式是否相等的结论。重写作为一种抽象技术，被越来越多学者深入研究，逐渐形成其理论体系。目前重写已经成为代数规范语言[66]（algebraic specification language）、函数式编程语言[67]（functional language）的核心。在基于类型论的定理证明器（如 Coq[42]）中，重写技术对逻辑规约和类型推导也起到了不可或缺的关键作用[68]。

重写模型作为一种抽象的底层模型，与自动机类似，其本身的语义是非常简单的。若要利用重写模型对复杂的计算甚至是系统行为进行描述与建模，则需要对重写模型的语法及语义进行扩展。这需要面临两方面的挑战：

1. 规则的扩展，即语法扩展。标准的重写模型包含一个规则集合，这些规则描述了计算的方法。规则之间处于对等的地位，不存在种类的差别。然而在实

际的软硬件系统中，行为具有多样性，比如同步行为、异步行为、确定性行为、不确定性行为等。如果不在模型的层次上对各种行为加以区分，则会对建模人员带来诸多不便。因此需要对标准重写模型的规则定义进行扩展，以便于描述系统中丰富的行为种类。

2. 规则应用策略的扩展，即语义扩展。重写规则只从形式上定义了重写模型的构成，而规则应用策略规定了重写规则的应用方式，即语义。在对标准重写模型的规则进行扩展后，如何定义不同种类规则的应用策略，才能使其描述的计算更加贴近实际系统中的行为，是必须考虑的问题。

本章基于现有的多种重写模型的扩展，提出了规范化条件重写模型（normalization conditional rewrite system），它在模型层次支持等价规则、条件规则、确定性规则等不同种类的规则。我们还定义了规范化条件重写的规则应用策略，描述了确定性计算与非确定性计算的一种协作方式。

本章余下部分组织结构如下：首先在第 2.2 小节简要介绍相关的重写模型扩展工作；在第 2.3 小节，我们给出重写模型的形式化定义；紧接着在第 2.4 小节，针对涉及等价关系的计算以及带条件的计算，我们给出重写模型的两种经典扩展形式；在第 2.5 小节，针对确定性与非确定性计算，我们提出新的规范化条件重写模型；最后对本章进行简要小结。

2.2　相关工作

重写模型作为一种计算模型，主要包括两个部分——操作对象及操作规则。对重写模型表达能力的扩展主要也包括两个方向。

一个方向是对操作对象进行扩展。最初的重写模型，其操作对象是抽象的无结构的对象。根据应用的需求，逐渐产生了针对字符串的串重写模型[69]（string rewrite system）、针对项表达式的项重写模型[52]（term rewrite system）、针对图的图重写模型[70]（graph rewrite system）等各种各样的重写模型。由于各种对象所具有的结构性质不甚相同，因此各种重写模型的扩展都具有自身表达能力的优劣。由于本文的目的是对嵌入式系统进行建模，因此本文主要讨论较适合于系统建模的项重写模型（以下简称“重写模型”）。

另一个方向是对规则及其应用策略进行扩展。经典重写模型的操作规则是有向的，但如果规则两边的表达式之间不存在主次关系，则有向的规则将变得毫无意义。这时候我们需要引入等价关系。模重写模型[53] 与类重写模型[71]（class rewrite system）就是为了描述等价关系而产生的重写模型的扩展，两者的主要区别在于规则应用策略的不同。范式重写模型[72]（normal rewrite system）和重写逻辑[59,60] 分

别基于模重写模型和类重写模型，通过增加有向规则对等价关系进行描述，进一步提高了模型对等价关系的表达能力。另一方面，条件重写模型[54] 在规则中增加了条件约束，使其能描述重写规则应用时应满足的约束条件，从而对真实系统中的控制流具备了更灵活的支持。然而，针对确定性和非确定性行为，目前还没有很好的解决方案能对其进行区分描述。

本文提出的规范化条件重写模型及对应的重写策略，目的在于在模型层面区分确定性行为与非确定性行为，使建模人员在应用该模型对系统进行建模分析时更加方便、更加具有针对性，也为验证工具对两类行为的分析奠定了基础。

2.3　重写模型

作为一种计算模型，从语法的角度看，重写模型包含两个部分——操作对象与操作规则。本文讨论的重写模型，其操作对象称为项表达式（term），操作规则称为重写规则（rewrite rule）。

2.3.1　项表达式

项表达式是重写模型讨论、操作的对象，它是一个抽象的、非解释性的代数表达式。

词汇表（signature）是一个函数符号（function symbol）集合，记作 $\mathcal{F}$。其中的每一个函数符号 $f \in \mathcal{F}$ 都具有固定的元数 n（$n \geq 0$），即该函数符号所需参数的数量。元数为 n 的函数符号 f 记作 $f^{(n)}$；当 n 可从上下文推断确定时，$f^{(n)}$ 可简写成 f。元数为 0 的函数符号称为常数。

当给定一个词汇表和一个变量（符号）集合时，我们可以采用如下归纳方式来定义一个项表达式的集合。

定义 2.1 (项表达式)： 假定一个词汇表 $\mathcal{F}$ 和一个可数的变量（符号）集合 $\mathcal{X}$，由 $\mathcal{F}$ 和 $\mathcal{X}$ 构建的项表达式集合 $\mathcal{T}(\mathcal{F},\mathcal{X})$ 由以下语法规则定义：

$$\mathcal{T}(\mathcal{F},\mathcal{X}) \quad ::= \quad x \mid f^{(n)}(t_1, t_2, \ldots, t_n)$$

其中 $x \in \mathcal{X}$，$f^{(n)} \in \mathcal{F}$，$t_1, t_2, \ldots, t_n \in \mathcal{T}(\mathcal{F},\mathcal{X})$。

下面我们举一个用来表示自然数集合的项表达式集合的例子。

例 2.1： 假定 $\mathcal{F} = \{\mathsf{zero}^{(0)}, \mathsf{s}^{(1)}\}$，$\mathcal{X} = \{x, y, z, \ldots\}$，则 zero、s(zero)、s(s(zero))、s(s(x)) 都是属于 $\mathcal{T}(\mathcal{F},\mathcal{X})$ 的项表达式。

需要注意，项表达式只是一个抽象的语法表达式，它本身并不蕴含任何语义信息。但我们可以人为地赋予它某种语义，并且通过定义基于项表达式的规则或等式，将其语义反映出来。在例 2.1 中，函数符号 zero 可以看作是自然数 0，而 s 表示它的参数 x 的后继，即 x 的下一个自然数 $x+1$。于是例 2.1 只利用两个抽象的函数符号 zero 和 s，构造出可以表示全体自然数的项表达式集合 $\mathcal{T}(\mathcal{F},\mathcal{X})$——因为任意一个自然数，要么为 0，即 zero，要么为某个自然数 x 的后继，即 $\mathsf{s}(x)$。这就是 Peano 代数[73] 中对于自然数的表示方法。

另外需要注意，这里所讨论的变量，也是语法意义上的抽象符号。这与我们一般所指的指令式编程语言（如 C 语言）中的变量不一样，不会“储存”具体的“值”。一个带有变量的项表达式，如 $\mathsf{s}(\mathsf{s}(x))$，它可以被看作是一个项表达式，也可以被看作是一个项表达式的模式（pattern），即表示形如 $\mathsf{s}(\mathsf{s}(x))$ 的所有项表达式，比如 $\mathsf{s}(\mathsf{s}(\mathsf{zero}))$、$\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{zero})))$ 以及 $\mathsf{s}(\mathsf{s}(\mathsf{s}(y)))$，但 $\mathsf{s}(\mathsf{zero})$ 不能被 $\mathsf{s}(\mathsf{s}(x))$ 所表示。关于模式及其实例的概念，后文会有准确的定义。

在接下来的讨论及所有例子中，我们假定使用同一个可数的变量集合 $\mathcal{X} = \{x, y, z, \ldots\}$，不再对每个例子进行复述。

为了记号简洁，对于任意一元函数符号 $f^{(1)} \in \mathcal{F}$，我们定义

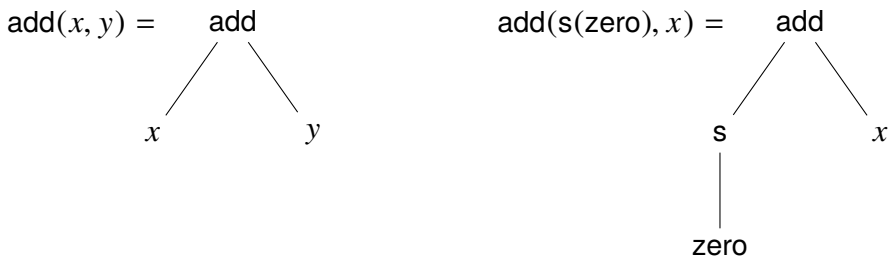
$$\begin{aligned} f^0(t) &\stackrel{\text{def}}{=} t, \\ f^{n+1}(t) &\stackrel{\text{def}}{=} f(f^n(t)), \end{aligned}$$

其中 $n \geq 0$，$t \in \mathcal{T}(\mathcal{F},\mathcal{X})$。

因此在例 2.1 中，$\mathsf{s}(\mathsf{zero})$ 可简写成 $\mathsf{s}^1(\mathsf{zero})$，代表自然数 1；$\mathsf{s}(\mathsf{s}(\mathsf{zero}))$ 可简写成 $\mathsf{s}^2(\mathsf{zero})$，代表自然数 2。

一个项表达式 t 可以看作是一棵带标记的有序树，它的叶子结点被变量或常数标记，它的分支结点被元数不为零的函数符号标记。

例 2.2： 假定 $\mathcal{F} = \{\mathsf{zero}^{(0)}, \mathsf{s}^{(1)}, \mathsf{add}^{(2)}\}$。项表达式 $\mathsf{add}(x, y)$、$\mathsf{add}(\mathsf{s}(\mathsf{zero}), x)$ 可分别用以下两棵树来表示：



假设有序树的每条边都被一个正整数标记，则一个正整数序列可以用来表示该有序树上从根结点开始的一条路径，从而表示该有序树上的一个结点的位置。

定义 2.2 (位置)： 位置是一个由正整数组成的有限序列，表示成 $n_1 \cdot n_2 \cdot \ldots \cdot n_m$，空序列记作 Λ。位置的集合用 $\mathcal{P}$ 表示。

如果标记在有序树某条边上的正整数 n 表示的是该边连接的子结点是该边连接的父结点的第 n 个子结点，那么可以定义一个映射 $subterm : \mathcal{T}(\mathcal{F},\mathcal{X}) \times \mathcal{P} \to \mathcal{T}(\mathcal{F},\mathcal{X}) \cup \{\bot\}$，其中 $\bot \notin \mathcal{T}(\mathcal{F},\mathcal{X})$ 表示一个不合法的项表达式：

$$
\begin{aligned}
subterm(t, \Lambda) &\stackrel{\text{def}}{=} t \\
subterm(x, i \cdot p) &\stackrel{\text{def}}{=} \bot \\
subterm(f^{(0)}, i \cdot p) &\stackrel{\text{def}}{=} \bot \\
subterm(f(t_1, \ldots, t_n), i \cdot p) &\stackrel{\text{def}}{=} subterm(t_i, p) \\
subterm(f(t_1, \ldots, t_n), j \cdot p) &\stackrel{\text{def}}{=} \bot
\end{aligned}
$$

其中 $x \in \mathcal{X}$，$f \in \mathcal{F}$，$t, t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F},\mathcal{X})$，$1 \le i \le n$，$j > n$，$p \in \mathcal{P}$。直观上解释，给定项表达式 t 和位置 p，$subterm(t,p)$ 表示 t 在位置 p 的子项表达式（subterm）。

例 2.3： 在例 2.2 中，

$$
\begin{aligned}
subterm(\mathsf{add}(x, y), \Lambda) &= \mathsf{add}(x, y) \\
subterm(\mathsf{add}(x, y), 1) &= x \\
subterm(\mathsf{add}(\mathsf{s}(\mathsf{zero}), x), 1 \cdot 1) &= \mathsf{zero} \\
subterm(\mathsf{add}(\mathsf{s}(\mathsf{zero}), x), 3) &= \bot
\end{aligned}
$$

基于映射 $subterm$，可以定义以下关于项表达式的概念及符号。给定项表达式 t，$\mathcal{P}os(t) \stackrel{\text{def}}{=} \{p \mid subterm(t,p) \neq \bot\}$ 是 t 的所有位置的集合。给定项表达式 t 及位置 $p \in \mathcal{P}os(t)$，$t|_p = subterm(t,p)$ 是 t 在位置 p 的子项表达式。给定项表达式 t, u 和位置 $p \in \mathcal{P}os(t)$，$t[u]_p$ 表示把 t 里的子项表达式 $t|_p$ 替换为 u 所得到的新的项表达式。

定义 2.3 (代换)： 代换（substitution）是从变量集合到项表达式集合的映射。给定词汇表 $\mathcal{F}$ 和代换 $\sigma : \mathcal{X} \to \mathcal{T}(\mathcal{F},\mathcal{X})$，代换 σ 可以通过以下定义扩展成从项表达式

集合到项表达式集合的映射：

$$\begin{aligned}\sigma(f^{(0)}) &\stackrel{\text{def}}{=} f^{(0)}\\ \sigma(f(t_1,\ldots,t_n)) &\stackrel{\text{def}}{=} f(\sigma(t_1),\ldots,\sigma(t_n))\end{aligned}$$

其中 $f \in \mathcal{F}$，$t_1,\ldots,t_n \in \mathcal{T}(\mathcal{F},\mathcal{X})$。

一个代换 $\sigma : \mathcal{X} \to \mathcal{T}(\mathcal{F},\mathcal{X})$ 的域定义为 $\mathcal{D}om(\sigma) \stackrel{\text{def}}{=} \{x \in \mathcal{X} \mid \sigma(x) \neq x\}$，即域中的变量经过 σ 映射后不等于自身。一个代换 σ 的域如果只有有限个元素 $\{x_1,\ldots,x_n\}$，则可以写作 $\sigma = \{x_1 \mapsto t_1,\ldots,x_n \mapsto t_n\}$，其中对于 $1 \leq i \leq n$ 有 $t_i = \sigma(x_i)$。

例 2.4：在例 2.2 中，给定 $\sigma = \{x \mapsto \mathsf{s(zero)}, y \mapsto \mathsf{zero}\}$，则 $\sigma(\mathsf{add}(x,y)) = \mathsf{add(s(zero),zero)}$，$\sigma(\mathsf{add(s(zero)},x)) = \mathsf{add(s(zero),s(zero))}$。

给定项表达式 s 和 t，如果存在代换 σ 使 $s = \sigma(t)$，则称 s 是 t 的一个实例（instance）。我们称计算 σ 的过程为模式匹配（pattern matching）。

2.3.2　重写规则与重写关系

定义了项表达式作为操作的对象，我们需要定义可用于操作项表达式的工具。如前文已提到，例 2.1 定义了描述自然数集合的项表达式集合。在例 2.2 中，如果将词汇表中的函数符号 $\mathsf{add}^{(2)}$ 看作是加号，则例 2.2 的项表达式集合可以表示所有定义在自然数上的加法表达式。于是直观上看，$\mathsf{add(s(zero),zero)}$ 表示的是加法表达式 $1+0$。加法表达式可被计算，而项表达式也类似。

同时需要注意，add 只是抽象的函数符号。虽然它可以被看作加号，当然它也可以被看作是乘号，使得对应的项表达式表示的是乘法表达式。因此，项表达式的“计算”方式，决定了该函数符号可以被解释的方式，即决定了它的语义。

定义 2.4 (重写规则)：给定词汇表 $\mathcal{F}$，一条重写规则是由项表达式 $l, r \in \mathcal{T}(\mathcal{F},\mathcal{X})$ 组成的有序对（二元组），记作 $l \to r$。其中 l 称作该重写规则的左项，r 称作该重写规则的右项。

定义 2.5 (重写模型[52])：给定词汇表 $\mathcal{F}$，重写模型是一个由若干重写规则组成的集合 $\mathcal{R} = \{l_i \to r_i\}_i$。

例 2.5： 假定 $\mathcal{F} = \{\mathsf{zero}^{(0)}, \mathsf{s}^{(1)}, \mathsf{add}^{(2)}\}$。可定义以下重写模型：

$$\begin{aligned}\mathcal{R} = \{ \quad & \mathsf{add}(\mathsf{zero}, y) \to y\,, \\ & \mathsf{add}(\mathsf{s}(x), y) \to \mathsf{s}(\mathsf{add}(x, y)) \quad \}。\end{aligned}$$

以加法表达式的角度，例 2.5 的两条重写规则可以直观地看作：(1) 表达式 $0+y$ 可“计算”成 y；(2) 表达式 $(x+1)+y$ 可“计算”成 $(x+y)+1$。

准确地、形式化地定义这个过程，则有了**重写**的概念。

定义 2.6 (重写)： 给定重写模型 $\mathcal{R}$，项表达式 $u, v \in \mathcal{T}(\mathcal{F}, \mathcal{X})$，位置 $p \in \mathcal{P}os(u)$，重写规则 $(l \to r) \in \mathcal{R}$。如果存在代换 σ 满足 $u|_p = \sigma(l)$ 以及 $v = u[\sigma(r)]_p$，则称 u 在位置 p 应用重写规则 $l \to r$ 重写为 v，记作 $u \to^{p}_{l\to r} v$。记号中的 p 和 $l \to r$ 可忽略不写。

重写的过程可看作归约、化简的过程。它把项表达式 u 中存在的某个左项 l 的实例替换成了右项 r 相对应的实例（相同的代换 σ）。从树的角度来看，重写进行了位置 p 的子树的替换，如图 2.1。

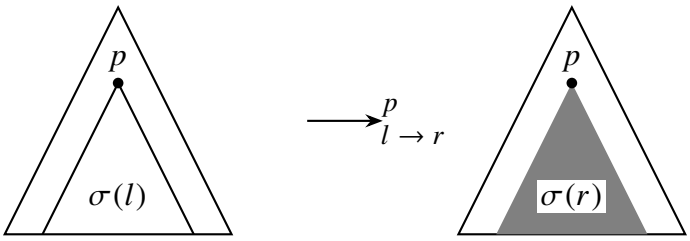


图 2.1　重写的树视图

例 2.6： 利用例 2.5 的重写模型 $\mathcal{R}$，可以得到以下重写序列，其中被重写替换的子项表达式（即重写规则左项的实例）用下划线标出：

$$\begin{aligned} t_1 &= \underline{\mathsf{add}(\mathsf{s}(\mathsf{zero}), \mathsf{zero})} \to \mathsf{s}(\underline{\mathsf{add}(\mathsf{zero}, \mathsf{zero})}) \to \mathsf{s}(\mathsf{zero})\ ; \\ t_2 &= \underline{\mathsf{add}(\mathsf{s}^2(\mathsf{zero}), \mathsf{s}(\mathsf{zero}))} \to \mathsf{s}(\underline{\mathsf{add}(\mathsf{s}(\mathsf{zero}), \mathsf{s}(\mathsf{zero}))}) \\ &\quad \to \mathsf{s}^2(\underline{\mathsf{add}(\mathsf{zero}, \mathsf{s}(\mathsf{zero}))}) \to \mathsf{s}^2(\mathsf{s}(\mathsf{zero}))\ . \end{aligned}$$

从上例不难看出，例 2.5 的重写模型 $\mathcal{R}$ 定义了自然数加法的计算方式，即通过重写规则赋予了抽象函数符号 add 自然数加法的语义。由于 $\mathcal{R}$ 不满足自然数乘

法的性质，因此无法将 add 解释为自然数乘法符号。因此，重写规则的应用方式（即重写）给重写规则规定了语义，而重写规则又给函数符号赋予了语义。

重写可以看作一种计算过程，也可以看作一种关系。

定义 2.7 (重写关系)： 给定定义在词汇表 $\mathcal{F}$ 上的重写模型 $\mathcal{R}$，$\mathcal{R}$ 对应的重写关系 $\to_{\mathcal{R}}$ 定义为 $\mathcal{T}(\mathcal{F},\mathcal{X})$ 上的二元关系：

$$\to_{\mathcal{R}} \stackrel{\text{def}}{=} \{\langle u,v\rangle \mid \text{存在 } p \text{ 和 } (l\to r)\in\mathcal{R}，\text{满足 } u\to^{p}_{l\to r} v\}$$

给定重写关系 $\to_{\mathcal{R}}$，它的对称闭包记作 $\leftrightarrow_{\mathcal{R}}$，自反传递闭包记作 $\to^{*}_{\mathcal{R}}$，自反对称传递闭包记作 $\leftrightarrow^{*}_{\mathcal{R}}$。

重写是一种非确定的技术。重写的非确定性体现在两个方面：(1) 给定项表达式 t，重写发生的位置 p 允许是任意的；(2) 给定项表达式 t，重写应用的重写规则 $l\to r$ 允许是任意的。例如在例 2.7 中，从项表达式 t 出发，产生了两条不同的重写序列。第一条序列先在位置 Λ（即树的根结点）进行重写，然后继续在位置 Λ 进行重写；第二条序列则先在位置 2（即根结点的第 2 个子结点）进行重写，然后继续在位置 Λ 进行重写。

例 2.7： 利用例 2.5 的重写模型 $\mathcal{R}$，可以得到以下重写序列：

$$\begin{aligned} t &= \underline{\mathsf{add}(\mathsf{zero},\mathsf{add}(\mathsf{zero},x))} \to^{\Lambda} \underline{\mathsf{add}(\mathsf{zero},x)} \to^{\Lambda} x \\ t &= \mathsf{add}(\mathsf{zero},\underline{\mathsf{add}(\mathsf{zero},x)}) \to^{2} \underline{\mathsf{add}(\mathsf{zero},x)} \to^{\Lambda} x \end{aligned}$$

重写过程不一定是终止的，即某项表达式 t 可以一直被重写，其过程不会停止。如例 2.8，利用重写模型 $\mathcal{R}_1$，可以得到无穷的重写序列 $a\to_{\mathcal{R}_1} a\to_{\mathcal{R}_1} a\to_{\mathcal{R}_1}\ldots$；利用重写模型 $\mathcal{R}_2$，可以得到无穷的重写序列 $f(a,b)\to_{\mathcal{R}_2} f(b,a)\to_{\mathcal{R}_2} f(a,b)\to_{\mathcal{R}_2}\ldots$。

例 2.8： 给定 $\mathcal{F}_1=\{a^{(0)}\}$，定义

$$\mathcal{R}_1 = \{a\to a\};$$

给定 $\mathcal{F}_2=\{a^{(0)},b^{(0)},f^{(2)}\}$，定义

$$\mathcal{R}_2 = \{f(x,y)\to f(y,x)\};$$

定义 2.8 (终止性)： 给定重写关系 $\to_{\mathcal{R}}$，如果对任意项表达式 t 都不存在无穷的重写序列 $t \to_{\mathcal{R}} t_1 \to_{\mathcal{R}} t_2 \to_{\mathcal{R}} \ldots$，则称重写关系 $\to_{\mathcal{R}}$ 是终止的（terminating）。重写模型 $\mathcal{R}$ 是终止的，当且仅当重写关系 $\to_{\mathcal{R}}$ 是终止的。

定义 2.9 (范式)： 给定重写模型 $\mathcal{R}$，如果项表达式 s 无法应用任意重写规则 $(l \to r) \in \mathcal{R}$ 进行重写，则称 s 是（关于 $\mathcal{R}$ 的）范式。

引理 2.1： 如果重写模型 $\mathcal{R}$ 是终止的，那么对任意项表达式 $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$，存在范式 $s \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ 满足 $t \to^*_{\mathcal{R}} s$。此时称 s 是 t（关于 $\mathcal{R}$）的范式，重写序列 $t \to^*_{\mathcal{R}} s$ 可记作 $t \to^!_{\mathcal{R}} s$。求范式的过程称作规范化（normalization）。

重写的终止性是重写领域研究的重要问题之一。给定任意一个重写模型 $\mathcal{R}$，判定 $\mathcal{R}$ 是否终止的问题，是一个不可判定（undecidable）的问题[74,75]。重写模型的终止性问题不是本文讨论的主题，在此不详述，更多探讨可参考文献 [52,76–80]。

由于重写的非确定性，即使重写模型是终止的，任意项表达式的范式也不一定是唯一的。如例 2.9，根据 $\mathcal{R}$，有 $a \to^!_{\mathcal{R}} b$ 以及 $a \to^!_{\mathcal{R}} c$，即 b 和 c 都是关于 $\mathcal{R}$ 的 a 的范式，可见 a 的范式并不唯一。

例 2.9： 给定 $\mathcal{F} = \{a^{(0)}, b^{(0)}, c^{(0)}\}$，定义

$$\mathcal{R} \;=\; \{a \to b\,, a \to c\}\ 。$$

定义 2.10 (合流性)： 给定重写关系 $\to_{\mathcal{R}}$，如果对任意满足 $u \;{}^*_{\mathcal{R}}\!\leftarrow s \to^*_{\mathcal{R}} v$ 的项表达式 s, u, v，都存在 t 使 $u \to^*_{\mathcal{R}} t \;{}^*_{\mathcal{R}}\!\leftarrow v$，则称 $\to_{\mathcal{R}}$ 是合流的（confluent）。重写模型 $\mathcal{R}$ 是合流的，当且仅当重写关系 $\to_{\mathcal{R}}$ 是合流的。

重写的合流性也是重写领域研究的重要问题之一。给定任意一个重写模型 $\mathcal{R}$，判定 $\mathcal{R}$ 是否合流的问题，也是一个不可判定的问题[81]。与终止性类似，重写模型的合流性问题不是本文讨论的重点，在此不详述，更多探讨可参考文献 [52,82–89]。

需要注意的是，把重写看作计算的过程，终止性保证计算结果存在，而合流性保证计算结果唯一，即计算过程的非确定性不影响计算结果的确定性。

引理 2.2： 如果重写模型 $\mathcal{R}$ 是终止的且合流的，那么任意项表达式 t 关于 $\mathcal{R}$ 的范式存在且唯一，记作 $t{\downarrow_{\mathcal{R}}}$。在 $\mathcal{R}$ 已知的情况下，简记作 $t{\downarrow}$。

2.3.3　等式规则与等价关系

重写规则是有方向性的，因此重写是不可逆的过程。给定 $s \to_{\mathcal{R}} t$，s 和 t 处于不等价的关系。然而在某些情况下，我们需要描述两个项表达式之间等价的关系。例如针对例 2.5 的重写模型，项表达式 $\mathsf{add}(x, \mathsf{zero})$ 已经是自身的范式，即 $\mathsf{add}(x, \mathsf{zero}) = \mathsf{add}(x, \mathsf{zero}){\downarrow}$。但正如之前的描述，我们想赋予 add 自然数加法的语义。我们知道交换律是自然数加法具有的性质，在此意义上，我们认为项表达式 $\mathsf{add}(x, \mathsf{zero})$ 应该可以进一步化简为 x。因此交换律的语义应该以形如 $\mathsf{add}(x, y) \to \mathsf{add}(y, x)$ 的重写规则被加入到该重写模型中。然而，从例 2.8 的 $\mathcal{R}_2$ 可以看出，这种形式的重写规则会破坏重写模型的终止性，从而破坏“计算”的终止性。

为此，需要引入等式规则（equation）与等价关系的概念。

定义 2.11 (等式规则)： 给定词汇表 $\mathcal{F}$，一条等式规则是由项表达式 $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ 组成的无序对（无序二元组），记作 $l \sim r$。

定义 2.12 (等价模型[53])： 给定词汇表 $\mathcal{F}$，等价模型是一个由若干等式规则组成的集合 $\mathcal{E} = \{l_i \sim r_i\}_i$。

由于等式规则是无序对，因此 $l \sim r$ 等同于 $r \sim l$。$(l \sim r) \in \mathcal{E}$ 当且仅当 $(r \sim l) \in \mathcal{E}$。

例 2.10： 假定 $\mathcal{F} = \{\mathsf{add}^{(2)}\}$。可定义以下等价模型：

$$\mathcal{E} = \{ \quad \mathsf{add}(x, y) \sim \mathsf{add}(y, x)\ , \\ \mathsf{add}(\mathsf{add}(x, y), z) \sim \mathsf{add}(x, \mathsf{add}(y, z)) \quad \}。$$

例 2.10 的规则分别定义了函数符号 add 的交换律和结合律。

应用等式规则的方式与应用重写规则的方式类似：

定义 2.13 (等价)： 给定等价模型 $\mathcal{E}$，项表达式 $u, v \in \mathcal{T}(\mathcal{F}, \mathcal{X})$，位置 $p \in \mathcal{P}os(u)$，等式规则 $(l \sim r) \in \mathcal{E}$。如果存在代换 σ 满足 $u|_p = \sigma(l)$ 以及 $v = u[\sigma(r)]_p$，则称 u 和 v 在位置 p 关于等式规则 $l \sim r$ 等价，记作 $u \leftrightarrow^p_{l \sim r} v$。记号中的 p 和 $l \sim r$ 可忽略不写。

由于 $(l \sim r) \in \mathcal{E}$ 当且仅当 $(r \sim l) \in \mathcal{E}$，因此 $u \leftrightarrow^p v$ 成立当且仅当 $v \leftrightarrow^p u$ 成立。记号 $\leftrightarrow$ 表明了应用等式规则的过程是个对称、可逆的过程。

例 2.11： 利用例 2.10 的等价模型，可得到以下等价序列：

$$\mathsf{add}(\mathsf{add}(x,y),z) \leftrightarrow^{1} \mathsf{add}(\mathsf{add}(y,x),z) \leftrightarrow^{\Lambda} \mathsf{add}(y,\mathsf{add}(x,z)) \leftrightarrow^{\Lambda} \mathsf{add}(\mathsf{add}(x,z),y)。$$

定义 2.14 (等价关系)： 给定定义在词汇表 $\mathcal{F}$ 上的等价模型 $\mathcal{E}$，$\mathcal{E}$ 对应的等价关系 $\leftrightarrow_{\mathcal{E}}$ 定义为 $\mathcal{T}(\mathcal{F},\mathcal{X})$ 上的二元关系：

$$\leftrightarrow_{\mathcal{E}} \stackrel{\text{def}}{=} \{\langle u,v\rangle \mid \text{存在 } p \text{ 和 } (l \sim r) \in \mathcal{E}，\text{满足 } u \leftrightarrow^{p}_{l\sim r} v\}$$

给定等价关系 $\leftrightarrow_{\mathcal{E}}$，它的对称闭包是它自身，自反传递闭包记作 $\leftrightarrow^{*}_{\mathcal{E}}$。

由于等价关系是对称的，任意等价关系 $u \leftrightarrow_{\mathcal{E}} v$ 都可以扩展成一个无穷等价序列 $u \leftrightarrow^{*}_{\mathcal{E}} v$。因此区分等价关系 $\leftrightarrow_{\mathcal{E}}$ 及其自反传递闭包 $\leftrightarrow^{*}_{\mathcal{E}}$ 并无实际意义。我们将 $u \leftrightarrow^{*}_{\mathcal{E}} v$ 记作 $u =_{\mathcal{E}} v$。

2.4　重写模型的扩展

本小节介绍重写模型的两种经典扩展：(1) 模重写模型，能描述等价关系的重写模型；(2) 条件重写模型，能描述条件控制的重写模型。

2.4.1　模重写

等价关系为抽象的函数符号赋予了“等价”的语义。等价关系的引入，是为了增强重写的能力。Peterson 和 Stickel 提出 模重写技术的目的就是定义一种二者融合的方式。

定义 2.15 (模重写模型[53])： 给定词汇表 $\mathcal{F}$，模重写模型是一个由重写模型 $\mathcal{R}$ 和等价模型 $\mathcal{E}$ 组成的有序对（二元组），记作 $\mathcal{R}_{\mathcal{E}} \stackrel{\text{def}}{=} \langle \mathcal{R},\mathcal{E}\rangle$。

定义 2.16 (模重写)： 给定模重写模型 $\mathcal{R}_{\mathcal{E}}$，项表达式 $u,v \in \mathcal{T}(\mathcal{F},\mathcal{X})$，位置 $p \in \mathcal{P}os(u)$，重写规则 $(l \to r) \in \mathcal{R}$。如果存在代换 σ 满足 $u|_p =_{\mathcal{E}} \sigma(l)$ 以及 $v = u[\sigma(r)]_p$，则称 u 在位置 p 应用重写规则 $l \to r$ 模重写为 v，记作 $u \to^{p}_{(l\to r)_{\mathcal{E}}} v$。记号中的 p 和 $(l \to r)_{\mathcal{E}}$ 可忽略不写。

重写技术替换的是项表达式 u 中存在的规则左项的实例 $\sigma(l)$，而模重写技术替换的则是 u 中与规则左项实例 $\sigma(l)$ 关于 $\mathcal{E}$ 等价的子项表达式。这使得重写过程得以“利用”各函数符号的等价语义。

例 2.12： 假定 $\mathcal{F} = \{\mathsf{zero}^{(0)}, \mathsf{s}^{(1)}, \mathsf{add}^{(2)}\}$。可定义模重写模型 $\mathcal{R}_\mathcal{E}$，其中：

$$\begin{aligned}\mathcal{R} = \{ \quad & \mathsf{add}(\mathsf{zero}, y) \to y\,, \\ & \mathsf{add}(\mathsf{s}(x), y) \to \mathsf{s}(\mathsf{add}(x, y)) \quad \}\ ; \\ \mathcal{E} = \{ \quad & \mathsf{add}(x, y) \sim \mathsf{add}(y, x)\,, \\ & \mathsf{add}(\mathsf{add}(x, y), z) \sim \mathsf{add}(x, \mathsf{add}(y, z)) \quad \}。\end{aligned}$$

可得到以下模重写序列：

$$\begin{aligned} t_1 &= \underline{\mathsf{add}(x, \mathsf{s}(\mathsf{zero}))} \to \mathsf{s}(\underline{\mathsf{add}(\mathsf{zero}, x)}) \to \mathsf{s}(x)\ ; \\ t_2 &= \underline{\mathsf{add}(\mathsf{add}(x, \mathsf{s}(\mathsf{zero})), y)} \to \mathsf{s}(\underline{\mathsf{add}(\mathsf{zero}, \mathsf{add}(x, y))}) \to \mathsf{s}(\mathsf{add}(x, y))\ 。 \end{aligned}$$

需要注意的是，给定重写模型 $\mathcal{R}$ 和等价模型 $\mathcal{E}$，模重写只定义了二者规则应用的其中一种方式，即规定了模重写模型的特定的语义。利用别的规则应用策略，还可以得到不同的模型语义，例如 Lankford 和 Ballantyne 提出的*类重写模型*[71]。

类似于重写，我们可以定义模重写对应的模重写关系、终止性、范式和合流性。

定义 2.17 (模重写关系)： 给定定义在词汇表 $\mathcal{F}$ 上的模重写模型 $\mathcal{R}_\mathcal{E}$，$\mathcal{R}_\mathcal{E}$ 对应的模重写关系 $\to_{\mathcal{R}_\mathcal{E}}$ 定义为 $\mathcal{T}(\mathcal{F}, \mathcal{X})$ 上的二元关系：

$$\to_{\mathcal{R}_\mathcal{E}} \stackrel{\text{def}}{=} \{\langle u, v\rangle \mid \text{存在 } p \text{ 和 } (l \to r) \in \mathcal{R}，\text{满足 } u \to^{p}_{(l\to r)_\mathcal{E}} v\}$$

给定模重写关系 $\to_{\mathcal{R}_\mathcal{E}}$，它的对称闭包记作 $\leftrightarrow_{\mathcal{R}_\mathcal{E}}$，自反传递闭包记作 $\to^*_{\mathcal{R}_\mathcal{E}}$，自反对称传递闭包记作 $\leftrightarrow^*_{\mathcal{R}_\mathcal{E}}$。

定义 2.18： 给定模重写关系 $\to_{\mathcal{R}_\mathcal{E}}$，如果对任意项表达式 t 都不存在无穷的模重写序列 $t \to_{\mathcal{R}_\mathcal{E}} t_1 \to_{\mathcal{R}_\mathcal{E}} t_2 \to_{\mathcal{R}_\mathcal{E}} \ldots$，则称模重写关系 $\to_{\mathcal{R}_\mathcal{E}}$ 是终止的。模重写模型 $\mathcal{R}_\mathcal{E}$ 是终止的，当且仅当模重写关系 $\to_{\mathcal{R}_\mathcal{E}}$ 是终止的。

定义 2.19： 给定模重写模型 $\mathcal{R}_\mathcal{E}$，如果项表达式 s 无法应用任意重写规则 $(l \to r) \in \mathcal{R}$ 进行模重写，则称 s 是（关于 $\mathcal{R}_\mathcal{E}$ 的）范式。

引理 2.3： 如果模重写模型 $\mathcal{R}_\mathcal{E}$ 是终止的，那么对任意项表达式 $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$，存在范式 $s \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ 满足 $t \to^*_{\mathcal{R}_\mathcal{E}} s$。此时称 s 是 t（关于 $\mathcal{R}_\mathcal{E}$）的范式，模重写序列 $t \to^*_{\mathcal{R}_\mathcal{E}} s$ 可记作 $t \to^!_{\mathcal{R}_\mathcal{E}} s$。

定义 2.20： 给定模重写关系 $\to_{\mathcal{R}_\mathcal{E}}$，如果对任意满足 $u \ {}_{\mathcal{R}_\mathcal{E}}^{*}\!\leftarrow s \to_{\mathcal{R}_\mathcal{E}}^{*} v$ 的项表达式 s,u,v，都存在 t 和 t' 使 $u \to_{\mathcal{R}_\mathcal{E}}^{*} t =_\mathcal{E} t' \ {}_{\mathcal{R}_\mathcal{E}}^{*}\!\leftarrow v$，则称 $\to_{\mathcal{R}_\mathcal{E}}$ 是合流的。模重写模型 $\mathcal{R}_\mathcal{E}$ 是合流的，当且仅当模重写关系 $\to_{\mathcal{R}_\mathcal{E}}$ 是合流的。

引理 2.4： 如果模重写模型 $\mathcal{R}_\mathcal{E}$ 是终止的且合流的，那么任意项表达式 t 关于 $\mathcal{R}_\mathcal{E}$ 的范式存在且关于 $\mathcal{E}$ 等价。也就是说，如果 $t \to_{\mathcal{R}_\mathcal{E}}^{!} s_1$，$t \to_{\mathcal{R}_\mathcal{E}}^{!} s_2$，那么 $s_1 =_\mathcal{E} s_2$。此时也可称 t 关于 $\mathcal{R}_\mathcal{E}$ 的范式关于 $=_\mathcal{E}$ 唯一，将任一范式记作 $t{\downarrow_{\mathcal{R}_\mathcal{E}}}$。在 $\mathcal{R}_\mathcal{E}$ 已知的情况下，简记作 $t{\downarrow}$。

模重写技术可以看作是将重写技术中的语法等价（相等关系 $=$）扩展为语义等价（等价关系 $=_\mathcal{E}$）。重写模型是模重写模型的等价关系 $=_\mathcal{E}$ 为空时的特例，即 $\mathcal{R} = \mathcal{R}_\varnothing$。关于模重写的终止性和合流性的研究，可参考文献 [52,90–95]。

2.4.2　条件重写

对于标准的重写技术，触发重写的条件是模式匹配成功。假如给定形如 $f(x)$ 的项表达式，我们希望当 $x \geq 2$ 时，$f(x)$ 可以重写为 a；当 $x < 2$ 时，$f(x)$ 可以重写为 b。那么根据例 2.5，我们需要增加如下规则：

$$\begin{aligned} f(\mathsf{s}^2(x)) &\to a \\ f(\mathsf{zero}) &\to b \\ f(\mathsf{s}(\mathsf{zero})) &\to b \end{aligned}$$

如果判断是否满足 $x \geq 3$，则右项为 b 的规则需要增加至 3 条；以此类推，如果判断是否满足 $x \geq n$，右项为 b 的规则需要增加至 n 条。诸如此类的条件约束需要通过重写规则左项的模式来进行设计，这就给设计规则的过程带来极大不便。

给重写规则增加条件约束，则可以大大地增加重写规则对条件控制的表达能力。例如我们希望可以通过以下形式的规则来表示上述的情况：

$$\begin{aligned} f(x) &\to a \quad \text{if} \quad x \geq \mathsf{s}^2(\mathsf{zero}) \\ f(x) &\to b \quad \text{if} \quad x < \mathsf{s}^2(\mathsf{zero}) \end{aligned}$$

于是，人们提出了条件重写模型。

定义 2.21 (条件重写规则)： 给定词汇表 $\mathcal{F}$，条件重写规则由重写规则和若干等式构成，形如

$$l \to r \quad \Leftarrow \quad u_1 = v_1 \wedge \ldots \wedge u_n = v_n \ ,$$

其中 $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$，对 $1 \le i \le n$ 有 $u_i, v_i \in \mathcal{T}(\mathcal{F}, \mathcal{X})$。其中 l 称作该条件重写规则的左项，r 称作该条件重写规则的右项，$u_i = v_i$ 称作该条件重写规则的条件约束。

定义 2.22 (条件重写模型[54])： 给定词汇表 $\mathcal{F}$，条件重写模型是一个由若干条件重写规则组成的集合 $\mathcal{R} = \{l_i \to r_i \Leftarrow C_i\}_i$。

例 2.13： 假定 $\mathcal{F} = \{\mathsf{zero}^{(0)}, \mathsf{s}^{(1)}, \mathsf{add}^{(2)}, f^{(2)}, a^{(0)}, b^{(0)}, \mathsf{true}^{(0)}, \mathsf{false}^{(0)}, \mathsf{not}^{(1)}, \mathsf{lessthan}^{(2)}\}$。可定义以下条件重写模型：

$$\mathcal{R} = \{ \quad f(x) \to a \ \Leftarrow \ \mathsf{not}(\mathsf{lessthan}(x, \mathsf{s}^2(\mathsf{zero}))) = \mathsf{true} \ , \\ f(x) \to b \ \Leftarrow \ \mathsf{lessthan}(x, \mathsf{s}^2(\mathsf{zero})) = \mathsf{true} \quad \}。$$

例 2.13 中的两条规则对应了本小节开头提出的重写需求。

虽然我们给出了条件重写规则的语法形式，但是其语义并不清晰。条件重写规则里的条件约束 $u_i = v_i$，是代表 u_i 和 v_i 语法上的相等呢？还是代表 $u_i \leftrightarrow^* v_i$ 成立呢？是代表存在 t 使 $u_i \to^* t \ {}^*\!\leftarrow v_i$ 呢？还是 $u_i \to^* v_i$ 成立呢？关于条件约束的不同语义解释有多种讨论[96–98]，这里我们采用最常用的实现方法：存在 t 使 $u_i \to^* t \ {}^*\!\leftarrow v_i$ 成立。

定义 2.23 (条件重写)： 给定条件重写模型 $\mathcal{R}$，项表达式 $u, v \in \mathcal{T}(\mathcal{F}, \mathcal{X})$，位置 $p \in \mathcal{P}os(u)$，条件重写规则 $(l \to r \Leftarrow C_i) \in \mathcal{R}$。如果存在代换 σ 满足 $u|_p = \sigma(l)$，$v = u[\sigma(r)]_p$，以及对任意 $(u_j = v_j) \in C_i$ 存在 t_j 使 $u_j \to^* t_j \ {}^*\!\leftarrow v_j$，则称 u 在位置 p 应用条件重写规则 $l \to r \Leftarrow C_i$ 重写为 v，记作 $u \to^p_{l \to r} v$。记号中的 p 和 $l \to r$ 可忽略不写。

条件重写的直观意思是，被重写的子项表达式不仅需要是重写规则左项 l 的实例，且其对应的代换 σ 需要满足该规则的所有条件约束。

针对例 2.13，为了实现条件约束的求解，还需要加入以下规则：

$$\mathsf{lessthan}(x, \mathsf{zero}) \quad \to \quad \mathsf{false}$$

$$\begin{aligned}
\mathsf{lessthan}(\mathsf{zero}, \mathsf{s}(y)) &\to \mathsf{true} \\
\mathsf{lessthan}(\mathsf{s}(x), \mathsf{s}(y)) &\to \mathsf{lessthan}(x, y) \\
\mathsf{not}(\mathsf{false}) &\to \mathsf{true} \\
\mathsf{not}(\mathsf{true}) &\to \mathsf{false} \quad 。
\end{aligned}$$

重写模型对应于条件重写模型中所有条件约束均为空的特殊情况。重写模型的相关概念，包括重写关系、终止性、范式及合流性，均可自然地扩展成条件重写模型的对应概念，在此不再赘述。给定任意模重写模型 $\mathcal{R}_{\mathcal{E}}$，将其中的重写模型 $\mathcal{R}$ 替换成条件重写模型，则得到条件模重写模型及其相关概念。

2.5　规范化条件重写模型

在模重写和条件重写的基础上，本文提出一种新的重写模型扩展，叫规范化条件重写模型。

条件重写模型在应用某条重写规则时，需要判断其条件约束是否成立，即是否存在 t 使 $u_i \to^* t \ {}^*\!\leftarrow v_i$ 成立，这个过程是不可判定的[81]。理想的做法是通过 u_i 和 v_i 的范式来确定条件是否成立，即判断是否有 $u_i{\downarrow} = v_i{\downarrow}$。此时，只需要在语法上判断是否相等即可。这就要求该条件重写模型是终止的，或者至少要求用于求范式的重写规则是终止的[98–100]。另一方面，根据我们对实际系统进行建模的经验，用于描述系统行为的规则（如例 2.13 的 $f(x) \to a$ 和 $f(x) \to b$）与用于求解条件约束的规则（如上述关于 lessthan 的规则）在一般情况下是独立的，且由于条件约束对应于目标系统中的条件判断，因此满足终止性与合流性。所以这两种规则在建模应用时存在层次关系，将它们归属于同一个重写规则集合从概念上和技术上都是欠妥的。

本文提出的规范化（条件）重写模型，则是基于上述考虑，将（条件）模重写模型进一步划分层次，在模型层面对确定性行为和非确定性行为予以区分。

定义 2.24 (规范化重写模型)： 给定词汇表 $\mathcal{F}$，规范化重写模型是一个由重写模型 $\mathcal{R}, \mathcal{S}$ 和等价模型 $\mathcal{E}$ 组成的三元组，记作 $\mathcal{R}_{\mathcal{SE}} \stackrel{\text{def}}{=} \langle \mathcal{R}, \mathcal{S}, \mathcal{E} \rangle$。其中由 $\mathcal{S}$ 和 $\mathcal{E}$ 组成的模重写模型 $\mathcal{S}_{\mathcal{E}}$ 必须满足终止性和合流性。

重写不一定是终止的，因此不能将任意重写模型的重写过程看作是化简的过程。例如规则 $a \to f(a)$ 会将项表达式 a 重写为更加复杂的表达式。但是当重写模型具有终止性时，重写过程就可以看作是一个化简的过程。规范化重写模型 $\mathcal{R}_{\mathcal{SE}}$ 要求 $\mathcal{S}_{\mathcal{E}}$ 是终止且合流的，根据引理 2.4，这意味着 $\mathcal{S}_{\mathcal{E}}$ 所描述的行为具有确定性

——其重写结果不受重写过程的不确定性影响，得到的范式在 $=_{\mathcal{E}}$ 等价的意义下唯一。这使得 $\mathcal{S}_{\mathcal{E}}$ 可看作一个良定义的化简函数：计算结果（范式）存在且唯一。因此，重写模型 $\mathcal{S}$ 可称作化简模型（simplifier），其对应的重写规则可称作化简规则（simplification rule）。

下面给出规范化重写模型的规则应用策略，即其形式化语义：

定义 2.25 (规范化重写)： 给定规范化重写模型 $\mathcal{R}_{\mathcal{SE}}$，项表达式 $u, v \in \mathcal{T}(\mathcal{F}, \mathcal{X})$，重写规则 $(l \to r) \in \mathcal{R}$。如果存在位置 p 和项表达式 s,t 满足 $u{\downarrow_{\mathcal{S}_{\mathcal{E}}}} =_{\mathcal{E}} s \to^{p}_{l\to r} t$ 且 $v = t{\downarrow_{\mathcal{S}_{\mathcal{E}}}}$，则称 u 应用重写规则 $l \to r$ 规范化重写为 v，记作 $u \to_{(l\to r)_{\mathcal{SE}}} v$。记号中的 $(l \to r)_{\mathcal{SE}}$ 在不存在二义性时可忽略不写。

简单地说，如果 $u \to_{(l\to r)_{\mathcal{SE}}} v$，则 u 和 v 之间存在关系 $u \to^{!}_{\mathcal{S}_{\mathcal{E}}} u{\downarrow_{\mathcal{S}_{\mathcal{E}}}} =_{\mathcal{E}} s \to^{p}_{l\to r} t \to^{!}_{\mathcal{S}_{\mathcal{E}}} t{\downarrow_{\mathcal{S}_{\mathcal{E}}}} = v$。从技术上看，给定项表达式 u，要想对 u 进行规范化重写，需要先对 u 进行规范化，得到 u 关于 $\mathcal{S}_{\mathcal{E}}$ 的范式 $u{\downarrow_{\mathcal{S}_{\mathcal{E}}}}$。然后在 $u{\downarrow_{\mathcal{S}_{\mathcal{E}}}}$ 关于 $=_{\mathcal{E}}$ 的等价类中选定项表达式 s，利用 $\mathcal{R}$ 对其进行标准重写得到项表达式 t。最后对 t 进行规范化得到其关于 $\mathcal{S}_{\mathcal{E}}$ 的范式 $t{\downarrow_{\mathcal{S}_{\mathcal{E}}}}$。

规范化重写技术在应用重写规则前，要求项表达式必须是范式。在应用重写规则后，又显式地进行规范化，使最终得到的项表达式成为范式。这就保证对于任意规范化重写序列 $u_0 \to u_1 \to \ldots \to u_n$，对任意 $i \in [1, n]$ 满足 u_i 是关于 $\mathcal{S}_{\mathcal{E}}$ 的范式，即重写序列中的项表达式始终处于“最简”形式。从行为的角度分析，规范化重写保证了每次执行非确定性行为前，所有可能的确定性行为都必须被执行完毕；而每次执行完一次非确定性行为，接下来可能发生的所有确定性行为都需要被执行。

规范化重写模型通过对规则分类，使模型的语义更加清晰：$\mathcal{E}$ 描述与等价相关的语义，$\mathcal{S}$ 描述与函数计算、形式简化、确定性行为相关的语义，$\mathcal{R}$ 描述最核心的、非确定性行为相关的语义。

例 2.14： 假定 $\mathcal{F} = \{::^{(2)}, \mathsf{farmer}^{(1)}, \mathsf{wolf}^{(1)}, \mathsf{lamb}^{(1)}, \mathsf{grass}^{(1)}, \mathsf{left}^{(0)}, \mathsf{right}^{(0)}, \mathsf{change}^{(1)}\}$，其中函数符号 :: 采用中缀记法，即 $x :: y$ 表示 $::(x, y)$。可定义规范化重写模型 $\mathcal{R}_{\mathcal{SE}}$，其中：

$$
\begin{aligned}
\mathcal{E} = \{ \quad & x :: y \sim y :: x \,, \\
& (x :: y) :: z \sim x :: (y :: z) \quad \} \,; \\
\mathcal{S} = \{ \quad & \mathsf{change}(\mathsf{left}) \to \mathsf{right} \,, \\
& \mathsf{change}(\mathsf{right}) \to \mathsf{left} \quad \} \,;
\end{aligned}
$$

$$\begin{aligned}\mathcal{R} = \{\quad & \mathsf{farmer}(x) \to_1 \mathsf{farmer}(\mathsf{change}(x)),\\ & \mathsf{farmer}(x)::\mathsf{wolf}(x) \to_2 \mathsf{farmer}(\mathsf{change}(x))::\mathsf{wolf}(\mathsf{change}(x)),\\ & \mathsf{farmer}(x)::\mathsf{lamb}(x) \to_3 \mathsf{farmer}(\mathsf{change}(x))::\mathsf{lamb}(\mathsf{change}(x)),\\ & \mathsf{farmer}(x)::\mathsf{grass}(x) \to_4 \mathsf{farmer}(\mathsf{change}(x))::\mathsf{grass}(\mathsf{change}(x))\ \}。\end{aligned}$$

可得到以下规范化重写序列：

$$\begin{aligned} t \quad = \quad & \mathsf{farmer}(\mathsf{left})::\mathsf{wolf}(\mathsf{left})::\mathsf{lamb}(\mathsf{left})::\mathsf{grass}(\mathsf{left})\\ \to_3 \quad & \mathsf{farmer}(\mathsf{right})::\mathsf{lamb}(\mathsf{right})::\mathsf{wolf}(\mathsf{left})::\mathsf{grass}(\mathsf{left})\\ \to_1 \quad & \mathsf{farmer}(\mathsf{left})::\mathsf{lamb}(\mathsf{right})::\mathsf{wolf}(\mathsf{left})::\mathsf{grass}(\mathsf{left})\\ \to_2 \quad & \mathsf{farmer}(\mathsf{right})::\mathsf{wolf}(\mathsf{right})::\mathsf{lamb}(\mathsf{right})::\mathsf{grass}(\mathsf{left})\\ \to_3 \quad & \mathsf{farmer}(\mathsf{left})::\mathsf{lamb}(\mathsf{left})::\mathsf{wolf}(\mathsf{right})::\mathsf{grass}(\mathsf{left})\\ \to_4 \quad & \mathsf{farmer}(\mathsf{right})::\mathsf{grass}(\mathsf{right})::\mathsf{wolf}(\mathsf{right})::\mathsf{lamb}(\mathsf{left})\\ \to_1 \quad & \mathsf{farmer}(\mathsf{left})::\mathsf{grass}(\mathsf{right})::\mathsf{wolf}(\mathsf{right})::\mathsf{lamb}(\mathsf{left})\\ \to_3 \quad & \mathsf{farmer}(\mathsf{right})::\mathsf{lamb}(\mathsf{right})::\mathsf{wolf}(\mathsf{right})::\mathsf{grass}(\mathsf{right})\ 。\end{aligned}$$

例 2.14 描述的是农夫过河的问题。left 和 right 分别表示河的左岸和右岸；$\mathsf{farmer}(x)$、$\mathsf{wolf}(x)$、$\mathsf{lamb}(x)$ 和 $\mathsf{grass}(x)$ 分别表示农夫、狼、羊和草当前所处位置为河的 x 岸；$\mathsf{change}(x)$ 表示 x 岸的对岸；函数符号 :: 作为连接符构成农夫、狼、羊和草在任一时刻的状态，如 $\mathsf{farmer}(\mathsf{left})::\mathsf{lamb}(\mathsf{left})::\mathsf{wolf}(\mathsf{right})::\mathsf{grass}(\mathsf{left})$ 表示当前状态为农夫在左岸、羊在左岸、狼在右岸、草在左岸。$\mathcal{E}$ 的等价规则分别表示函数符号 :: 具有交换律和结合律，因此 :: 连接的多个项表达式构成了一个多重集合，其中的括号可以省略。$\mathcal{S}$ 的化简规则定义了化简 change 的方式。$\mathcal{R}$ 的四条规则定义了系统状态发生改变的四种方式：(1) 农夫独自一人过河；(2) 农夫带着狼过河；(3) 农夫带羊过河；(4) 农夫带草过河。$\mathcal{R}$ 的规则用数字进行了编号。例 2.14 的重写序列描述了一种可能的方案，从农夫、狼、羊和草都处于左岸的初始状态，变成四者都处于右岸的状态。序列中用右箭头的数字下标表示规范化重写应用的规则编号。

定义 2.26 (规范化重写关系)： 给定定义在词汇表 $\mathcal{F}$ 上的规范化重写模型 $\mathcal{R}_{\mathcal{SE}}$，$\mathcal{R}_{\mathcal{SE}}$ 对应的规范化重写关系 $\to_{\mathcal{R}_{\mathcal{SE}}}$ 定义为 $\mathcal{T}(\mathcal{F},\mathcal{X})$ 上的二元关系：

$$\to_{\mathcal{R}_{\mathcal{SE}}} \ \stackrel{\text{def}}{=}\ \{\langle u, v\rangle \mid \text{存在 } (l \to r) \in \mathcal{R}，\text{满足 } u \to_{(l\to r)_{\mathcal{SE}}} v\}$$

给定规范化重写关系 $\to_{\mathcal{R}_{\mathcal{S}\mathcal{E}}}$，它的对称闭包记作 $\leftrightarrow_{\mathcal{R}_{\mathcal{S}\mathcal{E}}}$，自反传递闭包记作 $\to^*_{\mathcal{R}_{\mathcal{S}\mathcal{E}}}$，自反对称传递闭包记作 $\leftrightarrow^*_{\mathcal{R}_{\mathcal{S}\mathcal{E}}}$。

类似地，条件约束可以增加规范化重写模型对条件控制的建模能力，扩展得到规范化条件重写模型。

定义 2.27 (规范化条件重写模型)： 给定词汇表 $\mathcal{F}$，规范化条件重写模型是一个由条件重写模型 $\mathcal{R},\mathcal{S}$ 和等价模型 $\mathcal{E}$ 组成的三元组，记作 $\mathcal{R}_{\mathcal{S}\mathcal{E}} \stackrel{\text{def}}{=} \langle \mathcal{R},\mathcal{S},\mathcal{E}\rangle$。其中由 $\mathcal{S}$ 和 $\mathcal{E}$ 组成的条件模重写模型 $\mathcal{S}_{\mathcal{E}}$ 必须满足终止性和合流性。

上文已经提到，条件重写模型是重写模型的一种自然扩展。从语法上看，规范化条件重写模型仅仅是将规范化重写模型 $\langle \mathcal{R},\mathcal{S},\mathcal{E}\rangle$ 中的标准重写模型 $\mathcal{R}$ 和 $\mathcal{S}$ 扩展为条件重写模型，且同样对 $\mathcal{S}_{\mathcal{E}}$ 的终止性和合流性有要求。

虽然从语法上看，规范化条件重写模型只是简单地将重写规则替换为条件重写规则。但从语义上，却有别于简单的替换：

定义 2.28 (规范化条件重写)： 给定规范化条件重写模型 $\mathcal{R}_{\mathcal{S}\mathcal{E}}$，项表达式 $u,v \in \mathcal{T}(\mathcal{F},\mathcal{X})$，条件重写规则 $(l \to r \Leftarrow C) \in \mathcal{R}$。如果存在位置 p、代换 σ 和项表达式 s,t 满足：

(i) $u{\downarrow_{\mathcal{S}_{\mathcal{E}}}} =_{\mathcal{E}} s$；

(ii) $s|_p = \sigma(l)$，$t = s[\sigma(r)]_p$；

(iii) 对任意 $(u_j = v_j) \in C$，存在 w_j 使 $u_j \to^*_{\mathcal{S}_{\mathcal{E}}} w_j \ {}^*_{\mathcal{S}_{\mathcal{E}}}\!\!\leftarrow v_j$；

(iv) $v = t{\downarrow_{\mathcal{S}_{\mathcal{E}}}}$；

则称 u 应用条件重写规则 $l \to r \Leftarrow C$ 规范化条件重写为 v，记作 $u \to_{(l\to r)_{\mathcal{S}\mathcal{E}}} v$。记号中的 $(l \to r)_{\mathcal{S}\mathcal{E}}$ 在不存在二义性时可忽略不写。

与规范化重写类似，如果 $u \to_{(l\to r)_{\mathcal{S}\mathcal{E}}} v$，则 u 和 v 之间存在关系 $u \to^!_{\mathcal{S}_{\mathcal{E}}} u{\downarrow_{\mathcal{S}_{\mathcal{E}}}} =_{\mathcal{E}} s \to^p_{l\to r} t \to^!_{\mathcal{S}_{\mathcal{E}}} t{\downarrow_{\mathcal{S}_{\mathcal{E}}}} = v$。但需要注意其中的条件重写步骤 $s \to^p_{l\to r} t$，它判断条件约束 $u_j = v_j$ 是否成立时，不是递归地使用模型 $\mathcal{R}_{\mathcal{S}\mathcal{E}}$，而是直接使用 $\mathcal{S}_{\mathcal{E}}$。也就是说，$u_j$ 和 v_j 需要满足 $u_j \to^*_{\mathcal{S}_{\mathcal{E}}} w_j \ {}^*_{\mathcal{S}_{\mathcal{E}}}\!\!\leftarrow v_j$ 而不是 $u_j \to^*_{\mathcal{R}_{\mathcal{S}\mathcal{E}}} w_j \ {}^*_{\mathcal{R}_{\mathcal{S}\mathcal{E}}}\!\!\leftarrow v_j$。由于条件模重写模型 $\mathcal{S}_{\mathcal{E}}$ 是终止且合流的，$u_j \to^*_{\mathcal{S}_{\mathcal{E}}} w_j \ {}^*_{\mathcal{S}_{\mathcal{E}}}\!\!\leftarrow v_j$ 可以进一步简化为 $u_j{\downarrow_{\mathcal{S}_{\mathcal{E}}}} =_{\mathcal{E}} v_j{\downarrow_{\mathcal{S}_{\mathcal{E}}}}$，这就在一定程度上解决了本小节开头提出的不可判定性问题，使规范化条件重写从技术上可以被实现。

与规范化重写类似，从技术上看，规范化条件重写也必须经过规范化、条件重写、规范化三个过程。这保证了规范化条件重写序列中的任意项表达式都始终处于“最简”形式。

定义 2.29 (规范化条件重写关系)： 给定定义在词汇表 $\mathcal{F}$ 上的规范化条件重写模型 $\mathcal{R}_{\mathcal{SE}}$，$\mathcal{R}_{\mathcal{SE}}$ 对应的规范化条件重写关系 $\rightarrow_{\mathcal{R}_{\mathcal{SE}}}$ 定义为 $\mathcal{T}(\mathcal{F},\mathcal{X})$ 上的二元关系：

$$\rightarrow_{\mathcal{R}_{\mathcal{SE}}} \stackrel{\text{def}}{=} \{\langle u,v\rangle \mid \text{存在 } (l \rightarrow r \Leftarrow C) \in \mathcal{R}，\text{满足 } u \rightarrow_{(l\rightarrow r)_{\mathcal{SE}}} v\}$$

给定规范化条件重写关系 $\rightarrow_{\mathcal{R}_{\mathcal{SE}}}$，它的对称闭包记作 $\leftrightarrow_{\mathcal{R}_{\mathcal{SE}}}$，自反传递闭包记作 $\rightarrow^*_{\mathcal{R}_{\mathcal{SE}}}$，自反对称传递闭包记作 $\leftrightarrow^*_{\mathcal{R}_{\mathcal{SE}}}$。

例 2.15： 假定 $\mathcal{F}=\{\mathsf{zero}^{(0)},\mathsf{s}^{(1)},f^{(2)},a^{(0)},b^{(0)},\mathsf{true}^{(0)},\mathsf{false}^{(0)},\mathsf{not}^{(1)},\mathsf{lessthan}^{(2)}\}$。可定义规范化条件重写模型 $\mathcal{R}_{\mathcal{SE}}$，其中：

$$\begin{aligned}
\mathcal{E} = \quad & \varnothing\ ; \\
\mathcal{S} = \{ \quad & \mathsf{lessthan}(x,\mathsf{zero}) \rightarrow \mathsf{false}\ , \\
& \mathsf{lessthan}(\mathsf{zero},\mathsf{s}(y)) \rightarrow \mathsf{true}\ , \\
& \mathsf{lessthan}(\mathsf{s}(x),\mathsf{s}(y)) \rightarrow \mathsf{lessthan}(x,y)\ , \\
& \mathsf{not}(\mathsf{false}) \rightarrow \mathsf{true}\ , \\
& \mathsf{not}(\mathsf{true}) \rightarrow \mathsf{false} \qquad\qquad \}\ ; \\
\mathcal{R} = \{ \quad & f(x) \rightarrow a \ \Leftarrow\ \mathsf{not}(\mathsf{lessthan}(x,\mathsf{s}^2(\mathsf{zero}))) = \mathsf{true}\ , \\
& f(x) \rightarrow b \ \Leftarrow\ \mathsf{lessthan}(x,\mathsf{s}^2(\mathsf{zero})) = \mathsf{true} \qquad\quad \}。
\end{aligned}$$

例 2.15 的规范化条件重写模型对应例 2.13（及其新增规则）的条件重写模型。

例 2.16： 假定 $\mathcal{F}=\{\mathsf{zero}^{(0)},\mathsf{s}^{(1)},\mathsf{add}^{(2)},\mathsf{clock}^{(1)},\mathsf{broken}^{(0)},\mathsf{true}^{(0)},\mathsf{false}^{(0)},\mathsf{not}^{(1)},\mathsf{lessthan}^{(2)}\}$。可定义规范化条件重写模型 $\mathcal{R}_{\mathcal{SE}}$，其中：

$$\begin{aligned}
\mathcal{E} = \{ \quad & \mathsf{add}(x,y) \sim \mathsf{add}(y,x)\ , \\
& \mathsf{add}(\mathsf{add}(x,y),z) \sim \mathsf{add}(x,\mathsf{add}(y,z)) \quad \}\ ; \\
\mathcal{S} = \{ \quad & \mathsf{add}(\mathsf{zero},y) \rightarrow y\ , \\
& \mathsf{add}(\mathsf{s}(x),y) \rightarrow \mathsf{s}(\mathsf{add}(x,y))\ , \\
& \mathsf{lessthan}(x,\mathsf{zero}) \rightarrow \mathsf{false}\ , \\
& \mathsf{lessthan}(\mathsf{zero},\mathsf{s}(y)) \rightarrow \mathsf{true}\ , \\
& \mathsf{lessthan}(\mathsf{s}(x),\mathsf{s}(y)) \rightarrow \mathsf{lessthan}(x,y) \quad \}\ ; \\
\mathcal{R} = \{ \quad & \mathsf{clock}(x) \rightarrow_1 \mathsf{clock}(\mathsf{add}(x,\mathsf{s}(\mathsf{zero}))) \ \Leftarrow\ \mathsf{lessthan}(x,\mathsf{s}^{11}(\mathsf{zero})) = \mathsf{true}\ ,
\end{aligned}$$

$$\mathsf{clock}(x) \to_2 \mathsf{clock}(\mathsf{zero}) \Leftarrow \mathsf{lessthan}(x, \mathsf{s}^{11}(\mathsf{zero})) = \mathsf{false}\,,$$
$$\mathsf{clock}(x) \to_3 \mathsf{broken} \qquad \}。$$

可得到以下规范化条件重写序列：

$$\begin{aligned}
t &= \mathsf{clock}(\mathsf{s}^0(\mathsf{zero})) \to_1 \mathsf{clock}(\mathsf{s}^1(\mathsf{zero})) \to_1 \mathsf{clock}(\mathsf{s}^2(\mathsf{zero})) \to_1 \mathsf{clock}(\mathsf{s}^3(\mathsf{zero})) \\
&\quad \to_1 \ldots \to_1 \mathsf{clock}(\mathsf{s}^{11}(\mathsf{zero})) \to_2 \mathsf{clock}(\mathsf{s}^0(\mathsf{zero})) \to_1 \mathsf{clock}(\mathsf{s}^1(\mathsf{zero})) \to_1 \ldots\,; \\
t &= \mathsf{clock}(\mathsf{s}^0(\mathsf{zero})) \to_1 \mathsf{clock}(\mathsf{s}^1(\mathsf{zero})) \to_1 \mathsf{clock}(\mathsf{s}^2(\mathsf{zero})) \to_3 \mathsf{broken}\,; \\
t &= \mathsf{clock}(\mathsf{s}^0(\mathsf{zero})) \to_1 \mathsf{clock}(\mathsf{s}^1(\mathsf{zero})) \to_1 \mathsf{clock}(\mathsf{s}^2(\mathsf{zero})) \to_1 \mathsf{clock}(\mathsf{s}^3(\mathsf{zero})) \\
&\quad \to_1 \ldots \to_1 \mathsf{clock}(\mathsf{s}^{11}(\mathsf{zero})) \to_3 \mathsf{broken}\ 。
\end{aligned}$$

例 2.16 描述的是一个只显示时针读数（0–11）的时钟，且这个时钟随时可能损坏。函数符号 zero、s 和 add 用于表示自然数及自然数加法，之前已详细介绍，在此不再赘述；函数符号 true、false 分别表示命题为真和命题为假，not 表示逻辑“非”运算，lessthan(x, y) 表示命题“x 小于 y”，这四个函数符号用于定义重写规则的条件约束；clock(x) 表示时钟当前读数为 x；broken 表示时钟当前已损坏。$\mathcal{E}$ 的等价规则分别表示自然数加法 add 的交换律和结合律。化简模型 $\mathcal{S}$ 包含两部分，前两条规则定义了 add 的语义，即其计算方式；后三条规则定义了 lessthan 的语义，即如何计算命题“x 小于 y”的真值。$\mathcal{R}$ 的重写规则定义了系统状态发生改变的三种方式：规则 1 表示，如果时钟当前读数 x 小于 11，则下一个状态的读数为 $x+1$；规则 2 表示，如果时钟当前读数 x 不小于 11，则下一个状态的读数为 0（归零）；规则 3 表示，无论时钟当前读数是多少，下一个状态时钟可能损坏。例 2.16 的重写序列描述了从初始状态（时钟读数为 0）开始，该时钟可能发生的（其中）三种事件序列，其中箭头下标的数字表示该规范化条件重写应用的规则编号。序列 1 表示时钟读数从 0 递增至 11，然后归零，周而复始，该时钟始终保持正常运作。序列 2 表示时钟读数从 0 递增至 2，然后发生损坏。序列 3 表示时钟读数从 0 递增至 11，然后发生损坏。

从例 2.16 可以看出，时钟状态是观察者关心的行为，而自然数加法、命题真值的计算只是为描述时钟状态服务，观察者并不关心。从建模的角度来看，将所有规则平等看待，非常不利于建模人员或其它模型使用者理解模型的语义。规范化条件重写模型以一种划分的方式，将模型的主要语义和辅助语义予以区分，同时也将模型的确定性行为与非确定性行为予以区分。

规范化条件重写模型中关于化简模型 $\mathcal{S}$ 的构思来源于 Jouannaud 和 Li 提出的

范式重写模型[72]，其源头又可以追溯到 Nipkow 在高阶重写中应用的 $\beta\eta$ 范式[65]。与本文的规范化条件重写模型不同的是，范式重写要求在应用 $\mathcal{R}$ 的规则进行标准重写前，需要对被重写的项表达式进行 $\mathcal{SE}$ 模式匹配而不是 $\mathcal{E}$ 模式匹配，其中 $\mathcal{SE} \stackrel{\text{def}}{=} \mathcal{S} \cup \mathcal{S}^{-1} \cup \mathcal{E}$。也就是说，在定义 2.28 中，条件 (i) 需要替换为 $u{\downarrow_{\mathcal{S}_\varepsilon}} =_{\mathcal{SE}} s$。$\mathcal{SE}$ 模式匹配的复杂度比 $\mathcal{E}$ 模式匹配的复杂度高，且在不对 $\mathcal{S}$ 进行约束的情况下，$\mathcal{SE}$ 模式匹配极有可能是不可判定的。由于范式重写模型的提出在于给重写模型的终止性及汇合性分析提供一般化的理论框架，因此判定性问题不在其考虑范围。但作为建模工具提出的规范化条件重写模型，则不得不考虑判定性问题带来的实现问题。另一方面，范式重写由于涉及到 $\mathcal{SE}$ 等价类，因此其 $\mathcal{S}$ 规则集并不完全是关于确定性行为的描述。

规范化条件重写模型与标准重写模型及其几种相关扩展的表达能力对比总结如表 2.1 所示。

表 2.1　重写模型及其扩展的表达能力对比

模型	等价关系	条件约束	确定性行为
重写模型	✗	✗	✗
模重写模型	✓	✗	✗
类重写模型	✓	✗	✗
条件重写模型	✗	✓	✗
重写逻辑	✓	✓	✗
范式重写模型	✓	✗	☑
规范化条件重写模型	✓	✓	✓

注：“✓”表示“支持”；“☑”表示“支持有限”，“✗”表示“不支持”。

2.6　本章小结

本章对已有的重写模型及其经典扩展进行了形式化定义。基于模重写模型与条件重写模型这些经典扩展，针对实际系统中的确定性行为与非确定性行为的语义区分这一问题，本章提出了规范化重写模型及规范化条件重写模型，并对其语法及语义进行了严格的形式化定义，且将其表达能力与相关的重写模型扩展形式进行了对比。规范化条件重写模型的提出，有利于建模过程及验证过程对确定性行为与非确定性行为进行划分，为基于该模型的嵌入式系统建模验证框架奠定了基础。

第 3 章　嵌入式系统的建模与验证

形式化模型是否适合应用于形式化建模与验证，除了取决于其自身的表达能力、验证能力外，还取决于它的易用性。不同于自动机、Petri 网等图形化模型，重写模型作为一种代数模型，其抽象的形式给建模人员的使用过程带来了额外的理解成本。本章通过提出一套针对嵌入式系统特性的建模方法，来提高规范化条件重写模型作为一种建模模型的易用性。基于语义映射，该建模方法可以在工具集 Maude 中予以实现。利用该建模方法，我们对两个真实的、经过大量测试的嵌入式系统进行了行为建模，并对其模型应用了模型检测等验证技术，成功地发现系统中存在未被测试检测到的缺陷。结果表明该方法在实际应用中具有可行性。

3.1　引言

随着嵌入式系统在生产、生活中越来越广泛的应用，嵌入式系统与我们的关系越来越紧密。特别是嵌入式系统在诸如交通运输、医疗系统、金融系统等安全攸关系统中扮演的重要角色，使其安全性、可靠性、正确性保障与我们的生命财产安全息息相关。

形式化建模与验证是提高嵌入式系统可靠性与安全性的重要方法之一。通过对嵌入式系统的行为进行建模，得到可以描述该系统行为的形式化模型。然后利用支持该模型的相关工具，对模型进行仿真、测试或形式化验证，从而检查该模型的行为是否满足开发人员的预期，进而反映原系统的行为是否满足用户的需求。

嵌入式系统的其中一个特点是具有高度并发性，这主要体现在三个方面。第一，由于嵌入式系统通常由硬件与软件协同完成其功能，硬件系统所具有的并发性不可忽视。其次，由于现代嵌入式系统处理单元逐渐变得强大，使得嵌入式软件日趋复杂。软件部分不仅仅限于完成单一任务，而且还可以搭载操作系统对复杂任务进行协调调度。软件部分的多线程进一步增加了嵌入式系统的并发性。第三，大型嵌入式系统一般由多层次构成，底层组件作为上层系统的一部分，与其它嵌入式组件协作共同完成复杂的功能，这造成了系统层面的并发性。

作为自带并发语义的一种形式化模型，重写模型及其扩展在近年来作为一种系统建模模型受到越来越多的关注，被应用到硬件描述、网络协议、安全协议等领域中[60,64]。若要将其应用于嵌入式系统的实际建模验证项目，除了需要扩展现有模型提升其对软件顺序行为的描述能力外，仍需要面临两方面的挑战：

1. 建模方法：与基于状态的形式化模型（如自动机）不同，重写模型是基于抽

象的项表达式与规则的。如何将重写模型的这些抽象元素与嵌入式系统的状态及复杂行为一一对应起来，需要一套系统的建模方法进行指导。

2. 工具支持：在根据建模方法对目标嵌入式系统进行建模，得到其对应的重写模型后，从应用的角度，我们需要一套工具支持该模型的分析与验证，否则建模毫无意义。

本章基于第 2 章提出的规范化条件重写模型，提出了一套系统建模方法，并针对嵌入式系统所具有的若干特性，如层次结构、高度并发、并发行为与顺序行为并存、系统结构动态变化、实时性等，将该建模方法进一步细化。基于已有的重写工具集 Maude，我们将本章提出的建模方法在 Maude 中予以实现，并通过两个真实的应用案例，验证了该方法的可行性与实用性。

本章其余部分组织结构如下：第 3.2 小节简述基于重写模型的系统建模验证相关工作；第 3.3 小节基于规范化条件重写模型，提出嵌入式系统的建模方法；第 3.4 小节介绍本文的建模方法如何在工具集 Maude 中予以实现；第 3.5 小节与 3.6 小节应用本章提出的方法分别对机车优化控制系统和速率单调调度系统进行建模与验证；最后对本章内容进行简要小结。

3.2　相关工作

重写模型作为一种抽象的计算模型，长期以来被作为底层技术应用于定理证明器、函数式编程语言、SMT 求解器等领域中[66,68,101]。近年来，重写模型开始作为一种建模模型得到关注，并被应用于系统验证的工作中。

Bluespec 是一种基于条件重写模型的硬件描述语言[55]。由于重写本身所具有的并发性，Bluespec 十分适用于硬件的并发行为描述。相比于 VHDL、Verilog 等传统硬件描述语言，Bluespec 以一种更加抽象的方式对目标系统行为进行描述与设计，有效提高了硬件开发效率。同时，由于 Bluespec 语言在条件重写模型的基础上支持 Haskell 的类型定义，使其对复杂类型、自定义类型的支持更加丰富。利用 Bluespec 语言编写的硬件模型支持以仿真的方式进行模型验证，Singh 和 Shukla 也提出了一套方法将 Bluespec 模型转换成 PROMELA 模型[102] 并用模型检测工具 SPIN[103] 对其进行形式化验证[58]。同时，Braibant 和 Chlipala 也提出利用交互式定理证明工具 Coq 对 Bluespec 的一个子集进行形式化验证[104]。由于 Bluespec 是专门针对硬件设计的建模语言，因此它无法描述顺序执行的软件行为。

重写逻辑[59,60] 是针对建模与验证设计的一种重写模型扩展。基于重写逻辑实现的语言及工具集 Maude[61]，以及其针对实时系统的扩展 Real-Time Maude[62]，支持模型仿真、可达性验证、模型检测、交互式定理证明等多种验证手段。由于重

写模型的并发特点，Maude 主要被应用于生物信息、安全协议、通信协议等系统的建模与验证[60,64]。Maude 的应用案例为基于重写模型的建模工作确立了一套方法框架：项表达式用于建模系统状态，重写规则用于建模系统行为[105]。Maude 也被尝试用于软件和嵌入式系统的建模与验证工作中[106]。由于重写逻辑在模型层面不支持确定性行为的表达，因此软件的顺序执行行为只能通过两种方式进行描述：(1) 利用重写规则进行模拟，这将显式地增加重写模型的状态空间（可达项表达式的集合），为后端的验证过程引入困难；(2) 利用等式规则进行模拟，这可能导致重写过程不可判定，且这种建模方式使等式规则的语义不明确。

本文在上一章提出的规范化条件重写模型，在模型层次将确定性行为和非确定性行为进行区分，目的就是解决在嵌入式系统建模的过程中，如何对软件的顺序行为进行建模的问题。

3.3　基于重写模型的系统建模

将图 1.1 的形式化建模验证流程进一步细化，便得到了图 3.1 所示的基于重写模型的形式化建模验证流程。该方法要求先对目标系统的行为进行建模，得到足以描述系统行为的重写模型。将模型与目标属性输入相应的验证工具进行形式化验证，从而得到系统的验证结果。利用验证结果，开发人员可以对目标系统进行修改或调整。

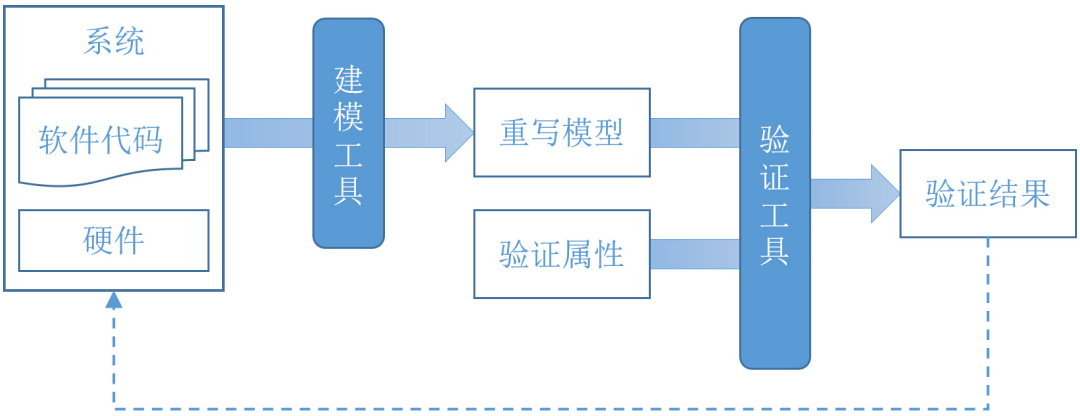


图 3.1　基于重写模型的形式化建模与验证流程

在这个流程中，建模通常是一个抽象的过程。建模过程需要针对用户所关心的系统属性，对系统的某个侧面进行抽象描述。而抽象的层次，则需要建模人员根据经验进行斟酌。如果抽象层次太低，系统模型将包含大量细节，模型行为将更贴近目标系统的行为，验证结果也更有指导意义。然而模型中细节过多可能导

致的后果是在验证过程中，验证工具无法处理如此大规模的模型验证。另一方面，如果抽象层次太高，虽然有利于后端验证过程的进行，但由于模型的行为描述太抽象，对应的验证结果可能对目标系统没有太多指导意义。

下面先介绍基于重写模型的系统建模思路，然后针对嵌入式系统所具有的一些特征，介绍如何利用重写模型对这些特征进行建模。

3.3.1　建模框架

3.3.1.1　系统状态建模

对系统进行建模，首先需要对系统某一时刻的运行状态进行建模。对于基于状态的模型（比如自动机等）来说，系统的状态可以对应为模型的状态，这比较易于理解。而对于基于项表达式和规则的重写模型来说，系统状态则需要通过项表达式来进行建模。

具体地说，在基于重写模型进行系统建模时，我们通过定义函数符号来对我们所关心的系统元素（如变量、状态等）进行“编码”。通过人为地给函数符号及其每一个参数赋予语义，来给每一个项表达式赋予语义。比如在例 2.14 中，我们规定函数符号 farmer、wolf、lamb、grass 各自表示农夫、狼、羊、草当前所处的状态，而它们的第一个参数则表示当前该对象处于河的哪一侧。我们又规定函数符号 left 和 right 分别表示河的左岸和右岸，于是项表达式 farmer(left) : : lamb(left) : : wolf(right) : : grass(left) 就可以用于表示系统的某个状态，该状态为农夫在左岸、羊在左岸、狼在右岸、草在左岸。又比如在例 2.16 中，我们规定函数符号 clock 表示一个时钟当前的状态，而它的第一个参数表示该时钟当前的读数。而我们又给 s 和 zero 赋予了自然数的语义，于是项表达式 clock(s^3(zero)) 就可以表示系统的某个状态，该状态的时钟读数为 3。

需要特别注意的是，项表达式可以对系统中用户自定义的数据类型进行描述。如定义 2.1 所示，由于项表达式利用归纳方式进行定义，与函数式编程语言（如 Haskell）中数据类型的定义方式相同，因此可以对诸如结构体、链表、二叉树等自定义数据类型进行建模。

3.3.1.2　系统行为建模

系统行为，可以理解为系统状态的改变方式。对于基于状态的模型来说，连接模型状态之间的变迁，就是模型状态的改变方式。因此对于这些模型来说，系统行为可以利用变迁进行建模。而对于重写模型，系统状态由项表达式来表示，因此作为定义项表达式之间变换关系的重写规则，可以用于建模系统的状态变迁，即

系统行为。

具体地说，重写规则的左项表示变迁发生前的系统状态（或部分状态），规则的右项表示变迁发生后的系统状态（或部分状态），而重写规则的条件约束则定义了发生该变迁所需满足的条件。需要注意的是，由于重写规则可以包含变量符号，因此一条重写规则可以描述一种状态变迁模式，即一类状态变迁，而不仅仅是某条状态变迁。比如在例 2.14 中，规则 $\mathsf{farmer}(x)::\mathsf{lamb}(x) \to \mathsf{farmer}(\mathsf{change}(x))::\mathsf{lamb}(\mathsf{change}(x))$ 描述的行为是，如果当前状态为农夫和羊在河的同侧 x，那么下一个状态，农夫和羊将会同处于河的另一侧 $\mathsf{change}(x)$。x 可以是 left（左岸），也可以是 right（右岸）。又比如在例 2.16 中，规则 $\mathsf{clock}(x) \to \mathsf{clock}(\mathsf{add}(x, \mathsf{s}(\mathsf{zero}))) \Leftarrow \mathsf{lessthan}(x, \mathsf{s}^{11}(\mathsf{zero})) = \mathsf{true}$ 表示，如果当前状态的时钟示数为 x，且满足 $x < 11$，那么下一个系统状态有可能为时钟示数显示 $x + 1$。注意这里“有可能”的意思是，由于重写的不确定性，该状态的下一个系统状态，根据规则 $\mathsf{clock}(x) \to \mathsf{broken}$，也有可能为 broken，即表示时钟损坏。

虽然重写规则本身并不对规则两侧的项表达式作任何形式上的约束，但在利用重写模型进行系统建模时，我们通常会在两侧使用“同类型”的项表达式。这指的是，规则两侧的项表达式所建模的系统状态或元素是相同类型的。比如例 2.16 的规则 $\mathsf{clock}(x) \to \mathsf{broken}$，两侧的项表达式 $\mathsf{clock}(x)$ 和 broken 所建模的都是系统中时钟的状态。如果存在规则 $\mathsf{clock}(x) \to x$，左项表示的是时钟的状态，而右项是一个自然数，则该规则从系统行为的角度来看是不合理的。

3.3.2　组件层次与并发

一个嵌入式系统可能由若干个子模块构成，系统的功能需要由各子模块之间协作共同完成；另一方面，与通用计算机不同，嵌入式系统一般用于实现特定功能，它往往作为更上层系统的一部分，与其它嵌入式系统共存。因此，要对嵌入式系统进行建模，则要求模型本身支持组件行为的可组合性及其并发的描述。

首先需要指出，在定义 2.6 中，重写对项表达式的影响被位置 p 所约束。也就是说，重写是局部进行的——当我们对某个项表达式应用重写规则进行重写时，被替换的部分不需要是整个项表达式，可以只限定于某个子项表达式。转换成系统建模的视角，这表明，通过重写规则定义的系统行为，不一定改变了整个系统状态，可以只是系统一部分的状态，即系统中某个组件的状态。

基于以上思路，多组件系统的层次结构可以利用项表达式的层次结构（定义 2.1）进行描述。假设我们用项表达式 $t_1, \ldots, t_n$ 分别对系统中 n 个组件的状态进行建模，那么可以定义一个新的 n 元函数符号 $f^{(n)}$，用项表达式 $f(t_1, \ldots, t_n)$

来对整个系统状态进行建模。

至于组件行为的组合，得益于重写的局部性，定义在子项表达式上的重写规则，可以自然地应用到整个项表达式上。也就是说，子组件的行为可以在系统中独立进行。比如说，假设我们定义了重写模型 $\mathcal{R}_i$ 对子组件 i 的行为进行建模。那么 $\mathcal{R}_i$ 的规则可以被应用于项表达式 t_i，表示子组件 i 的状态变化。当子组件 i 被置于整个系统 $f(t_1,\ldots,t_n)$ 中看待时，子组件 i 的局部行为并不受影响，$\mathcal{R}_i$ 的规则仍然可直接被应用于 $f(t_1,\ldots,t_n)$ 的子项表达式 t_i，并且不影响其它子项表达式 t_j（$j \neq i$）。正如例 2.14，我们把农民、狼、羊和草各自看作系统中的一个子组件，整个系统的状态由函数符号“::”将各子组件连接构成。规则 $\mathsf{farmer}(x) \rightarrow \mathsf{farmer}(\mathsf{change}(x))$ 实际上定义了“农民”这个子组件的行为，但它仍然可以被（局部）应用于系统状态 $\mathsf{farmer}(\mathsf{right})::\mathsf{lamb}(\mathsf{right})::\mathsf{wolf}(\mathsf{left})::\mathsf{grass}(\mathsf{left})$，产生新的系统状态 $\mathsf{farmer}(\mathsf{left})::\mathsf{lamb}(\mathsf{right})::\mathsf{wolf}(\mathsf{left})::\mathsf{grass}(\mathsf{left})$，表示农夫独自到了河对岸，而系统中其它子组件不受影响。重写技术的局部性，使组件之间的并发行为在重写模型中得到表达[59]。

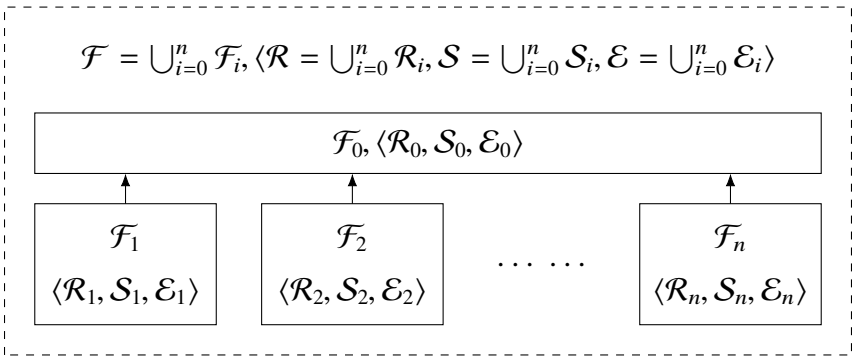


图 3.2　规范化条件重写模型的层次结构

图 3.2 描述了当系统中存在多组件时，利用规范化条件重写模型对系统进行建模的模型层次结构。假定模型 $\langle \mathcal{R}_i, \mathcal{S}_i, \mathcal{E}_i \rangle$（$1 \leq i \leq n$）是子组件 i 的行为模型。当考虑整个系统时，除了将 $\mathcal{R}_i$、$\mathcal{S}_i$、$\mathcal{E}_i$ 直接叠加（取并集），还需要定义新的词汇表 $\mathcal{F}_0$ 对系统整体状态进行建模（比如例 2.14 的函数符号“::”），以及定义新的模型 $\langle \mathcal{R}_0, \mathcal{S}_0, \mathcal{E}_0 \rangle$ 描述各组件之间的交互行为（比如同步、异步通信）。然后整个系统则可以由模型 $\langle \mathcal{R}, \mathcal{S}, \mathcal{E} \rangle$ 进行描述，其中 $\mathcal{R} = \bigcup_{i=0}^{n} \mathcal{R}_i$、$\mathcal{S} = \bigcup_{i=0}^{n} \mathcal{S}_i$、$\mathcal{E} = \bigcup_{i=0}^{n} \mathcal{E}_i$。如果系统存在更上层结构，则“递归地”进行以上建模过程。

3.3.3　组件交互

上一小节介绍了组件行为的组合建模，以及各组件局部行为之间的并发。当组件之间存在交互时，需要定义新的模型 $\langle \mathcal{R}_0, \mathcal{S}_0, \mathcal{E}_0 \rangle$ 用于描述组件之间的交互行为。下面分别从同步和异步两种方式来介绍重写模型如何对组件之间的交互行为进行建模。

3.3.3.1　同步

组件之间的同步通信，实际上是指若干组件在满足某条件时同时发生状态改变。从重写模型的角度来说，这需要重写规则同时应用于各组件对应的子项表达式。因此，描述同步行为的重写规则应该定义在描述整个系统的项表达式上，即包含 $f \in \mathcal{F}_0$ 的项表达式。该规则的左项不应该仅仅包含某个子组件对应的子项表达式，而应该包含该同步行为涉及的所有组件对应的多个子项表达式。比如在例 2.14 中，重写规则 $\mathsf{farmer}(x) :: \mathsf{wolf}(x) \to \mathsf{farmer}(\mathsf{change}(x)) :: \mathsf{wolf}(\mathsf{change}(x))$ 是一条描述同步行为的规则。规则的左项包含了组件“农民”和“狼”，只有满足农民和狼的位置相同（都为 x）这一条件，该同步行为才能发生。产生的状态变化是：农民和狼同时来到了 x 的另一侧 $\mathsf{change}(x)$。

举个例子，比如我们希望描述一个具有两个时钟的系统，我们定义二元函数符号 $f^{(2)}$ 来建模整个系统的状态，它的两个参数都是例 2.16 中定义的 clock，分别代表两个时钟当前的状态。因此 $f(\mathsf{clock}(\mathsf{zero}), \mathsf{clock}(\mathsf{s}(\mathsf{zero})))$ 表示当前系统状态为：时钟 1 的示数为 0，时钟 2 的示数为 1。如果我们希望两个时钟同步运行，那么我们不能使用例 2.16 中给单个时钟定义的重写规则，它们会产生重写序列：

$$\begin{aligned} f(\mathsf{clock}(\mathsf{s}^0(\mathsf{zero})), \mathsf{clock}(\mathsf{s}^0(\mathsf{zero}))) &\to f(\mathsf{clock}(\mathsf{s}^1(\mathsf{zero})), \mathsf{clock}(\mathsf{s}^0(\mathsf{zero}))) \\ &\to f(\mathsf{clock}(\mathsf{s}^2(\mathsf{zero})), \mathsf{clock}(\mathsf{s}^0(\mathsf{zero}))) \\ &\to f(\mathsf{clock}(\mathsf{s}^2(\mathsf{zero})), \mathsf{clock}(\mathsf{s}^1(\mathsf{zero}))) 。 \end{aligned}$$

以上序列表明两个时钟是各自并发运行的，之间没有任何关系。我们需要定义以下同步规则[①]

$$f(\mathsf{clock}(x), \mathsf{clock}(x)) \to f(\mathsf{clock}(\mathsf{add}(x, \mathsf{s}^1(\mathsf{zero}))), \mathsf{clock}(\mathsf{add}(x, \mathsf{s}^1(\mathsf{zero}))))$$

描述满足我们需求的同步行为：当系统中的两个时钟示数相同（都为 x）时，两个

① 注意这里为了方便理解，该规则省略了应满足的约束条件 $x < 11$。

时钟的示数同时加 1。于是有以下重写序列，描述了系统中两个时钟同步运行：

$$\begin{aligned} f(\mathsf{clock}(\mathsf{s}^0(\mathsf{zero})), \mathsf{clock}(\mathsf{s}^0(\mathsf{zero}))) &\rightarrow f(\mathsf{clock}(\mathsf{s}^1(\mathsf{zero})), \mathsf{clock}(\mathsf{s}^1(\mathsf{zero}))) \\ &\rightarrow f(\mathsf{clock}(\mathsf{s}^2(\mathsf{zero})), \mathsf{clock}(\mathsf{s}^2(\mathsf{zero}))) \\ &\rightarrow f(\mathsf{clock}(\mathsf{s}^3(\mathsf{zero})), \mathsf{clock}(\mathsf{s}^3(\mathsf{zero}))) 。 \end{aligned}$$

3.3.3.2　异步

异步事件触发的行为可以抽象地拆解成两个行为：组件 1 触发事件 A，事件 A 使组件 2 的状态发生变化。如果将事件 A 看作系统中的一个子组件，则这两个行为可以分别看作两个同步行为。在对异步行为进行建模时，我们采取这种方式把异步行为拆解成两个同步行为进行建模。

下面举个例子说明。假设组件 1 的状态可以从 a 变成 b，从 b 变成 c，从 c 变成 d；当它从状态 b 变成状态 c 时，同时会给系统中的组件 2 发送一个消息 m。假设组件 2 在状态 e 会等待从组件 1 发送过来的消息 m；收到消息后状态会变成 g。那么我们将组件 1 和组件 2 之间用于传输消息 m 的信道也作为系统的一个组件进行考虑。定义三元函数符号 $f^{(3)}$ 用于建模系统状态，它的第一个参数表示组件 1 的状态，第二个参数表示信道的状态，第三个参数表示组件 2 的状态。于是我们可以用以下规则来对上述异步行为进行描述：

$$\begin{aligned} a &\rightarrow_1 b \\ f(b, x, y) &\rightarrow_2 f(c, m, y) \\ c &\rightarrow_3 d \\ f(x, m, e) &\rightarrow_4 f(x, \mathsf{empty}, g) , \end{aligned}$$

其中常数 empty 表示信道中无消息。规则 2 是组件 1 和信道组件的同步行为，它表示“组件 1 的状态从 b 变成 c”这个动作必须和“信道中有消息 m 出现”这个动作同时发生，且注意该同步行为不涉及组件 2——其状态 y 不发生改变。而规则 4 则是信道组件和组件 2 的同步行为，它表示“信道中的消息 m 消失”这个动作必须与“组件 2 的状态从 e 变成 g”这个动作同时发生。规则 2 和规则 4 是典型的异步行为建模规则。由上述规则可以产生如下重写序列：

$$f(a, \mathsf{empty}, e) \rightarrow f(b, \mathsf{empty}, e) \rightarrow f(c, m, e) \rightarrow f(d, m, e) \rightarrow f(d, \mathsf{empty}, g) 。$$

该序列表明，组件 1 向组件 2 发出消息 m 后，不需等待组件 2 的响应，即可继续运行（从状态 c 变成状态 d）。需要注意的是，状态 b 与 g 可以进一步细化为与消息 m 有关的项表达式，如 $b = b'(m)$、$g = g'(m)$，表示组件 1 向组件 2 传递信息。

3.3.4　软硬件行为

嵌入式系统通常由软硬件混合构成，而软件与硬件的行为差异在于，硬件行为通常具有并发性，而软件代码通常以顺序方式执行。从建模的角度对组件的顺序行为与并行行为进行考虑，它们可以进一步被抽象成两类行为——确定性行为与不确定性行为，分别对应于规范化条件重写模型 $\mathcal{R}_{\mathcal{SE}}$ 的规则集 $\mathcal{S}$ 与 $\mathcal{R}$。

对组件的局部行为，若是顺序执行的（例如不涉及共享变量访问的软件代码），则将其建模为局部确定性行为，即建模为 $\mathcal{S}_i$ 的重写规则；若该行为具有不确定性（例如硬件模块随时可能发生物理失效），则将其建模为局部不确定性行为，即通过 $\mathcal{R}_i$ 的重写规则进行建模。对组件间的交互行为，考虑到它可能对系统中其它行为产生影响（例如对共享变量进行写操作可能影响其它线程对该变量的读操作），因此将其建模为全局的不确定性行为，即描述为 $\mathcal{R}_0$ 的重写规则。

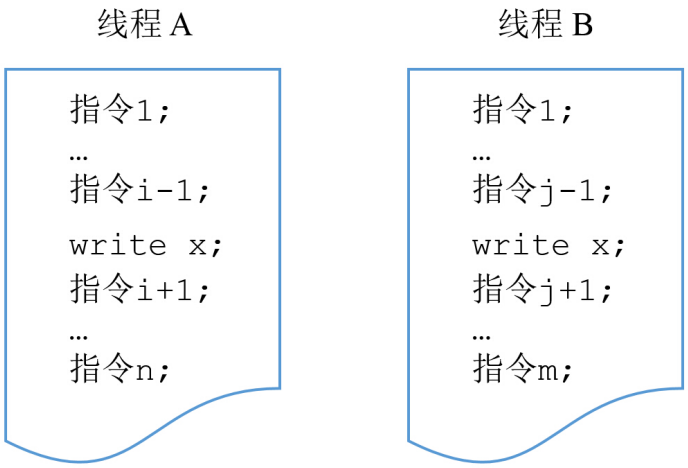


图 3.3　两个线程的并发

如图 3.3 的例子，假设变量 x 是全局变量，线程 A 的指令 i 和线程 B 的指令 j 都对 x 进行写操作，因此这两条指令被调度运行的先后次序将对整个系统的运行结果产生影响，它们应当被建模成 $\mathcal{R}_0$ 中的全局不确定性规则。假设线程 A 的其它指令以及线程 B 的其它指令都与变量 x 无关，虽然这些指令相互之间并发运行，但实际上系统对这些指令的调度并不会影响运行结果，因此它们可被建模成 $\mathcal{S}_i$ 的局部确定性规则。

从模型组合的角度来看，不同组件的局部确定性行为 $\mathcal{S}_i$ 作用于项表达式的不同位置，因此这些行为组合得到的系统行为模型 $\mathcal{S}$ 具有终止性与合流性[107]，满足规范化条件重写模型 $\mathcal{R}_{\mathcal{SE}}$ 的要求。

根据规范化重写的定义（定义 2.25），$\mathcal{S}$ 规则的应用是附属于 $\mathcal{R}$ 规则的，以上建模策略让系统中行为的重要程度在规范化重写模型中有所体现。另一方面，根据定义 2.25，应用 $\mathcal{R}$ 规则前后都需要对整个项表达式应用 $\mathcal{S}$ 规则进行规范化，这就保证了规范化重写序列与系统状态变迁路径的对应性。同时，在这一建模策略下，由于 $\mathcal{S}$ 规则对应的行为被“嵌入”到 $\mathcal{R}$ 规则对应的行为中，相当于若干行为被抽象成单一行为，这使得系统的状态空间数得以减少[①]，为后端的验证过程提供了便利。

3.3.5　组件的动态结构

在嵌入式系统中，存在组件结构的动态行为。比如在事件响应系统中，接收到一个事件请求后，系统可能创建一个新线程对该事件进行处理；又比如在无线传感网络中，因为主观连接需求或者客观的通信信号问题，网络中的节点拓扑通常会产生变化。这些行为都涉及到组件结构的改变。这就对建模模型提出了两个问题：如何对这种组件结构变化进行建模？如何对新增的组件行为进行建模？

首先是对组件结构变化的建模方式。由于项表达式描述的不仅仅是系统中某些变量的状态，而是整个系统的状态，其中包括系统的结构信息。在第 3.3.2 小节中提到，系统组件的数量由函数符号的元数进行描述。如果系统中组件数量可变，那么我们需要函数符号的元数可变——这违背了词汇表 $\mathcal{F}$ 的定义。然而，得益于规范化重写模型丰富的表达能力，我们可以利用等价模型 $\mathcal{E}$ 来模拟元数可变的函数符号。实际上，解决思路早已在例 2.14 中给出——定义二元（中缀）函数符号“::”。如果它具有结合律，即以下等价规则属于 $\mathcal{E}$，

$$(x :: y) :: z \quad \sim \quad x :: (y :: z) \ ,$$

那么由“::”构成的项表达式中的括号可以省略。它定义了一个列表结构，列表中的元素数量可变。例如 $a :: b$、$a :: b :: c :: d$、$a :: b :: c :: d :: e$ 都是列表。如果函数符号“::”同时还具有交换律，即以下等价规则属于 $\mathcal{E}$，

$$x :: y \quad \sim \quad y :: x \ ,$$

① 这意味着被抽象的中间状态必须与验证属性无关。

那么由“::”构成的项表达式中，不仅括号可以省略，而且元素之间的顺序变得无关重要。它定义了一个多重集合结构，多重集合中的元素数量可变，且元素无序。例如 $a::a::c::d$ 和 $a::d::c::a$ 都是多重集合，且 $(a::a::c::d) =_{\mathcal{E}} (a::d::c::a)$。下面假设“::”同时具有交换律和结合律。

基于这种结构可变的模型，改变结构的行为可以被建模成对应的改变项表达式结构的重写规则。例如要描述以下行为：线程 f 在状态 a 创建一个新线程 g，g 的初始状态为 a'，而线程 f 在创建 g 以后进入状态 b。那么我们可以用以下重写规则：

$$f(a) \quad \rightarrow \quad f(b)::g(a')$$

来描述线程 g 的创建过程。又比如在例 2.14 中，我们可以用以下规则：

$$\mathsf{wolf}(x)::\mathsf{lamb}(x) \quad \rightarrow \quad \mathsf{wolf}(x)$$

描述狼把在同侧的羊吃掉的行为。这也是一种系统组件结构发生变化的行为。

在对“产生新组件”这一行为进行建模后，还需要对新组件本身的行为进行建模。由于新组件的行为是可预知的（比如处理某类事件的方式），因此描述新组件行为的重写规则只需如其它行为一样，定义在规则集 $\mathcal{S}$ 或 $\mathcal{R}$ 中即可。

在这里我们可以看到基于重写模型建模的一大优点：基于项表达式的状态建模，可以描述系统结构的变化，使建模人员不需要对动态产生的新组件数量上限进行假设。

3.3.6　实时性

许多嵌入式系统的正确性不仅要求满足逻辑正确性，还需要满足某些时间约束，这类系统称之为*实时系统*。对于实时系统的建模，要求模型中支持对“时间”的建模。针对重写模型，我们在系统的项表达式中，增加一个用于描述时间的子项表达式，例如项表达式

$$f(t_1, \ldots, t_n, \mathit{time})$$

中，$f^{(n+1)}$ 的最后一个参数。而位于 time 位置的子项表达式可以采用自然数集（如例 2.1）表示离散的时间，也可以采用非负实数集[108] 表示连续的时间，这取决于建模人员对模型精确度的要求，也取决于验证工具的验证能力。

需要注意的是，由于我们将“时间”作为组件在模型中进行描述，因此系统中所有需要耗时的行为都将被建模成同步行为，需要与“时间”组件进行同步。例如下列规则：

$$f(a,\ x_{time}) \quad \rightarrow \quad f(b,\ x_{time}+1)$$

描述了系统从状态 a 变成状态 b 的行为，这一行为需要耗时 1 个时间单位。

3.4　支持工具集

本小节将基于语义映射的方式，将上述建模框架在工具集 Maude 中予以实现。

3.4.1　Maude

Maude[61] 是一种基于重写逻辑[60] 的语言和工具，它能支持形式化建模和对模型的分析验证。

3.4.1.1　建模

Maude 将系统建模成为模块（module）。一个模块相当于一个重写逻辑模型（rewrite theory）$\mathcal{R}^{\mathcal{L}} = \langle \Sigma, A, E, R \rangle$，其中：

- Σ 是一个带类型的词汇表。它包括函数符号、类型[①]（sort）和子类型（subsort）的声明。函数符号允许使用中缀记法、前缀记法、后缀记法或混合记法，利用下划线“_”表示参数的位置。项表达式的定义与重写模型相同。
- $\langle \Sigma, E \cup A \rangle$ 是一个带成员关系的等词逻辑理论[109]（membership equational logic theory）。其中，E 是条件等式（conditional equation）和成员关系规则的集合，A 是一个性质集合，比如交换律、结合律等。
- R 是一个带标签的条件重写规则集合。它描述了系统的单步状态变迁。每一条规则的形式为 $[l] : t \rightarrow t'$ **if** $\bigwedge_{j=1}^{n} cond_j$，其中 $cond_j$ 是形如 $u_j = v_j$ 的约束条件，l 是该规则的标签，t, t', u_j, v_j 都是项表达式。该规则的语义与条件重写规则（定义 2.21）的语义相同。

Maude 还支持面向对象风格的建模。类声明

$$\texttt{class}\ \ C\ \ |\ \ att_1 : s_1, \ldots, att_n : s_n$$

① 注意这里应指类别，与“类型”有所区别。为了便于读者理解，本文称之为“类型”。

定义了一个新类，类名为 C，它包含类属性 $att_1, \ldots, att_n$，它们分别具有类型 $s_1, \ldots, s_n$。类 C 的一个对象（或称“实例”）表示为项表达式

$$< O : C \mid att_1 : val_1, \ldots, att_n : val_n >。$$

其中，O 是该对象的标识（identifier），具有类型 `Oid`；val_i 是属性 att_i 的值。所有类的对象都具有类型 `Object`。重写规则可以针对某个类的项表达式进行定义。子类（subclass）可以继承其父类的所有类属性以及相关的重写规则。

Real-Time Maude[62] 是 Maude 的一个扩展，它同时给 Maude 语言和工具集增加了新功能。它主要的目的是让 Maude 可以支持实时系统的建模和分析验证。在 Maude 的基础上，Real-Time Maude 内置了一个名为“`Time`”的描述时间的模块。在 Maude 模块的基础上，Real-Time Maude 在集合 R 中加入了带标签的单元计时规则（labeled tick rule）。单元计时规则具有以下形式

$$[l] : \{s\} \rightarrow \{s'\} \textbf{ in time } r \textbf{ if } cond \ ,$$

用于描述系统的时间行为。它表示当条件 $cond$ 满足时，系统从状态 s 变成目标状态 s'，（全局）时间往前推移 r 个时间单位。为了区别于单元计时规则，R 中原来的不涉及时间的重写规则，称作瞬时规则（instantaneous rule），应用该规则时（全局）时间保持不变。

需要注意的是，Maude 面向对象风格的模型以及其扩展 Real-Time Maude 都只是一种语法封装，它们的底层仍基于重写逻辑模型 $\mathcal{R}^{\mathcal{L}}$ 和重写逻辑。

3.4.1.2　模型分析及形式化验证

给定一个重写逻辑模型，Maude 和 Real-Time Maude 给用户提供了许多配套工具对模型进行分析。例如，`rewrite` 指令可以符号化地、抽象地执行目标模型，在模型的层次上对系统进行仿真；给定一个初始状态（用项表达式表示），`search` 指令可以搜索所有可达的状态，从中判断是否存在满足指定性质的状态；Maude 配套的交互式定理证明器[63]（Inductive Theorem Prover，ITP）可以通过交互式的方法，用于证明模型具有的等词逻辑性质。

在本文中，我们更关心的是 Real-Time Maude 配套的形式化验证工具——*LTL* 模型检测器[110]（LTL model checker）。它可以检查是否模型中的任意一条行为路径都满足给定的时序逻辑公式。状态命题（state proposition）被定义为具有类型 `Prop`

的项表达式。它们的语义由集合 E 中以下形式的条件等式进行规定：

$$\texttt{ceq}\ \ statePattern\ \ \texttt{|=}\ \ prop\ \ \texttt{=}\ \ b\ \ \texttt{if}\ \ cond\ \ ,$$

其中，b 是具有类型 `Bool` 的项表达式。该条件等式表示，当 *cond* 成立且当前状态的项表达式是 *statePattern* 的实例时，*prop* 的值等于 b。所有这种形式的条件等式共同定义了命题 *prop* 成立的条件：命题 *prop* 在状态 s 成立，当且仅当 s 满足 s `|=` *prop* 等于 `true`。时序逻辑公式[111]（temporal logic formula）由状态命题和时序逻辑操作符（operator）构成。时序逻辑操作符包括但不限于 `~`（非）、`\/`（或）、`[]`（“总是”）、`<>`（“最终”）、`U`（“直到”）。Real-Time Maude 支持时控的（timed）和非时控的（untimed）LTL 模型检测。本文的案例将会用到非时控的 LTL 模型检测命令

$$\texttt{(mc}\ \ s\ \ \texttt{|=u}\ \ \Phi\ \ \texttt{.)}$$

该命令检查的是，给定初始状态 s，从 s 开始的所有行为序列是否在所有时刻都满足时序逻辑公式 Φ。

3.4.2　建模框架实现

如果将重写逻辑模型 $\mathcal{R}^{\mathcal{L}} = \langle \Sigma, A, E, R \rangle$ 的 Σ、A、E、R 分别与规范化条件重写模型 $\mathcal{R}_{\mathcal{SE}} = \langle \mathcal{R}, \mathcal{S}, \mathcal{E} \rangle$ 的 $\mathcal{F}$、$\mathcal{E}$、$\mathcal{S}$、$\mathcal{R}$ 对应起来，两者存在许多相似之处，然而它们的主要区别在于：

(i) 重写逻辑模型 $\mathcal{R}^{\mathcal{L}}$ 的性质集合 A 只支持交换律、结合律、单位元性质（identity）和幂等性（idempotency），而规范化条件重写模型 $\mathcal{R}_{\mathcal{SE}}$ 的等价模型 $\mathcal{E}$ 支持一般化的等式规则，覆盖了 A 的支持范围；

(ii) 重写逻辑模型 $\mathcal{R}^{\mathcal{L}}$ 的重写过程基于类重写[71]，即重写规则应用的对象是项表达式关于 E 的等价类，而规范化条件重写（定义 2.28）基于规范化过程，即重写规则应用的对象是项表达式关于 $\mathcal{S}$ 的范式；

(iii) 重写逻辑模型 $\mathcal{R}^{\mathcal{L}}$ 带类型信息，而规范化条件重写模型 $\mathcal{R}_{\mathcal{SE}}$ 不带类型信息。

区别 (ii) 是两者在理论模型上的区别。然而，Clavel 等人在文献 [61] 中指出，Maude 在具体实现时，为了保证工具效率，采取了与规范化重写相似的重写策略。

因此，本文提出算法 1，将（部分）规范化条件重写模型转换为重写逻辑模型。转换的关键点在于：重写逻辑模型的集合 A 所支持的（等式）性质不及规范化条件重写模型的集合 $\mathcal{E}$ 一般化，因此对于 $\mathcal{E}$ 中不被支持的等式规则，需要将其有向

Input: 规范化条件重写模型四元组 $\mathcal{F}, \mathcal{E}, \mathcal{S}, \mathcal{R}$

Output: 重写逻辑模型四元组 Σ, A, E, R

$\Sigma = \mathcal{F}$;

$R = \mathcal{R}$;

$A = \varnothing$;

foreach $e \in \mathcal{E}$ **do**

  if e 表示交换律、结合律、单位元性质或幂等性 **then**

    $A = A \cup \{e\}$;

  end

end

$\mathcal{E}' = \mathcal{E} \setminus A$;

bool $exist = false$;

// $directed(\mathcal{E}')$ 计算 $\mathcal{E}'$ 中等式有向化产生的所有可能的重写规则集

foreach $\vec{\mathcal{E}}' \in directed(\mathcal{E}')$ **do**

  if 模重写模型 $(\vec{\mathcal{E}}' \cup \mathcal{S})_A$ 是终止且合流的 **then**

    $exist = true$;

    break ;

  end

end

if $exist = true$ **then**

  $E = \vec{\mathcal{E}}' \cup \mathcal{S}$;

else

  转换失败;

end

算法 **1:** 规范化条件重写模型转换成重写逻辑模型

化，转换成重写规则放入集合 E。所以该算法转换成功与否的关键，就在于有向化后的规则集 $\vec{\mathcal{E}}'$ 能否满足重写逻辑对集合 E 的要求，即必须是终止且合流的。

基于算法 1，我们将第 3.3 小节提出的建模方法在 Maude 中予以实现，并通过两个真实应用案例（第 3.5 小节和第 3.6 小节），验证它的可行性。

3.5　应用案例：铁路机车节能优化控制系统

铁路机车节能优化控制系统[112]（以下简称“优化控制系统”）是由清华大学软件学院智能交通实验室与中车信息技术有限公司合作开发的一套铁路机车智能巡航控制与节能产品，用以辅助司机通过自动控制机车来最大限度降低机车的能耗。该系统对机车进行控制的过程中涉及到自动控制与司机手动控制状态的转换。本小节基于第 3.3 小节提出的建模方法，利用 Maude 工具对该系统的状态转换逻辑进行建模和形式化验证。通过建模过程和验证方法的应用，我们定位了系统中的若干缺陷，并与开发人员进行了确认，帮助开发人员进行修复。

3.5.1　背景介绍

图 3.4　铁路机车节能优化控制系统

优化控制系统的组成及其涉及到的周边部件如图 3.4 所示。LKJ2000 负责向优化控制系统提供铁路线路信息以及实时信息（如信号灯、是否限速等）。在手动控制状态下，机车驾驶员通过司控器进行机车的档位控制。继电器组和机车控制系统对机车进行物理控制。智能显示屏显示机车的实时状态。而优化控制系统本身主要由三部分构成。安全信息平台负责处理从 LKJ2000 接收的数据，向优化控制器提供必要的数据。优化控制器是整个系统的核心，负责列车控制逻辑和控制策略的计算。交互单元则是驾驶员与优化控制器的交互媒介。

机车在运行过程中分为自动控制和手动控制两种状态。机车自动控制时，优

化控制器基于离线计算的控制策略，结合在线根据机车状态、线路信息和实时信息计算得到的调整策略，对机车进行控制。当机车处于手动控制状态时，由驾驶员通过司控器对机车进行控制。自动控制状态和手动控制状态之间的转换，可以由驾驶员主动发起，也可能由于机车的状态或铁路的实时信息满足一定条件，从而触发自动和手动控制状态的转换。

机车控制状态的转换由优化控制器负责完成。优化控制器综合从安全信息平台接收的数据、从交互单元获取的操作数据、以及司控器的档位数据，根据业务逻辑计算当前的控制策略（包括是否自动控制、控制档位等）。控制状态转换的逻辑基于事件处理机制。也就是说，优化控制器把安全信息平台向其发送的数据到达看作一个事件，把驾驶员通过交互单元进行的操作也看作一个事件，驾驶员通过司控器调整档位也被当作一个事件。优化控制器对这些外部事件进行处理，产生一个事件（数据帧）发送给继电器组，通知继电器组当前应采取的控制状态及控制档位。图 3.5 是一个从自动控制状态转换成手动控制状态的流程示例。

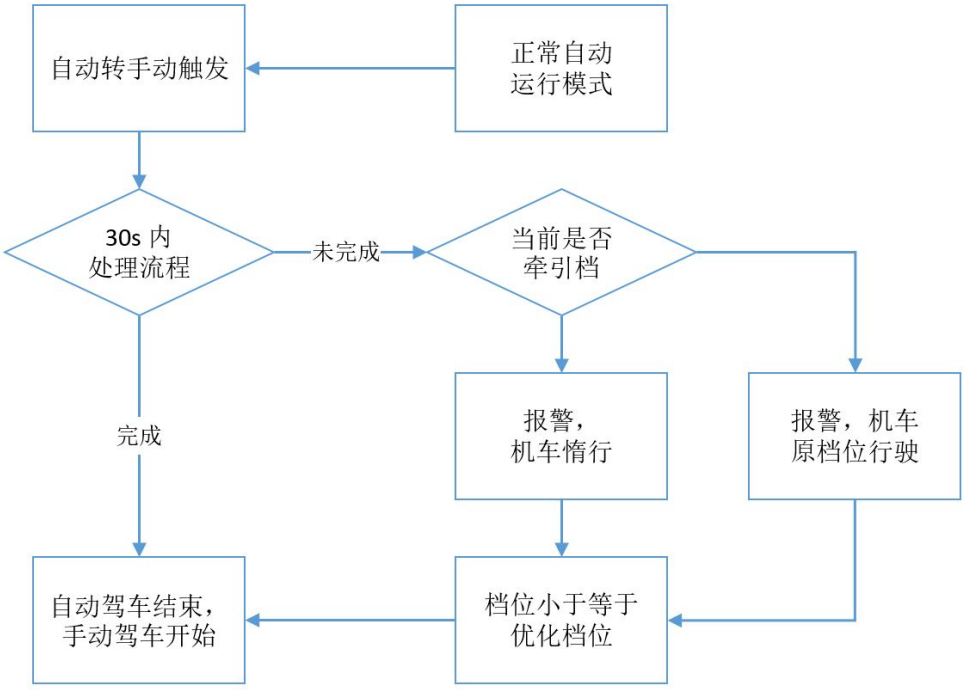


图 3.5　自动控制到手动控制的转换流程示例

优化控制器内部由若干独立模块组成，各模块拥有独立的处理器，且以多线程的方式工作。模块之间的交互也通过事件处理机制完成。因此整个优化控制器，乃至整个优化控制系统，是一个高度并发的分布式系统。系统是否对每一个事件都进行了正确的响应，事件处理的正确性是否受到并发影响，是开发人员及用户

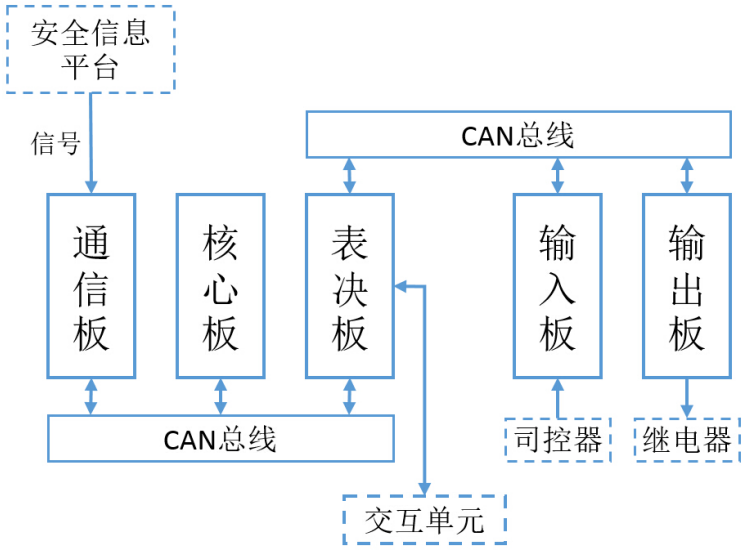


图 3.6　优化控制器内部结构

迫切关心的问题。本小节主要关心涉及机车控制状态转换的事件处理。

3.5.2　系统描述

优化控制器的内部结构如图 3.6 所示，主要由通信板、核心板、表决板、输入输出板以及连接各板卡的总线组成。输入板的功能是将司控器的档位信息发送给表决板。输出板的功能是将表决板的计算结果（控制状态和档位信息）发送到继电器。通信板从安全信息平台接收线路信息以及实时信息，经处理后转发到核心板。核心板接收来自通信板的线路信息和实时信息，根据优化算法计算优化控制策略。然后核心板将优化计算结果发送至表决板。表决板综合考虑来自核心板的优化控制策略、通过交互单元获得的用户操作信息、以及来自输入板的档位信息，根据业务逻辑产生机车的控制策略（自动或手动控制、档位信息等），发送到输出板。

每一块板卡都维护了一个线程池，板卡的每一个端口都有一个单独的线程负责监听。当有新消息帧到来时，负责监听的线程根据该消息帧的类型，创建一个新线程对其进行处理，然后该监听线程继续监听端口消息。被创建的新线程根据自身业务逻辑对该消息帧进行相应的操作，对其计算、转发、触发一个业务流程、或再创建另一个线程等，执行完毕后线程释放。板卡的业务逻辑由多个线程协作共同完成。线程的调度由板上的操作系统完成。

3.5.3　系统建模

我们选择将优化控制系统中的通信板、核心板、表决板、输入输出板以及连接各板卡之间的总线进行建模，并且为安全信息平台和交互单元建立简单的模型，描述它们与优化控制系统之间的交互。

对该系统建模的难点在于：

1. 对复杂的数据结构及其存储进行建模。系统由 C 语言实现，代码量巨大，涉及的数据类型复杂，频繁使用各种结构体类型。这要求对数据类型进行建模，且对相应的存储结构（内存）也进行建模。
2. 对板卡间的通信进行建模。
3. 对线程调度进行建模。线程调度由板载操作系统负责，在建模过程中需要考虑线程调度的影响。
4. 对线程以及线程的创建释放进行建模。线程的行为是该系统进行事件处理及实现业务逻辑的关键行为，需要对其进行较准确的刻画。

下面就以上关键点，结合第 3.3 小节提出的建模方法与第 3.4.2 小节定义的规范化条件重写模型与 Maude 模型之间的语义映射，对该系统的建模过程进行介绍。

3.5.3.1　复杂数据类型及内存

在第 3.3.1.1 小节已经提到，得益于项表达式的归纳结构，我们可以利用它定义非常复杂的数据类型。同时又得益于 Maude 语言提供的灵活语法，复杂数据结构得以直观地表示。例如我们用 Maude 类型 `RtCore` 对 C 结构体类型 RtCore 进行建模，`RtCore` 的定义如下：

```
sort RtCore .
op `(rt-gear:_`,rt-enable-status:_`)
     : Nat Nat -> RtCore [ctor] .
```

第 1 行用 `sort` 关键字声明了一种新类型。其余两行用 `op` 关键字声明了一个新的函数符号 `(rt-gear:_,rt-enable-status:_)`，它的两个参数分别为 `Nat`（自然数）类型和 `Nat` 类型，返回类型为 `RtCore`。两个下划线“_”的位置为该函数符号的两个参数位置。于是项表达式 `(rt-gear:3,rt-enable-status:0)` 表示一个类型为 `RtCore` 的结构体数据，它的 rt-gear 域为 3，rt-enable-status 域为 0。

结构体类型也可以进行嵌套，比如以下表示结构体类型 RtVote 的类型 `RtVote`：

```
sort RtVote .
op `(rt-core:_`,rt-control-status:_`)
     : RtCore ControlStatus -> RtVote [ctor] .
```

它的第一个参数类型为 RtCore，是一个表示结构体的项表达式类型。

除了结构体，其它复合类型如第 3.3.5 小节提到的列表和多重集，都可以在 Maude 中进行建模。

针对内存，我们可以用类型 Pair 来对内存单元进行建模：

```
op `(_->_`) : Variable Value -> Pair [ctor] .
```

即 Pair 类型的表达式由一个 Variable 类型的“变量”表达式与一个 Value 类型的“值”表达式组成。Value 类型包括 Maude 内置的自然数 Nat 类型、布尔 Bool 类型以及我们自定义的各种数据结构类型。而内存则是由 Pair 类型表达式组成的一个多重集，由连接符“,”进行连接。例如项表达式

```
('x -> 3), ('y -> true)
```

是一个描述内存状态的项表达式，表示变量 x 值为 3，变量 y 值为 true。

3.5.3.2　板卡通信

为了简化模型，我们假设板卡之间的通信是定向的，即发送时需要指定目标板卡的目标端口。于是我们使用 Msg 类型来建模这样一条单向发送通道：

```
op `[==>_:_|_`] : Oid PortType MaybeFrame -> Msg .
```

其中，Oid 类型的参数指定目标板卡，PortType 类型的参数指定目标端口，第三个参数是发送的消息帧。而且我们假定这样的发送通道缓冲区大小为 1，每次只能发送一个消息帧。通道被占时会对发送产生阻塞。例如以下表达式

```
[==> VOTE-DES : PORT-CANINOUT
  | some (gear: 3 ,auto-or-manual: 1) ]
```

表示通道中包含向 VOTE-DES（表决板）的 PORT-CANINOUT 端口发送的消息帧，该帧包含档位数据及控制状态数据。

消息的发送与接收行为，则分别可以建模成该通道与发送方以及接收方的同步行为，如第 3.3.3.1 小节及第 3.3.3.2 小节所述。

3.5.3.3　线程调度

由于板卡上线程的调度由板载操作系统负责。为了简化模型，我们假设每块板卡上的系统对其线程进行轮转调度。于是我们将线程池建模成一个线程队列，调度时每次选取队首线程，执行该线程的一个行为（对应实际程序中的若干条指令）后，将该线程放入队列末端，调度模块开始新一轮的调度。

例如以下条件重写规则（crl）：

```
crl [core-socket-rcv-n-read-some] :
    < O : Core | cpu : T, pool : P >
    [==> O : S | some F ]
  => < O : Core | cpu : ideal, pool : enqueue(T', P) >
     [==> O : S | none ]
  if < socket-rcv(S) : Thread | st : read > := T
     /\ T' := < socket-rcv(S) : Thread
                | st : core-frame-parse(F) > .
```

符号 => 用于分隔重写规则的左项和右项，关键字 if 标识该规则的约束条件。该重写规则描述了核心板上负责监听端口 S 的线程 socket-rcv(S) 对通信通道中的消息帧 F 进行读取的行为。该规则左项表示板卡 Core 的 CPU 当前正在执行线程 T，而 T 的状态为 read，即正在试图读取信道中的消息帧。此时信道中恰好包含消息帧 F。应用该规则后，线程 T 的状态变成 core-frame-parse(F)，即正在解析消息帧 F，且线程 T 被放入线程队列 P 的末端，板卡 Core 的 CPU 此时为空闲状态（ideal）。由于该行为是涉及核心板与通信信道的同步行为，因此根据第 3.3.4 小节所述，应该将其建模为 $\mathcal{R}$ 规则，根据算法 1 即 R 中的重写规则。

若板卡的 CPU 处于空闲状态，调度模块会马上执行线程队列的队首线程，如以下等式所示：

```
eq < O : Core | cpu : ideal, pool : (T ; P) >
   = < O : Core | cpu : T, pool : P > .
```

注意由于该行为是核心板的局部顺序行为，因此根据第 3.3.4 小节所述，应该将其建模为 $\mathcal{S}$ 规则，而根据算法 1，对应于 E 中的等式。

3.5.3.4　线程创建与释放

线程的创建与释放涉及到系统中组件数量的变化。根据第 3.3.5 小节提出的方法，我们利用元素数量可变的结构来对线程进行建模。从上面例子可以看到，在

我们的模型中，线程被建模成队列里的一个元素。因此，线程的创建与释放，体现在线程队列的长度变化。

例如以下规则：

```
crl [core-thread-add] :
    < O : Core | cpu : T, pool : P >
  => < O : Core | cpu : ideal,
                  pool : enqueue(T', enqueue(NEW, P)) >
   if < socket-rcv(S) : Thread
        | st : thread-add(handle, C) > := T
     /\ T' := < socket-rcv(S) : Thread | st : read >
     /\ NEW := < handle : Thread | st : handle(C) > .
```

该规则描述了板卡 Core 上的线程 socket-rcv(S) 创建了一个名为 handle 的新线程 NEW 去处理命令消息 C 的行为。该新线程创建以后被放入线程队列 P 的队尾。类似地，由于该行为属于同步行为，因此用 $\boldsymbol{R}$ 中的规则进行建模。

线程释放行为对应的建模规则与创建行为类似。

3.5.4　模型验证

由于优化控制系统在列车运行过程中会不断地从安全信息平台获取到铁路的线路信息和实时信息，会从交互单元不定期地获取到驾驶员的操作指令，也会从司控器不定期地获取到档位的变化信息，因此它是典型的反应式系统。系统的外界输入空间是无限的，使我们无法对整个模型应用模型检测方法进行形式化验证。针对本案例，我们采用测试和形式化验证结合的方法对其进行分析。

我们采取的具体做法是，先针对某个待验证性质（比如“机车最终会处于手动控制状态或者惰行状态”），随机生成对应的外界输入序列。在模型中，给定生成的输入序列作为初始状态，应用模型检测方法验证该模型是否满足预期的性质。比如给定初始状态 init，以下模型检测命令验证列车是否最终会处于手动控制状态（由命题 manual 定义）或惰行状态（由命题 slide 定义）：

```
(mc init |=u <> (([] manual) \/ ([] slide)) .)
```

若该命令返回 true，则表示模型在该初始状态下满足验证性质；否则命令将返回一条反例路径，展示违反性质的系统运行轨迹。根据反例路径，开发人员即可对模型进行调整，或对系统进行修复。

虽然这种分析方法不是完备的，但针对这种高度并发的分布式系统，该方法比纯粹的测试方法更有效。虽然该系统已经经过测试人员的大量测试，但我们利用这种测试与验证结合的分析方法，总共发现系统中存在的 3 个缺陷，其中 2 个缺陷属于系统对事件处理机制的缺陷，另外 1 个缺陷是系统进行控制状态转换的逻辑缺陷。3 个缺陷都已经与开发人员进行确认，并由其在系统中进行修复。结果表明，我们的建模分析方法能有效提高系统的可靠性。

3.5.5　案例小结

在本案例应用中，我们利用 Maude 对一个真实的机车优化控制系统进行了建模。在模型中，我们描述了系统软件中涉及的复杂数据结构、子系统之间的通信交互、线程的创建释放及调度等细节。利用测试与验证结合的模型分析方法，我们发现系统中的 3 个缺陷，并得到开发人员的确认。该系统目前运行稳定，并在沈阳铁路局通过了实车运用考核。

3.6　应用案例：速率单调调度系统

速率单调调度[113]（Rate-Monotonic Scheduling，RMS）是在工业应用中最重要的实时调度算法之一。针对 RMS，特别是针对 RMS 的可调度性（schedulability），学术界和工业界对其进行了深入的研究并取得大量成果。然而，针对 RMS 的理论研究仅仅停留在算法的抽象层面上，当面对一个实际存在的 RMS 系统实现时，这些理论研究成果由于不包含对实现细节的考虑而无法直接应用到 RMS 系统的分析中。另一方面，除了可调度性以外，RMS 系统实现的正确性（correctness）也是开发人员及用户十分关心的问题。

本小节利用 Real-Time Maude 工具对一个真实的 RMS 系统实现进行建模和验证。在模型中，我们考虑了系统开销以及硬件平台的部分细节。我们对该 RMS 系统的可调度性和实现正确性进行了形式化验证，并证明了我们的验证方法具有可靠性（soundness）和完备性（completeness）。

3.6.1　背景介绍

周期性任务调度是实时工业系统中最重要的问题之一。在某个调度算法的调度下，如果一个周期性任务集合中的每个任务实例的运行都不会超过其时限（deadline），则称该任务集根据该调度算法是可调度的（schedulable）。RMS 是针对抢占式硬实时系统的一种静态优先级调度算法。它由 Liu 和 Layland 在 1973 年提出[113]。它的核心思想是，任务实例的优先级应由其对应的任务周期确定，任务周期越小，

其任务实例拥有的优先级越高。Liu 和 Layland 在 [113] 中证明了 RMS 算法是最优的静态优先级调度算法。“最优”的意思是，给定一个周期性任务集合，如果存在某种静态优先级调度算法 $\mathcal{A}$，使得该任务集根据算法 $\mathcal{A}$ 是可调度的，那么该任务集根据 RMS 算法肯定也是可调度的。除了被证明最优，由于 RMS 算法十分易于实现，因而被广泛应用于安全攸关的实时环境中，如高速列车、航空航天器等。

Liu 和 Layland 证明了一个任务数为 n 的周期性任务集根据 RMS 算法可调度的充分条件是：$\Sigma_{i=1}^{n} C_i/T_i \leq n(2^{1/n} - 1)$，其中 C_i 和 T_i 分别是任务 τ_i 的运行时间和运行周期[113]。对 RMS 的研究主要分为两个方向。一是试图放宽 RMS 算法模型的约束条件，使其适用于更多的系统场景。比如文献 [114–117] 允许被调度对象任务集合中存在非周期性任务；文献 [118,119] 将 RMS 扩展成为时限单调调度（deadline-monotonic scheduling）；文献 [120] 允许各任务之间共享资源；文献 [121–124] 将 RMS 扩展到多处理器平台；而文献 [125–127] 则致力于提升系统的容错性。对 RMS 研究的另一个方向，则是试图获得更好的判定条件，对 RMS 及其各种扩展的可调度性进行判定[122,124,128–130]。由此可以看出，RMS 算法的重要性毋庸置疑。

当 RMS 被应用于实际系统开发时，特别是当该系统是安全攸关的系统时，保证 RMS 实现的正确性远比保证 RMS 算法的正确性要重要得多。当分析的对象是 RMS 算法的某个具体实现时，关于 RMS 的理论分析结果可能不再适用。即使系统从理论上满足可调度的充分条件，然而在实际运行时，由于系统本身存在系统开销，或者中断屏蔽机制使得中断处理产生滞后等原因，都可能导致可调度性不成立。而另一方面，要保证调度系统实现的正确性，即该调度系统的实现完全符合 RMS 算法描述，对于测试、仿真等传统方法来说是相当困难的，因为这些传统方法具有不完备性（incompleteness）。尽管已经有大量应用形式化方法来分析安全攸关系统的工作，比如利用模型检测、定理证明等技术验证系统的正确性[46,48,64,131]，然而据我们所知，针对 RMS 算法的验证工作，目前只有少数[132,133]。而针对 RMS 系统实现的验证工作，目前还没有。

本小节展示如何基于第 3.3 小节提出的建模方法，利用工具 Real-Time Maude 来对一个真实的 RMS 系统实现进行建模和验证。该 RMS 系统是一个应用于某型航天控制器中真实存在的调度系统。我们对其进行了某些关键性质的形式化验证。基于一个真实的系统实现，我们在模型中考虑了系统开销及硬件平台的部分细节。我们的模型是标准 RMS 模型[113] 的扩展。

3.6.2　系统描述

3.6.2.1　RMS 算法

假定一个任务集合只包含 n 个周期性任务 $\tau_1, \ldots, \tau_n$。给定其中一个任务 τ_i，它的周期用 T_i 表示，运行时间是 C_i。假定所有任务的第一个任务实例都从 0 时刻同时开始初始化。任务实例的时限只考虑是否可运行的约束条件，即：任务 τ_i 的任意实例的时限是 τ_i 下一个实例的初始化时刻。根据 RMS 算法，我们选择将任务排序，使得 $T_1 \leq T_2 \leq \ldots \leq T_n$。下标 i 越小，任务 τ_i 的优先级越高，即 τ_1 优先级最高，τ_n 优先级最低。RMS 算法假设模型满足以下条件：

(A1) 任意任务 τ_i 的所有任务实例都在时刻 kT_i 进行初始化，其中整数 $k \geq 0$；

(A2) 任意任务 τ_i 的运行时间 C_i 是常数，它不随时间而改变；

(A3) 所有任务之间互相独立，使得它们在初始化的那一刻就可以被运行，并且随时可被其它任务抢占，即不考虑任何阻塞；

(A4) 所有系统开销，如任务切换产生的系统开销等，均被忽略。

图 3.8(a) 给出了一个 RMS 算法调度的例子。任务 τ_1 的周期 $T_1 = 10$，运行时间 $C_1 = 3$；任务 τ_2 的周期 $T_2 = 20$，运行时间 $C_2 = 2$。在时刻 0，两个任务的第一个实例同时开始初始化。由于 τ_1 优先级较高，τ_1 的第一个实例得以马上运行，τ_2 的第一个实例处于就绪状态。在 τ_1 第一个实例运行的过程中，由于没有优先级更高的任务实例进行初始化，τ_1 连续运行 $C_1 = 3$ 个时间单位。在时刻 3，τ_1 的第一个实例运行结束，调度算法在就绪的任务实例中寻找优先级最高的任务实例予以执行，于是 τ_2 的第一个实例开始运行。在 τ_2 运行过程中，由于没有优先级更高的任务进行初始化（注意 τ_1 的周期为 10），τ_2 连续运行 $C_2 = 2$ 个时间单位至结束。在时刻 5，τ_2 的第一个实例运行结束，此时没有其它就绪的任务实例，系统处于空闲状态。直到时刻 10，任务 τ_1 的第二个实例进行初始化。由于此时没有达到任务 τ_2 的周期，τ_2 不进行初始化，τ_1 的实例就绪并运行。在时刻 20，根据 τ_1、τ_2 的周期，二者再次同时初始化，并以上述方式重复运行。

RMS 算法的模型是个理想模型。我们在本案例研究的对象是一个 RMS 系统实现，而非 RMS 算法本身。因此我们将要讨论的模型比 RMS 标准模型要复杂得多。由于我们要考虑实际系统的中断屏蔽机制，因此假设条件 (A1) 将不再满足。同时 (A4) 也将被放宽，以便得到一个更真实的分析模型。

3.6.2.2　RMS 系统

本案例讨论的 RMS 系统来自于某工业级航天控制系统，基于中断机制由 C 语言实现。该系统中只存在一类中断，由时间触发。时钟中断的周期为 T。系统存在

中断屏蔽标志位。当屏蔽标志位为 0 时，系统可中断；反之则不可中断。当中断请求发生时，如果系统处于可中断状态，则中断处理函数 *schedule*() 将被调用；否则如果系统处于不可中断状态，*schedule*() 将被阻塞，直到中断屏蔽标志位被清零。图 3.7 展示了中断处理函数 *schedule*() 的伪码，其中 *taskList* 是被调度的周期性任务组成的链表。这里假设该链表中的任务是按优先级降序排列的，即优先级最高的任务在链表头，优先级最低的任务在链表尾。变量 *taskList* 和 *timer* 是全局变量。在该系统实现中，只存在时钟中断这一类中断。任意任务 τ_i 的周期 T_i 是时钟中断周期 T 的整数倍。各周期性任务之间互相独立，满足假设条件 (A3)。

```
1:  function schedule()
2:      int_off();                                  ▷ to disable interrupts
3:      updateStatus(taskList);
4:      timer = timer + 1;
5:      p = taskList;
6:      while p do
7:          if p → status == INTERRUPT then
8:              return ;
9:          else if p → status == READY then
10:             p → status = RUNNING;
11:             int_on();                           ▷ to enable interrupts
12:             p → function();                     ▷ to execute the task
13:             int_off();
14:             p → status = DORMANT;
15:         end if
16:         p = p → next;
17:     end while
18: end function
19: function updateStatus(p)
20:     while p do
21:         if p → status == RUNNING then
22:             p → status = INTERRUPT;
23:         end if
24:         if timer % (p → period) == 0 then   ▷ task should be initiated
25:             if p → status == DORMANT then   ▷ previous job finishes
26:                 p → status = READY;
27:             else                                ▷ READY or INTERRUPT
28:                 reportTaskError(p);             ▷ task misses its deadline
29:             end if
30:         end if
31:         p = p → next;
32:     end while
33: end function
```

图 3.7　中断处理函数 *schedule*() 的类 C 伪码

在图 3.7 展示的伪码中，中断处理函数 *schedule*() 首先通过调用函数 *updateStatus*() 更新了 *taskList* 中所有任务的状态。这个更新的动作，实际上对当前中断周期需要被调度的任务进行了初始化。然后 *schedule*() 对链表 *taskList* 进行遍历，对就绪的任务（状态为 *READY*）轮流予以执行，或者当遇到一个被中断的任务（状态为 *INTERRUPT*[①]）时，进行返回操作（return）。由于 *taskList* 中的任务是按优先级降序排列的，这样的遍历方式意味着调度从优先级高的任务开始。至于函数 *updateStatus*()，它对每一个任务的更新操作分为两步：首先，如果该任务正在运行（状态为 *RUNNING*），则成为被中断状态；其次，当该任务处于应当进行初始化的时刻，如果该任务的上一个实例已经运行结束（状态为 *DORMANT*），则变成就绪状态，否则意味着它错过了时限，将产生一个错误。需要注意，函数 *schedule*() 仅仅在中断请求被处理时才会被调用。如果系统处于不可中断状态，即中断屏蔽标志位为 1，该函数则不会被调用。得益于中断屏蔽标志位，当函数 *schedule*() 正在更新任务状态或寻找下一个该被运行的任务时，它不能被中断。然而，当它正在运行某个周期性任务时（第 12 行），它可以被中断。这就造成了函数 *schedule*() 可能被嵌套调用。

为简便起见，本文用“调度过程”（scheduling）指代如下阶段：从中断请求被响应的时刻起，到第一个应当被运行的周期性任务开始运行（即图 3.7 中第 8 行或第 12 行）的时刻止。因此，“调度开销”包括三部分：

1. 当中断请求被响应时，从正在运行的任务函数切换到 *schedule*() 函数所耗费的上下文切换时间；
2. *schedule*() 用于搜索第一个应当被运行的任务，以及准备运行该任务所花费的时间，对应图 3.7 中第 2–11 行；
3. 从函数 *schedule*() 切换到应当被运行的任务函数所耗费的上下文切换时间。

“任务切换”（switching）指代以下阶段：从一个周期性任务实例完成其运行的时刻起，到下一个应当被运行的周期性任务开始运行的时刻止。同样，“任务切换开销”也包含三部分：

1. 当某个周期性任务实例运行完毕时，从该任务函数切换至 *schedule*() 所耗费的上下文切换时间；
2. *schedule*() 用于搜索下一个应当被运行的任务，以及准备运行该任务所花费的时间；
3. 从函数 *schedule*() 切换到应当被运行的任务函数所耗费的上下文切换时间。

① 需要注意，状态 *INTERRUPT* 表示该任务当前被中断，或者它曾经被中断而它目前仍未运行完毕。

3.6.3　系统建模

对硬件平台的部分技术细节（比如中断屏蔽技术）予以考虑，我们假设系统模型满足以下条件：

(A1') 任务 τ_i 的实例初始化时刻，等于在时刻 kT_i 发出的时钟中断请求被响应的时刻，其中整数 $k \geq 0$；

(A2) 任意任务 τ_i 的运行时间 C_i 是常数，它不随时间而改变；

(A3) 所有任务之间互相独立，使得它们在初始化的那一刻就可以被运行，并且随时可被其它任务抢占；

(A4') 系统的调度开销以及任务切换开销需要在模型中予以考虑，其它系统开销均被忽略。

这些假设条件使我们的模型有别于标准 RMS 模型。比如说，根据假设条件 (A1')，如果中断请求出现在任务切换的过程中，那么它的响应将被推迟，使得 τ_i 的任务实例不能在时刻 kT_i 进行初始化。它们将被推迟，直到任务切换过程结束时中断屏蔽位被清零。这与假设条件 (A1) 是不一样的。另一方面，条件 (A3) 假定任务实例在初始化时刻就进入就绪状态，随时可以被运行。然而，任何任务实例都不可能在初始化时刻就开始运行，因为调度过程也需要耗时。

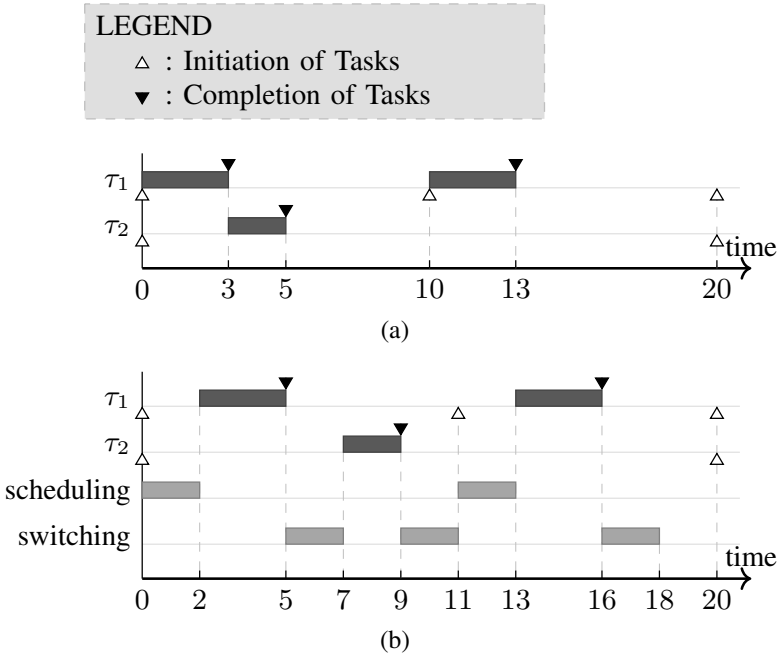


图 3.8　RMS 算法和 RMS 系统实现调度对比

根据这些假设条件，图 3.8(b) 展示了图 3.8(a) 的任务集在 RMS 系统实现中被调度的情况，其中假定调度开销和任务切换开销都为 2 个时间单位，时钟中断周

期为 10。与图 3.8(a) 不同，在图 3.8(b) 中，在时刻 0，首先开始被运行的是调度过程。在经过了 2 个时间单位后，调度过程结束，优先级最高的 τ_1 才开始被运行。在时刻 5，当 τ_1 运行结束后，系统需要进行长达 2 个时间单位的任务切换，然后在时刻 7，任务 τ_2 才开始被运行。特别需要注意的是，在时刻 10，虽然有时钟中断请求发生，但因为正处于任务切换过程中，中断请求被屏蔽，中断响应被推迟。直到时刻 11 任务切换结束时，中断请求才被响应。τ_1 的初始化时刻是 11，而不是 10。这个行为展示了条件 (A1') 与 (A1) 的不同。

在本小节，我们将首先介绍如何利用带类型的项表达式来对系统状态建模，然后展示如何利用重写规则对系统的关键行为进行建模。特别需要指出，系统中的瞬时行为（instantaneous behaviors）将在第 3.6.3.3、3.6.3.4 和 3.6.3.5 小节进行解释；在第 3.6.3.6 小节，将对系统的时间行为（timed behaviors）进行解释。根据第 3.3.6 小节所述，系统中涉及时间的行为将利用 $\mathcal{R}$ 规则进行描述，根据算法 1，即利用 Maude 的 *R* 规则进行建模。

3.6.3.1　基本类型

在建立的系统模型中，任务将由它们在 *taskList* 中的下标进行标识。所有下标具有自然数类型 `Nat`。我们定义类型 `MaybeNat` 对类型 `Nat` 进行封装，用于指代任务集中的某个周期性任务。类型 `MaybeNat` 具有两个构造子（constructor）：构造子 `some` 带有类型为 `Nat` 的参数 *n*，表示 *taskList* 中第 *n* 个任务；构造子 `none` 指代“没有任务”。

```
op none : -> MaybeNat [ctor] .
op some_ : Nat -> MaybeNat [ctor] .
```

其中关键字 `ctor` 表示所对应的函数符号是构造子。

类型 `Stack` 被定义用于对系统的栈结构进行建模。栈中存储的是被中断的任务（编号）。基于类型 `Stack`，我们定义了栈操作 `push`、`pop` 和 `peek`。

类型 `Counter` 被定义作为计数器，记录某个任务的已运行时间和运行时间。

在实际的代码中，*schedule*() 函数的全局变量 *timer* 在达到上界时将会被置 0。这一合理的技术细节没有在图 3.7 进行展示，但包含在我们的模型中。*timer* 的上界等于所有任务周期的最小公倍数。类型 `Timer` 被定义用于对 *timer* 进行建模。

3.6.3.2　系统状态的建模

整个对象系统可看作由以下几部分组成：被调度的任务集；RMS 调度模块；硬件平台，包括寄存器和栈；以及时钟中断源。其中 RMS 调度模块（即函数 *schedule*()）

的状态可以只用单一变量 *timer* 来表示。下面将给出其它部分的状态模型。

任务：　由于我们的模型只关心调度问题，任务将被抽象成一个类型为 Counter 的计数器。因为需要考虑调度过程和任务切换，这两个过程在模型中被当作两个*系统任务*。每一个任务都被建模成基类 Task 的某个子类的实例对象：

```
class Task | cnt : Counter .
op error : -> Object [ctor] .
```

其中 error 是用于表示某任务超时（即错过了时限）的实例对象。

被调度的周期性任务是类 PTask 的实例。类 PTask 是 Task 的子类，具有额外的属性 priority（表示任务优先级）、period（表示任务周期）和 status（表示任务状态）：

```
class PTask | priority : Nat, period : Nat,
              status : Status .
subclass PTask < Task .
```

其中类型 Status 具有四个常量构造子 RUNNING、INTERRUPT、READY 和 DORMANT，与图 3.7 相对应。

周期性任务的链表（即系统实现中的变量 *taskList*）用类型 TaskList 进行建模。TaskList 是由 PTask 的实例对象以及 error 对象组成的列表结构。周期性任务由该列表中的元素下标进行标识。

另一方面，系统任务是类 SysTask 的实例。类 SysTask 也是 Task 的子类，但没有额外的属性。与周期性任务不同的是，系统任务的集合具有类型 SysTasks，而 SysTasks 是多重集而非列表。系统任务由自身的对象标识 Oid 进行标识。

硬件：　我们的系统模型考虑了与中断处理有关的硬件部分——寄存器和栈。

寄存器被建模成类 Regs 的实例。该类的属性 pc 描述程序计数器 PC（Program Counter），属性 mask 表示中断屏蔽标志位，属性 ir 表示中断请求标志位：

```
class Regs | pc : TaskID,
             mask : Bool, ir : Bool .
```

其中，类型 TaskID 包含子类型 MaybeNat 和 Oid，用于标识某个任务（包括周期性任务和系统任务）。类 Regs 定义了对各属性的操作，如 getPc 和 setMask 等。

于是硬件状态被建模成具有类型 Hardware 的项表达式。Hardware 由两部分构成：类 Regs 的一个实例，和类型为 Stack 的一个项表达式。

中断源：　中断源被建模成类 IntSrc 的一个实例。类 IntSrc 具有两个属性：cycle 表示时钟中断周期 T；val 的数值随着时间推移从 T 递减为 0。

```
class IntSrc | val : Time, cycle : Time .
```

系统状态：　在我们的模型中，系统状态由以上部分组合构成，具有类型 System[①]：

```
op _____ : TaskList Timer SysTasks
           Hardware Object ~> System [ctor] .
mb (L T STS HW < O : IntSrc |>) : System .
```

其中，“~>”表示该函数符号是个*部分函数*（partial function）；关键字 mb 声明一种*成员关系*（membership），在这里表示，如果一个项表达式由 TaskList（周期任务列表）、Timer（变量 *timer*）、SysTasks（系统任务集合）、Hardware（硬件）以及类 IntSrc（中断源）的实例构成，那么它具有类型 System。

3.6.3.3　中断请求

当属性 val 的值递减为 0 时，中断请求由中断源触发。中断请求每隔周期 T 将会发出一次。请求的动作是瞬间完成的，因此将由以下瞬时条件重写规则进行建模。该规则作用于类型为 System 的项表达式：

```
crl [interrupt-request] :
  (L T STS HW ISRC)
  => (L T STS (HW).intReq reset(ISRC))
  if (ISRC).timeout .
```

其中，函数 _.timeout 检查属性 val 的值是否为 0；函数 _.intReq 将属性 ir 置 1，表示当前存在中断请求需要被响应。

然后该中断请求将等待被响应处理，其过程的建模将在第 3.6.3.5 小节进行解释。

3.6.3.4　任务初始化

周期性任务将按顺序被图 3.7 中的函数 *updateStatus*() 进行初始化。这一过程在我们的模型中被看作一个瞬时行为。它将由以下函数 updateStatus_with_ 进行建模：

① 根据 Maude 的使用习惯，变量符号将由大写字母表示。为了易于阅读，变量声明语句在此省略。

```
op updateStatus_with_ : TaskList Timer
                          -> TaskList .
```

该函数对 *taskList* 中的任务逐一应用函数 update_with_，使任务状态得到更新（对应图 3.7 的第 21–30 行）：

```
op update_with_ : Object Timer ~> Object .
ceq update < O : PTask | period : T,
                         status : ST >
      with TIMER
    = if ST == DORMANT
      then < O : PTask | status : READY >
      else error fi
    if TIMER rem T == 0 .
eq update < O : PTask | status : ST >
     with TIMER
   = if ST == RUNNING
     then < O : PTask | status : INTERRUPT >
     else < O : PTask |> fi [otherwise] .
```

其中变量符号 TIMER 表示全局变量 *timer* 的值。给定一个周期性任务，如果 TIMER（即 *timer*）可被任务周期 T 整除，那么该任务需要被初始化。在任务需要被初始化的情况下：如果它处于空闲状态（DORMANT），则将进入就绪状态（READY）；否则意味着该任务的上一个任务实例没有运行完毕，因此该任务超时，产生 error 对象。在任务不需要被初始化的情况下，只有当其处于运行状态（RUNNING）时，任务状态才需要被改变。

通过对比可以看出，模型中的函数 updateStatus_with_ 具有与图 3.7 中函数 *updateStatus*() 相同的行为。

3.6.3.5　中断处理与任务调度

当中断请求发生时，它可能不会马上被系统检测到。中断请求的检测要求中断屏蔽标志位 mask 为 0。一旦系统检测到存在中断请求，中断的处理分为两步：首先是硬件对中断信号的处理，比如清空中断请求标志位 ir、将当前函数的上下文压栈等；然后是调用中断处理函数 *schedule*()。这个过程用以下瞬时重写规则进行建模：

```
crl [interrupt-handle] :
  SYSTEM
  => ((SYSTEM).interrupt).startScheduling
  if (SYSTEM).existInt .
```

其中，函数 _.existInt 检查 mask 是否为 0 且 ir 为 1。函数 _.interrupt 对硬件的中断处理机制进行建模，它进行了四步操作：

(i) 清空标志位 ir，表示中断请求已被响应；

(ii) 将当前程序计数器 pc 压进栈中，保存被中断的上下文；

(iii) 将 pc 的值设成具有类型 Oid 的项表达式 scheduling，表示系统正处于调度阶段；

(iv) 将中断屏蔽标志位 mask 置 1，屏蔽即将到来的中断请求。

与周期性任务不同，虽然调度过程也作为系统任务被建模成一个计数器 Counter，但因其功能过于重要，以至于不能将其功能完全抽象化。我们将调度过程的行为划分为三部分。第一部分包含了调度过程的时间行为。这一部分通过将调度过程看作一个系统任务（类型为 SysTask）来描述其时间行为。时间行为的建模将在第 3.6.3.6 小节中详述。其它两部分共同定义了调度过程的功能。第二部分对应于图 3.7 的第 3–4 行。它更新了 *taskList* 的状态并且给 *timer* 加 1。这一部分行为被建模成函数 _.startScheduling。它将在调度过程的开始时刻作为瞬时动作发生，如以下规则 interrupt-handle 所示：

```
op _.startScheduling : System -> System .
eq (L T STS HW ISRC).startScheduling
   = ((updateStatus L with T)
      inc(T) STS HW ISRC) .
```

第三部分对应于图 3.7 的第 6–11 行，它负责搜索第一个应该被运行的周期性任务，并准备予以执行。这部分行为被建模成函数 _.finishScheduling，并在调度过程的结束时刻作为瞬时动作发生：

```
op _.finishScheduling : System -> System .
eq (L T STS HW ISRC).finishScheduling
   = (L T (finish scheduling in STS)
      HW ISRC).run1stTask .
```

其中，函数 finish_in_ 重置系统任务 scheduling 的计数器；函数 _.run1stTask 对第 6–11 行建模，从状态为 INTERRUPT 或 READY 的任务中

找到优先级最高的任务，根据其状态分别进行中断返回（interrupt return）或执行动作。

当系统任务 scheduling 的已运行时间达到它的运行时间时，调度过程结束。我们用以下规则建模这一瞬时行为：

```
crl [scheduling-finish] :
  SYSTEM => (SYSTEM).finishScheduling
  if SYSTEM := (L T STS HW ISRC)
     /\ (SYSTEM).running == scheduling
     /\ scheduling isComplete?in STS .
```

其中，函数 _.running 返回当前系统的 pc 值，即正在运行的任务标识；而函数 _isComplete?in_ 检查该任务的已运行时间是否已达到其运行时间。

与调度过程类似，任务切换过程 switching 也被划分为时间行为与其功能行为。当正在运行的周期性任务运行结束时，系统任务 switching 开始运行，直到自身的已运行时间达到自身的运行时间。两条类似的瞬时规则 switching-start 和 switching-finish 被用于对任务切换过程的功能行为进行建模。

3.6.3.6　系统的时间行为

系统的时间行为由两部分构成：所有任务的运行和时钟中断源的运行。两者可以同时被以下标准的单元计时规则[134]（tick rule）所建模①：

```
crl [tick]:
  {SYSTEM} => {delta(SYSTEM, R)} in time R
  if R le mte(SYSTEM) [nonexec] .
```

其中，函数 delta 定义了时间推移对系统状态产生的效果；函数 mte 表示的是，从当前时刻直至任意瞬时动作必须发生的时刻，系统允许的最大时间推移量（Maximum amount of Time allowed to Elapse）。实际上，对系统时间行为建模的关键就在于定义函数 delta 和 mte。需要注意的是，变量 R 相对于我们所指定的时间域②（time domain）来说是连续的（continuous）。

时间推移对我们的目标系统的影响是，它使 pc 指定的任务和时钟中断源的状态随着时间推移在向前推进。具体的表现是，当时间向前推移，正在运行的任务的计数器 cnt 在递增，而中断源的 val 值在递减：

① 关键字 nonexec 允许 Real-Time Maude 根据某种策略来应用重写规则。

② Real-Time Maude 包含了预定义的模块用于指定时间域是自然数集或是实数集，而这两者分别定义了离散的时间域和连续的时间域。

```
ceq delta((L T STS HW ISRC), R)
    = (deltaTask(ID, L, R)
       T STS HW (deltaIS(ISRC, R)))
    if ID := (HW).getPc /\ ID :: MaybeNat .
```

其中，规则的最后一个条件表示 ID 的类型为 MaybeNat，即表示当前正在运行的任务为周期性任务。类似地，如果 ID 的类型为 Oid，即当前运行的任务为系统任务，那么函数 deltaTask 将作用于 STS 而非 L。

mte 取决于下一个强制发生的瞬时动作的时刻。因此，它由三个因素决定：还有多长时间能完成当前正在运行的任务；还有多长时间产生下一个中断请求；当前是否有中断请求被系统检测到。

```
ceq mte(L T STS HW ISRC)
    = minimum(mteTask(ID, L),
              mteIS(ISRC), mteIr(HW))
    if ID := (HW).getPc /\ ID :: MaybeNat .
```

其中，如果当前存在中断请求被系统检测到，函数 mteIr 将返回 0；否则返回 INF，表示无穷（infinity）。当 ID 为类型 Oid 时，情况类似。

3.6.4　形式化验证

在本小节，我们针对不同的真实场景，对 RMS 系统的重写模型进行分析。需要注意，得益于 *timer* 的上界，从任意（合理的）初始状态出发，该模型的可达状态数是有限的。这使得我们可以应用非时控的模型检测器对模型进行形式化验证。

3.6.4.1　验证属性

本案例我们考虑两个验证属性：可调度性和正确性。通过可调度性，我们验证一个给定的周期性任务集合在该 RMS 系统的调度下是否满足可调度性。通过正确性，我们验证该 RMS 系统对周期性任务的调度方式是否按照 RMS 算法的规范进行。

为了验证可调度性，我们定义原子命题 taskTimeout。taskTimeout 成立的条件是在当前状态中，*taskList* 包含 error 对象，即某个任务发生超时：

```
op taskTimeout : -> Prop [ctor] .
eq {L T STS HW ISRC} |= taskTimeout
   = containError(L) .
```

其中，函数 containError 返回 true 当且仅当 L 中存在 error。可调度性可以被形式化地描述成时序逻辑公式 []（~taskTimeout），它表示命题 taskTimeout 总是不成立。由于该性质与时钟无关（clock-unrelated），给定初始状态 init，以下非时控模型检测命令可以用于验证可调度性是否在任意时刻总是成立。如果成立，该命令返回 true；否则，该命令返回一条反例路径：

```
(mc init |=u [](~taskTimeout) .)
```

本案例另外一个重要目的是验证该系统实现的正确性。我们定义原子命题 correct，它成立的条件是，当前运行的周期性任务是所有请求运行的周期性任务中优先级最高的：

```
op correct : -> Prop [ctor] .
ceq {L T STS HW ISRC} |= correct
    = if ID :: MaybeNat then shouldRun(ID, L)
      else true fi
    if ID := (HW).getPc .
```

其中，函数 shouldRun(ID, L) 返回 true 的条件是，标识为 ID 的任务在当前所有非空闲状态的任务中具有最高优先级。需要注意的是，在系统的运行过程中，如果存在任务超时，我们的验证需求不关心任务超时后的行为（比如错误恢复等）。因此，实现正确性可以被以下时序逻辑公式描述：([]correct)\/(correct U taskTimeout)，表示 correct 总是成立，或者保持成立直到命题 taskTimeout 成立。给定初始状态 init，正确性能被以下非时控模型检测命令所验证：

```
(mc init |=u ([]correct)
             \/ (correct U taskTimeout) .)
```

3.6.4.2　验证场景

我们在验证过程中使用以下参数配置，数据来源于我们的工业合作伙伴提供的真实数据：

- 时钟中断周期 T 为 $5ms$；
- 调度开销为 $38\mu s$，任务切换开销为 $20\mu s$；
- 初始状态的栈为空，程序计数器 pc 为空，中断屏蔽标志位 mask 为 0，中断请求标志位 ir 为 0。

我们在 10 种不同的场景下对我们的模型进行了验证。其中既包括我们的合作伙伴提供的真实场景，也包括我们自己设计的实验场景。其中四个场景描述如下：

- 场景 (i) 包含 2 个任务 τ_1 和 τ_2：$T_1 = 5ms, C_1 = 3ms, T_2 = 25ms, C_2 = 7ms$；
- 场景 (ii) 包含 2 个任务 τ_1 和 τ_2：$T_1 = 5ms, C_1 = 2ms, T_2 = 25ms, C_2 = 2.3ms$；
- 场景 (iii) 包含 3 个任务 τ_1、τ_2 和 τ_3：$T_1 = 5ms, C_1 = 2.7ms, T_2 = 10ms, C_2 = 2ms, T_3 = 25ms, C_3 = 3ms$；
- 场景 (iv) 包含 3 个任务 τ_1、τ_2 和 τ_3：$T_1 = 5ms, C_1 = 2.5ms, T_2 = 10ms, C_2 = 1.5ms, T_3 = 15ms, C_3 = 4.5ms$。

值得注意的是，得益于 Real-Time Maude 丰富的表达能力，对不同的任务集合，我们只需要定义不同的初始状态（类型为 `System`）即可，不需要对模型本身进行修改和调整。

在验证的过程中，我们的模型选择了连续的时间域以及极大时间采样策略（maximal time sampling strategy）。模型检测的结果显示，正确性在所有场景中成立。而对于可调度性，它在场景 (i–iii) 成立，但在场景 (iv) 中不成立。我们将模型检测命令返回的反例路径绘成图 3.9。从图中可以看到，在第 $15ms$ 时，任务 τ_3 的第一个实例还没有完成运行，而此时它的第二个任务实例进行初始化，于是发生了超时，可调度性不成立。

3.6.4.3　结果评估

本小节将说明我们的验证结果是可靠且完备的。

给定一个验证方法，如果这个方法产生的任何反例都是原问题的真正反例，则称这个验证方法是*可靠的*；如果应用该方法没有找到反例，能说明原问题也不存在反例，则称这个验证方法是*完备的*。要判断我们的方法是否可靠很简单，只需要检查产生的反例是否真反例。比如，图 3.9 的反例是真反例，这表明关于场景 (iv) 可调度性的验证结果是可靠的。然而需要证明完备性，却不太简单。我们的模型选择了连续时间域，使其行为更加贴近真实系统。然而这一选择也导致了系统的状态空间成为无穷大，不可能穷尽所有状态。

一般来说，我们没有办法证明非时控的模型检测方法对任意系统、任意时间采样策略以及任意验证属性都具有完备性。然而，Ölveczky 和 Meseguer 证明了非时控的时序逻辑模型检测方法，在应用极大时间采样策略对“某类”实时系统进行“某类”LTL 公式的验证时，是完备的[134]。“这类”实时系统称作*时间鲁棒的*（time-robust），而“这类”LTL 公式由*单元计时不变的*（tick-invariant）命题构成[①]。

① 在此我们避免引入时间鲁棒性和单元计时不变性的准确定义，因为这需要更深入的重写逻辑背景知识。

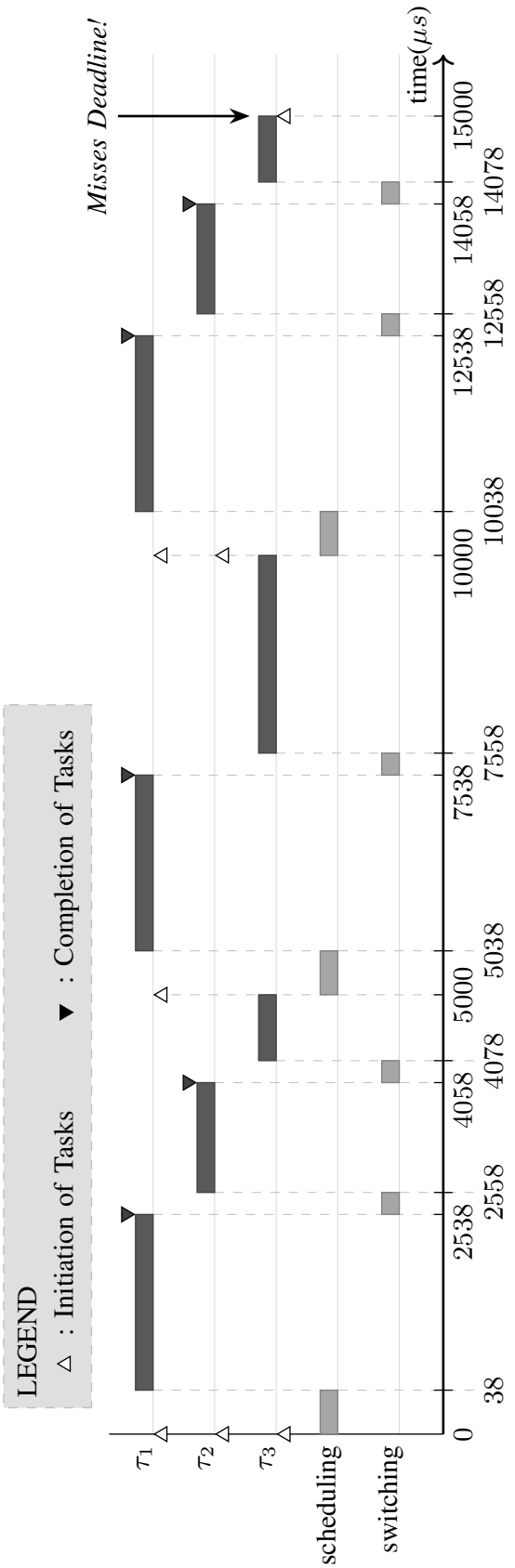


图 3.9　场景 (iv) 可调度性的反例

定理 3.1 ([134])： 给定一个时间鲁棒的实时重写逻辑模型 $\mathcal{R}^{\mathcal{L}}$，一个单元计时不变的原子命题集合 AP，一个由 AP 中命题构成的 LTL 公式 Φ （不包含 $\bigcirc$ 操作符）。那么非时控的时序逻辑模型检测方法在应用极大时间采样策略对 Φ 进行验证时，它是完备的。

因此，我们可以证明以下定理，表明第 3.6.4.2 小节的验证结果是完备的。

定理 3.2： 采用非时控模型检测方法去验证本案例的系统模型的可调度性和正确性，所得结果是完备的。

证明 首先证明我们的模型是时间鲁棒的；其次证明我们定义的原子命题 `taskTimeout` 和 `correct` 是单元计时不变的；最后应用定理 3.1 得到结论。更详细的证明，请参见附录 A。 □

3.6.5 相关工作

在本小节，我们从三个方面来对本案例应用及其相关工作进行对比。

先从可调度性判定的角度来说，Liu 和 Layland 给出了一个最常用的充分条件：一个周期性任务集合如果满足 $\Sigma_{i=1}^{n} C_i/T_i \leq n(2^{1/n}-1)$，那么它根据 RMS 算法是可调度的[113]。然后，Bini 等人提出了一个更有效的充分条件[130]，它具有跟前者同样的算法复杂度。在另一方面，判定可调度性的充分必要条件由 Sprunt 等人[115] 及 Audsley 等人[119] 分别提出，要求对任务集合进行更复杂的分析。然而，所有这些分析都建立在理想的模型上，并不考虑系统开销，真实性不如本文建立的模型。Katcher 等人基于几种主流的 RMS 实现方法，在可调度性分析中考虑了系统开销[135]。然而，本案例的目标系统不在他们的考虑范围之内。此外，相比于那些理论的分析，本文采用的基于形式化建模与验证的分析方法主要有三个优点。一是如果我们的可调度性检查给出系统“不可调度”的结果，它同时能够返回一条反例路径。这条反例路径可以指导开发人员调整系统设计，比如调整任务优先级、甚至改变调度算法。第二个优点是，如果想要在系统中应用一个全新的调度策略，本小节的分析方法只需要对模型进行修改就能对新策略进行分析，而理论分析方法则需要重新进行分析和推理。最后一个优点是，在分析中考虑系统开销和硬件细节会给模型引入不确定性。比如在我们的模型中，如果正在运行的任务恰好在中断请求发生时完成运行，则将可能发生两种不同的行为：(i) 系统进行任务切换，在任务切换过程中中断请求被屏蔽，因此调度过程和任务的初始化将被推迟；(ii) 系统立即响应该中断请求，则任务切换将被推迟。相比于本案例的自动验证方法，对这些不确定行为进行理论分析要复杂得多。

Tian 和 Duan[132] 以及 Cui 等人[133] 以类似的方法，利用模型检测技术对 RMS 算法进行了分析。与本案例不同的是，这两项工作使用了不同的建模语言和工具。Tian 和 Duan 使用 SPIN[103] 的一类扩展对 RMS 算法进行建模和验证[132]；而 Cui 等人则使用了逻辑编程语言 TMSVL[136] 及其模型检测工具[133]。这两项工作与本工作存在两个主要区别。首先，最重要的区别在于，这两项工作的分析对象都是 RMS 算法的标准理想模型[113]，而不是包含了更多复杂细节的系统实现，这也是本案例应用的主要出发点。实际上，如果我们假设调度过程和任务切换的时间等于 0，那么标准的 RMS 模型将成为我们的 RMS 系统模型的特例。因此，本案例建立的模型更具一般性。另外一个区别在于，如果需要在周期任务集合中添加一个新任务，Tian 和 Duan 的模型[132] 以及 Cui 等人的模型[133] 都需要进行修改，为新任务及其行为增加一个子模块。特别需要指出的是，Tian 和 Duan 模型中的调度部分同样需要调整以增加新任务[132]。然而，在我们的模型中，增加新任务只需要修改初始状态即可，模型不需作任何修改。这一点已经在第 3.6.4.2 小节中指出。另一方面，Tian 和 Duan 在模型中使用离散时间域[132]，而我们的模型则使用了参数化的时间域，允许实例化成离散时间域或连续时间域，非常灵活。Cui 等人也使用了连续的时间域，同时采用建模策略以减少状态空间规模[133]，与本工作的极大时间采样策略类似。但我们证明了此方法的完备性（定理 3.2），而 Cui 等人没有给出类似结论。

最后，Maude 和 Real-Time Maude 目前已成功地被应用到诸多领域[60]，特别是通信协议、安全协议。但还没有被应用于分析调度问题。据我们所知，这是首次利用 Real-Time Maude 分析 RMS 算法及其实现。

3.6.6　案例小结

在本案例应用中，我们利用 Real-Time Maude 对一个真实的 RMS 系统进行了建模。在建立的系统模型中，我们考虑了调度及任务切换所产生的系统开销，也考虑了硬件平台的部分细节。我们的模型包含了足够多的细节对真实的目标系统行为进行描述。通过对建立的模型应用模型检测技术，我们在不同的关键场景下验证了系统的可调度性和正确性。最后我们证明了验证结果是可靠且完备的。该系统目前在某工业级航天控制器中在线运行。

3.7　本章小结

本章基于已有的重写逻辑建模验证工作，针对嵌入式系统的特性，如结构层次化、高度并发、并发行为与顺序行为并存、系统结构动态变化、实时性等，提出

了基于规范化条件重写模型的建模方法。通过模型层面的语义映射，我们将该建模方法在工具集 Maude 中予以实现。利用该方法，我们对两个真实的嵌入式系统——机车优化控制系统和速率单调调度系统进行了建模分析。对于前者，我们应用了测试与验证结合的方法，成功发现系统中的潜在缺陷；该系统目前运行稳定，并在沈阳铁路局通过了实车运用考核。而对于后者，我们应用模型检测技术对系统的可调度性和正确性进行了形式化验证，并证明了其结果具有可靠性和完备性；经验证的系统目前在某工业级航天控制器中在线运行。这两个应用案例表明了本章提出的建模方法在实际应用中具有可行性。

第 4 章　C 语言程序终止性自动验证

上一章从建模方法论的角度，试图提高规范化条件重写模型作为一种建模模型的易用性。本章将从另外一个角度切入，进一步降低利用规范化条件重写模型进行建模验证的成本。在前文的讨论中，建模过程不针对任何特定的验证属性。这要求建立的模型应尽可能涵盖系统各方面的行为特征，并包含尽可能多的细节，使得针对模型任何属性的验证结果都可以反映系统的性质。因此，为了保证模型与系统的等价性以及抽象程度的合理性，建模过程只能通过人工进行。然而，如果只针对某种特定属性，如可达性、终止性等，则建模过程只需要针对系统行为的某个侧面进行。这将显著地降低建模的难度，使得建模过程的自动化成为可能。本章在已有对 C 语言程序自动建模的工作基础上，开发实现了针对 C 语言程序终止性的自动验证工具 Ceagle-TERM。

4.1　引言

程序的终止性问题是验证领域的一个重要问题。它的重要性主要体现在两个方面。首先，终止性是*完全正确性*（total correctness）的必要条件。一个程序（或算法）被称作完全正确，当且仅当它是终止的，且它的行为符合其规约（specification）的规定。用 Hoare 逻辑[137] 表示，三元组 $\{P\}\ \mathtt{C}\ \{Q\}$ 成立，当且仅当在满足谓词 P 的前提下，(i) 程序 C 终止；且 (ii) 执行完 C 后谓词 Q 成立。而一个程序（或算法）称作*部分正确*，则只要求在假设它终止的前提下，其行为符合规约的规定。也就是说，三元组 $\{P\}\ \mathtt{C}\ \{Q\}$ 成立，当且仅当在满足谓词 P 的前提下，如果程序 C 终止，则执行完 C 后谓词 Q 成立。举以下简单例子：

```
{x == 1} while (true) x = x; {x == 1}
```

由于其 `while` 循环不终止，该程序根据其规约，不是完全正确的，但它是部分正确的。

程序终止性的第二个重要性在于，对 CTL[138]、CTL*[139]、Fair-CTL[140] 以及 LTL[35] 性质的验证，可以转化为对安全性[141]（safety）和对终止性的验证[142]。由于目前大多数程序验证工具都针对安全性进行验证，因此终止性验证工具的研究理论上可以提升这些程序验证工具的验证能力。

嵌入式系统的软件部分通常由 C 语言实现，因此开发 C 语言程序的终止性自动验证工具，对保障嵌入式系统的可靠性具有重要意义。

程序的终止与否，有时候并不容易判断，如以下例子（Collatz 问题）：

```
while (x > 1) {
  if (even(x))  x = x / 2;
  else  x = 3 * x + 1;
}
```

目前有许多技术可以自动验证一个程序的终止性，而其中一种主流技术正是基于*整数重写模型*[143]。整数重写模型可以看作是规范化条件重写模型的一种特例。在第 2 章已经提到，重写模型的终止性是该领域的最重要问题之一，目前已经有大量关于重写模型终止性的验证理论及工具[52,76–80,144–150]。通过将程序（自动）建模成整数重写模型，程序终止性问题就可以被转化为重写模型的终止性问题，从而使用相应的技术进行解决。本文基于这种技术路线，开发了 C 语言程序终止性自动验证工具 Ceagle-Term。

本章其余部分组织结构如下：第 4.2 小节介绍相关的 C 程序终止性自动验证工具；第 4.3 小节讨论如何自动建立 C 程序的整数重写模型，以便对程序终止性进行验证；Ceagle-Term 工具将在第 4.4 小节进行介绍；最后对本章内容进行简要小结。

4.2　相关工作

对指令式程序的终止性进行自动验证，目前主要有三种主流技术。

排序函数[151]（ranking function）是最经典的用于证明程序终止性的技术。排序函数 $rank$ 将每一个程序状态 s 映射为抽象域（如正整数域）中的一个元素 $rank(s)$。如果对于程序的每一步执行 $s \to s'$，排序函数满足 $rank(s) > rank(s')$，且抽象域上的二元关系 $>$ *是良基的*（well-founded），即不存在无穷递减序列，则该程序是终止的。于是验证程序终止性的问题可以被归约为排序函数的构造问题。如果满足条件的排序函数存在，则可判断该程序终止。基于排序函数构造的自动验证工具主要有 OctaTerm[152]、PolyTerm[152]、FuncTion[153,154] 和 c2fsm+Aspic+Rank[155]。其中 OctaTerm 和 PolyTerm 并不支持 C 程序输入，而 FuncTion 和 c2fsm+Aspic+Rank 可以支持。

针对排序函数构造困难的问题，ARMC[156]、CProver[157,158]、TERMINATOR[159] 及其后继工具 T2[142] 等 C 程序终止性验证工具采用了递增式的排序函数构造技术。这些工具将安全性检查过程集成到排序函数的构造过程中。其主要原理是，若当前已经有排序函数 $rank$ 对程序的某些执行序列满足递减关系，则利用安全性检查

工具找到一条不能使 $rank$ 递减的执行序列 π；根据反例 π 对 $rank$ 进行改善，得到新的排序函数 $rank'$。重复此过程直到构造的排序函数能使程序的所有执行序列满足递减关系。同样基于排序函数结合安全性检查技术的 C 程序终止性验证工具还有 SeaHorn[160] 和 2LS[161]。在 SeaHorn 的实现中，安全性检查技术的使用方式与其它工具稍有不同，目的是为了在对当前排序函数进行改善时，能获得更多信息。而 2LS 则将过程间分析技术加入到排序函数的构造过程中。

另一个主流技术是将程序的终止性验证问题归约为重写模型的终止性判定问题。通过利用重写模型（或其扩展形式）对程序中影响终止性的行为进行自动建模，若模型具有终止性，则可判定该程序也具有终止性。这种技术方案的好处是可以充分利用重写领域对终止性的判定技术，如递归路径序[144]（recursive path order）、Knuth-Bendix 序[145]（Knuth-Bendix order）、依赖对[146,147]（dependency pair）等。基于这种技术方案的 C 程序终止性验证工具主要有 AProVE[148,162]、`llvm2KITTeL+KITTeL`[143]（以下简称为 `KITTeL`）和 c2lctrs+Ctrl[149,163]（以下简称为 Ctrl）。其中 `KITTeL` 和 Ctrl 的模型直接从 C 程序的控制流图中进行抽取，因此无法对涉及指针的程序行为进行描述。而 AProVE 先通过符号执行技术构造程序的一种中间表示形式——符号执行图（symbolic execution graph），然后再从中抽取重写模型。由于符号执行图包含了程序中涉及指针的行为，因此 AProVE 的技术方案能更好地处理包含内存操作的 C 程序。

近年来也出现了一些新兴的技术可用于验证 C 程序的终止性。例如工具 HipTNT+[164] 实现了一种基于函数摘要（summary）的终止性验证技术，通过推理得到每个函数的终止性或非终止性的摘要，然后综合得到整个程序的终止性验证结果。Ultimate Büchi Automizer[165] 实现了一种基于程序构造的终止性验证技术。它首先对程序进行执行路径采样，得到若干套索形状（lasso-shaped）的程序执行路径；然后根据这些路径构造出若干具有终止性的程序片段；重复该过程直到构造出来的程序片段集合可以涵盖目标程序的行为，则终止性得到证明。

本文开发的 C 程序终止性自动验证工具 Ceagle-Term 采用基于重写模型的验证技术，且利用符号执行图作为程序的中间表示形式，与 AProVE 工具类似。AProVE 作为一个闭源工具，其实现细节我们不得而知，但根据文献 [166] 的描述，AProVE 的符号执行图构造过程在遇到递归函数时，可能无法终止。Ceagle-Term 与 AProVE 的主要区别在于，Ceagle-Term 采用了单函数的内存模型，保证了符号执行图的构造过程在遇到递归函数时可以终止。

4.3　C 程序的整数重写模型

对 C 程序的终止性进行自动验证，其关键在于自动构建 C 程序对应的整数重写模型。本文采取 Ströder 等人提出的方法，基于符号执行技术构建 C 程序的符号执行图，再根据符号执行图生成适用于终止性验证的整数重写模型[166]。

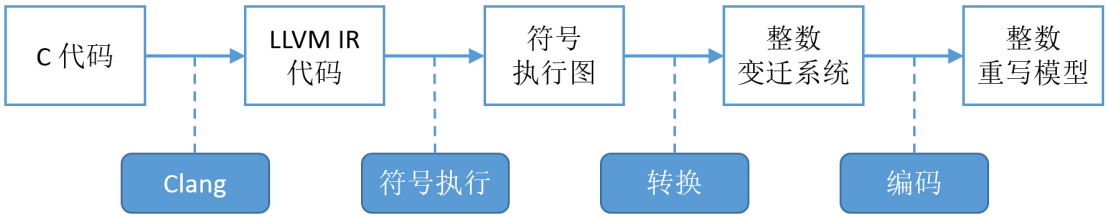


图 4.1　C 程序的整数重写模型自动构造

图 4.1 是 C 程序的整数重写模型自动构造流程。C 程序经过编译器 Clang 的编译后生成对应的 LLVM IR[167] 代码。基于这些 LLVM IR 代码，利用符号执行技术[168]，构造输入程序的符号执行图[169]。整数变迁系统[170] 描述了该符号执行图的终止性行为。最后利用重写模型丰富的表达能力，将该整数变迁系统的语义编码成整数重写模型的形式。

整个构造过程涉及到三个重要的模型：符号执行图、整数变迁系统和整数重写模型。在本小节接下来的部分将对这三个概念进行介绍，并对其构造的核心原理进行描述。

4.3.1　符号执行图

符号执行图是由 Giesl 等人提出的用于描述程序执行过程的数据结构[169]。一个符号执行图是一个有向图，它包含一个顶点集合和一个边集合。每个顶点代表一个抽象的程序执行状态，它包含程序当前的位置（PC）、变量赋值状态、内存分配情况等信息。程序执行图的边分为三类：

(i) 求值（evaluation）类型。求值类型的边表示：若其起始顶点表示的程序状态为 s，当前程序位置为 p（p 也是状态 s 中包含的信息），则该有向边的目标顶点代表的状态 s' 为 s 执行 p 位置的程序指令后产生的新状态。

(ii) 精化（refinement）类型。精化类型的边的目标顶点 v 所表示的状态是起始顶点 u 表示的状态的精化，即 v 所表示的程序状态集合是 u 所表示的程序状态集合的子集。

(iii) 抽象（abstraction）类型。与精化类型的边相对，抽象类型的边表示其目标顶

点 v 的状态是起始顶点 u 状态的抽象，即 u 所表示的程序状态集合是 v 所表示的程序状态集合的子集。

至于每个顶点所表示的*抽象程序状态*，本文对 Ströder 等人提出的定义[166] 进行了简化：

定义 4.1 (**抽象程序状态**)： 给定程序变量集合 $\mathcal{V}_{\mathcal{P}}$，符号化变量集合 $\mathcal{V}_{sym}$，程序位置集合 Pos，则一个抽象程序状态是一个六元组 $\langle p, LV, LAL, KB, AL, PT\rangle$。其中 $p \in Pos$；$LV : \mathcal{V}_{\mathcal{P}} \to \mathcal{V}_{sym}$；$LAL = \{[\![v_1, v_2]\!] \mid v_1, v_2 \in \mathcal{V}_{sym}, v_1 \le v_2\}$；$KB \subseteq QF\_IA(\mathcal{V}_{sym})$；$AL = \{[\![v_1, v_2]\!] \mid v_1, v_2 \in \mathcal{V}_{sym}, v_1 \le v_2\}$；$PT \subseteq \{(v_1 \hookrightarrow_{\mathtt{ty}} v_2) \mid v_1, v_2 \in \mathcal{V}_{sym}, \mathtt{ty}$是 LLVM 的类型$\}$。另外，抽象程序状态 ERR 表示可能违背了内存安全的状态；抽象程序状态 END 表示执行结束的程序状态。

直观上解释，LV（Local Variables）表示该状态的程序变量赋值情况，程序变量值由集合 $\mathcal{V}_{sym}$ 中的*符号化变量*表示；LAL（Local Allocation List）表示局部内存分配情况，二元组 $[\![v_1, v_2]\!]$ 表示在 v_1 和 v_2 之间的内存单元是已经过分配的；$QF\_IA(\mathcal{V}_{sym})$ 是指*无量词的*（Quantifier-Free）一阶公式，用于描述 $\mathcal{V}_{sym}$ 变量的*整数代数*（Integer Arithmetic）性质，KB（Knowledge Base）是该状态满足的性质集合；AL（Allocation List）与 LAL 类似，表示的是全局内存分配情况；PT（Pointer Table）表示内存状态，三元组 $(v_1 \hookrightarrow_{\mathtt{ty}} v_2)$ 表示内存地址 v_1 指向的内容是 v_2，具有类型 $\mathtt{ty}$。

定义 4.1 与 Ströder 等人在文献 [166] 中的定义最主要的区别在于，定义 4.1 只关心单一函数的抽象状态，原因在于我们将对多个函数进行单独分析；而后者在抽象程序状态中加入了栈结构信息，目的在于描述函数之间的调用过程。

基于定义 4.1，我们定义用于描述每个抽象程序状态的一阶谓词集合 $\langle a\rangle$：

定义 4.2 ([166])： 给定抽象程序状态 a，一阶谓词集合 $\langle a\rangle$ 是满足以下条件的最小集合：

$$
\begin{aligned}
\langle a\rangle \overset{\text{def}}{=}\ & KB \cup \{1 \le v_1 \wedge v_1 \le v_2 \mid [\![v_1, v_2]\!] \in LAL \cup AL\} \\
& \cup \{v_2 < w_1 \vee w_2 < v_1 \mid [\![v_1, v_2]\!], [\![w_1, w_2]\!] \in LAL \cup AL, (v_1, v_2) \neq (w_1, w_2)\} \\
& \cup \{v_2 = w_2 \mid (v_1 \hookrightarrow_{\mathtt{ty}} v_2), (w_1 \hookrightarrow_{\mathtt{ty}} w_2) \in PT \text{ 且 } \vDash \langle a\rangle \Rightarrow v_1 = w_1\} \\
& \cup \{v_1 \neq w_1 \mid (v_1 \hookrightarrow_{\mathtt{ty}} v_2), (w_1 \hookrightarrow_{\mathtt{ty}} w_2) \in PT \text{ 且 } \vDash \langle a\rangle \Rightarrow v_2 \neq w_2\} \\
& \cup \{v_1 > 0 \mid (v_1 \hookrightarrow_{\mathtt{ty}} v_2) \in PT\} \ 。
\end{aligned}
$$

需要注意的是，$\langle a\rangle$ 是归纳定义的。$\langle a\rangle$ 定义了抽象程序状态 a 满足的一阶谓

词性质集合。

符号执行图的构建基于符号执行规则进行。给定一个抽象程序状态，根据符号执行规则计算下一个（或两个）抽象程序状态。由于实际的程序状态空间可能是无穷的，为了用符号执行图的有穷顶点数表示无穷的程序状态空间，在构造过程中需要根据一定的策略对抽象程序状态进行抽象。下面先介绍符号执行规则，再介绍符号执行图的构造策略。

根据定义 4.1，我们对 Ströder 等人提出的符号执行规则[166] 进行简化。以下列举其中较关键的几条符号执行规则。

表 4.1　符号执行规则：`load` 指令（内存已分配）

load 已分配内存

$$\frac{\langle p, LV, LAL, KB, AL, PT\rangle}{\langle p^{+}, LV[\mathtt{x} := w], LAL, KB, AL, PT \cup \{LV(\mathtt{ad}) \hookrightarrow_{\mathtt{ty}} w\}\rangle}$$

如果满足以下条件

- p : "`x = load ty* ad`" 其中 $\mathtt{x}, \mathtt{ad} \in \mathcal{V}_{\mathcal{P}}$；
- 存在 $[\![v_1, v_2]\!] \in LAL \cup AL$
  使得 $\vDash \langle a\rangle \Rightarrow (v_1 \leq LV(\mathtt{ad}) \wedge LV(\mathtt{ad}) + size(\mathtt{ty}) - 1 \leq v_2)$；
- $w \in \mathcal{V}_{sym}$ 是新变量。

表 4.2　符号执行规则：`load` 指令（内存未分配）

load 未分配内存

$$\frac{\langle p, LV, LAL, KB, AL, PT\rangle}{ERR}$$

如果满足以下条件

- p : "`x = load ty* ad`" 其中 $\mathtt{x}, \mathtt{ad} \in \mathcal{V}_{\mathcal{P}}$；
- 不存在 $[\![v_1, v_2]\!] \in LAL \cup AL$
  使得 $\vDash \langle a\rangle \Rightarrow (v_1 \leq LV(\mathtt{ad}) \wedge LV(\mathtt{ad}) + size(\mathtt{ty}) - 1 \leq v_2)$。

表 4.1 和 4.2 分别是 `load` 指令从已分配内存和未分配内存中读取数据时对应的符号执行规则。如表 4.1 所示，当 `load` 指令准备从地址 `ad` 中读取数据时，需要先判断是否能根据当前抽象程序状态 a 的谓词表达式 $\langle a\rangle$ 推断出地址 `ad` 对应的内存块 $[\![LV(\mathtt{ad}), LV(\mathtt{ad}) + size(\mathtt{ty}) - 1]\!]$ 已经经过分配。如果该内存块已经经过分配，则可执行表 4.1 中的符号执行规则生成新的抽象程序状态，新状态中的程序变量 `x` 得到更新，程序指针指向 p 的下一条指令 p^{+}。如果不能推断出该内存块已经经过分配，则如表 4.2 所示，将产生抽象程序状态 ERR。

表 4.3　符号执行规则：`icmp eq` 指令（命题成立）

`icmp eq` 命题为真

$$\frac{\langle p, LV, LAL, KB, AL, PT\rangle}{\langle p^{+}, LV[\mathtt{x} := w], LAL, KB \cup \{w = 1\}, AL, PT\rangle}$$

如果满足以下条件

- p : "`x = icmp eq ty` t_1, t_2" 其中 $\mathtt{x} \in \mathcal{V}_{\mathcal{P}}$ 且 $t_1, t_2 \in \mathcal{V}_{\mathcal{P}} \cup \mathbb{Z}$;
- $\vDash \langle a\rangle \Rightarrow (LV(t_1) = LV(t_2))$;
- $w \in \mathcal{V}_{sym}$ 是新变量。

表 4.4　符号执行规则：`icmp eq` 指令（命题不成立）

`icmp eq` 命题为假

$$\frac{\langle p, LV, LAL, KB, AL, PT\rangle}{\langle p^{+}, LV[\mathtt{x} := w], LAL, KB \cup \{w = 0\}, AL, PT\rangle}$$

如果满足以下条件

- p : "`x = icmp eq ty` t_1, t_2" 其中 $\mathtt{x} \in \mathcal{V}_{\mathcal{P}}$ 且 $t_1, t_2 \in \mathcal{V}_{\mathcal{P}} \cup \mathbb{Z}$;
- $\vDash \langle a\rangle \Rightarrow (LV(t_1) \neq LV(t_2))$;
- $w \in \mathcal{V}_{sym}$ 是新变量。

表 4.3 和 4.4 是指令 `icmp` 的符号执行规则。这里只列出了 `icmp eq` 作为示例，其它比较操作（如 `ule`、`uge` 等）的规则类似。如果当前状态的性质集合 $\langle a\rangle$ 可以推断出 `icmp eq` 指令对应命题为真，如表 4.3 所示，则下一状态将程序变量 `x` 的值赋为 w，KB 告诉我们 $w = 1$。否则，如果 $\langle a\rangle$ 足以推断出该命题为假，则 `x` 对应的值为 0。如果 $\langle a\rangle$ 所包含的信息不能推断出目标命题的真假，这说明下一个状态需要分情况讨论。于是我们需要往当前的抽象程序状态加入更多的信息，对状态进行“精化”。

表 4.5　符号执行规则：精化（`icmp eq` 指令）

对 **`icmp eq`** 进行精化

$$\frac{\langle p, LV, LAL, KB, AL, PT\rangle}{\langle p, LV, LAL, KB \cup \{\varphi\}, AL, PT\rangle \;\mid\; \langle p, LV, LAL, KB \cup \{\neg\varphi\}, AL, PT\rangle}$$

如果满足以下条件

- p : "`x = icmp eq ty` t_1, t_2" 其中 $\mathtt{x} \in \mathcal{V}_{\mathcal{P}}$ 且 $t_1, t_2 \in \mathcal{V}_{\mathcal{P}} \cup \mathbb{Z}$;
- $\nvDash \langle a\rangle \Rightarrow \varphi$ 且 $\nvDash \langle a\rangle \Rightarrow \neg\varphi$;
- φ 是 $LV(t_1) = LV(t_2)$。

表 4.5 是对指令 `icmp eq` 进行精化的规则。若根据当前抽象状态的信息不能推断出命题 φ 的真假，则精化规则产生两个新的抽象程序状态，两个状态的集合

KB 分别加入了 φ 的真假信息，使后续的状态求值可以继续进行。其它涉及条件比较指令的精化规则，也可以类似地进行定义。

表 4.6　符号执行规则：`add` 指令

add

$$\frac{\langle p, LV, LAL, KB, AL, PT\rangle}{\langle p^{+}, LV[\mathtt{x} := w], LAL, KB \cup \{w = LV(t_1) + LV(t_2)\}, AL, PT\rangle}$$

如果满足以下条件

- p : "`x = add ty` t_1, t_2" 其中 $\mathtt{x} \in \mathcal{V}_{\mathcal{P}}$ 且 $t_1, t_2 \in \mathcal{V}_{\mathcal{P}} \cup \mathbb{Z}$;
- $w \in \mathcal{V}_{sym}$ 是新变量。

表 4.6 展示了代数运算指令 `add` 的符号执行规则，其它代数运算（如 `sub`、`mul` 等）的符号执行规则与其类似。

表 4.7　符号执行规则：`alloca` 指令（分配错误）

alloca 失败

$$\frac{\langle p, LV, LAL, KB, AL, PT\rangle}{ERR}$$

如果满足以下条件

- p : "`x = alloca ty, in` t" 其中 $\mathtt{x} \in \mathcal{V}_{\mathcal{P}}$ 且 $t \in \mathcal{V}_{\mathcal{P}} \cup \mathbb{Z}$;
- $\nvDash \langle a\rangle \Rightarrow (LV(t) > 0)$。

表 4.8　符号执行规则：`alloca` 指令（分配成功）

alloca 成功

$$\frac{\langle p, LV, LAL, KB, AL, PT\rangle}{\langle p^{+}, LV[\mathtt{x} := v_1], LAL \cup \{[\![v_1, v_2]\!]\}, KB \cup \{v_2 = v_1 + size(\mathtt{ty}) \cdot LV(t) - 1\}, AL, PT\rangle}$$

如果满足以下条件

- p : "`x = alloca ty, in` t" 其中 $\mathtt{x} \in \mathcal{V}_{\mathcal{P}}$ 且 $t \in \mathcal{V}_{\mathcal{P}} \cup \mathbb{Z}$;
- $\vDash \langle a\rangle \Rightarrow (LV(t) > 0)$;
- $v_1, v_2 \in \mathcal{V}_{sym}$ 是新变量。

表 4.7 和 4.8 是 `alloca` 指令的符号执行规则。分配内存成功的前提是，当前状态信息足以推断出分配的内存块大小为正数，否则将产生错误状态 ERR（如表 4.7）。当分配内存成功时，新状态的主要变化是其局部内存分配表 LAL 中增加了一块新的内存区域 $[\![v_1, v_2]\!]$。符号变量 v_1 与 v_2 的关系在性质集合 KB 中体现。

表 4.9　符号执行规则：`call` 指令

call

$$\frac{\langle p, LV, LAL, KB, AL, PT\rangle}{\langle p^{+}, LV[\mathtt{x} := w], LAL, KB, AL, PT\rangle}$$

如果满足以下条件

- p : "`x = call ty ...`" 其中 $\mathtt{x} \in \mathcal{V}_{\mathcal{P}}$；
- $w \in \mathcal{V}_{sym}$ 是新变量。

表 4.10　符号执行规则：`ret` 指令

ret

$$\frac{\langle p, LV, LAL, KB, AL, PT\rangle}{END}$$

如果满足以下条件

- p : "`ret ty` t" 其中 $t \in \mathcal{V}_{\mathcal{P}} \cup \mathbb{Z}$。

表 4.9 和 4.10 分别是 `call` 指令和 `ret` 指令的符号执行规则。由于本文采取单函数分析的方法，因此针对 `call` 和 `ret` 设计的符号执行规则相较于 Ströder 等人提出的规则得到了简化。应用 `call` 指令时，由于我们对所有函数逐一进行终止性分析，因此可以假设被 `call` 调用的函数具有终止性，且将返回某值。于是表 4.9 的执行规则中，变量 `x` 被赋予新的未知值。在程序中遇到 `ret` 指令时，由于是单函数分析，如表 4.10 所示，下一状态为结束状态 END。

表 4.11　符号执行规则：抽象

利用代换 μ 进行抽象

$$\frac{\langle p, LV, LAL, KB, AL, PT\rangle}{\langle p, LV', LAL', KB', AL', PT'\rangle}$$

如果满足以下条件

- a 有一条求值类型的入边；
- LV 和 LV' 的域相同，而且对所有 $\mathtt{x} \in \mathcal{V}_{\mathcal{P}}$ 满足 $LV(\mathtt{x}) = \mu(LV'(\mathtt{x}))$；
- $\vDash \langle a\rangle \Rightarrow \mu(KB')$；
- 如果 $[\![v_1, v_2]\!] \in LAL'$，那么 $[\![\mu(v_1), \mu(v_2)]\!] \in LAL$；
- 如果 $[\![v_1, v_2]\!] \in AL'$，那么 $[\![\mu(v_1), \mu(v_2)]\!] \in AL$；
- 如果 $(v_1 \hookrightarrow_{\mathtt{ty}} v_2) \in PT'$，那么 $(\mu(v_1) \hookrightarrow_{\mathtt{ty}} \mu(v_2)) \in PT$。

最后是对抽象程序状态的抽象规则，如表 4.11 所示。抽象后的新状态拥有同样的程序位置，新状态的信息可由原状态 a 的一阶谓词集合 $\langle a\rangle$ 推出。

有了符号执行规则，我们根据以下策略构造符号执行图[166]：

- 假设当前需要应用符号执行规则的状态顶点为 b，如果存在某个状态顶点 a 满足：(i) 存在从 a 到 b 的一条路径；(ii) a 和 b 的程序位置 p 相同；(iii) a 和 b 的 LV 映射域相同；(iv) b 存在一条求值类型的入边；(v) a 不存在精化类型的入边。那么
  - 如果状态 a 是状态 b 的抽象，那么构造一条抽象类型的边从 b 指向 a。
  - 否则，移除 a 的后继顶点，计算 a 和 b 的抽象 c 并构造一条抽象类型的边从 a 指向 c。如果 a 已经存在来自某顶点 q 的抽象类型入边，则将 a 删除并构造一条抽象类型的边从 q 指向 c。
- 否则，对 b 应用除抽象规则以外的符号执行规则，构造其后继顶点。

4.3.2　整数变迁系统

定义 4.3 (整数变迁系统[170])： 整数变迁系统是一个三元组 $\langle S, C, \rightsquigarrow \rangle$：

- S 是一组状态集合 $\{s_1, \ldots, s_n\}$；
- C 是一组条件集合 $\{c_1, \ldots, c_m\}$；
- $\rightsquigarrow \subseteq S \times C \times S$ 是一组变迁集合。

其中，条件 $c_i \subseteq QF\_IA(\mathcal{V} \cup \mathcal{V}')$ 是一组一阶公式，$\mathcal{V}$ 是整数类型的变量集合，$\mathcal{V}' = \{v' \mid v \in \mathcal{V}\}$ 表示经过变迁后的变量值。

整数变迁系统可以抽象地表示一个程序的状态空间和状态变迁。给定一个符号执行图，可以根据以下策略构造其对应的整数变迁系统[166]：

(i) 符号执行图中每一个顶点 a 都对应到整数变迁系统的一个状态 s_a；

(ii) 为符号执行图每一条从 a 到 b 的边构造一条对应的整数变迁系统的变迁 $\langle s_a, c, s_b \rangle$：

- 如果 $\langle a, b \rangle$ 不是一条抽象类型的边，则条件 $c = (\langle a \rangle \cup \{v' = v | v \in \mathcal{V}_{sym}(a)\})$，其中 $\mathcal{V}_{sym}(a)$ 表示状态 a 的所有符号化变量；
- 如果 $\langle a, b \rangle$ 是一条利用代换 μ 的抽象类型边，则条件 $c = (\langle a \rangle \cup \{v' = \mu(v) \mid v \in \mathcal{V}_{sym}(b)\})$。

4.3.3　整数重写模型

整数重写模型[143] 是规范化条件重写模型 $\mathcal{R}_{\mathcal{SE}} = \langle \mathcal{R}, \mathcal{S}, \mathcal{E} \rangle$ 的一种实例。当等价模型 $\mathcal{E}$ 取整数代数运算符的性质集合（如加法交换律、乘法结合律等），化简模型 $\mathcal{S}$ 取整数代数[①] 的计算规则时，规范化条件重写模型 $\mathcal{R}_{\mathcal{SE}}$ 实例化为整数重写模型，记作 $\mathcal{R}_{\mathcal{I}}$。

① 更准确地说，还需要考虑定义在整数代数表达式上的无量词一阶逻辑公式。

整数重写模型的重写规则通常具有如下形式：

$$f(x_1,\ldots,x_n) \quad \rightarrow \quad g(e_1,\ldots,e_m) \quad \Leftarrow \quad \varphi$$

其中，$e_1,\ldots,e_m$ 为整数代数表达式，$\varphi \in QF\_IA(\mathcal{V}\cup\mathcal{V}')$。

给定一个整数变迁系统，系统中的每一条变迁 $\langle s_i, c, s_j\rangle$ 都可以用一条重写规则进行编码[143,171]：

$$s_i(x_1,\ldots,x_n) \quad \rightarrow \quad s_j(e_1,\ldots,e_n) \quad \Leftarrow \quad \varphi_c$$

其中，$x_1,\ldots,x_n$ 按 $\mathcal{V}$ 中变量的固定序排列，且

- 如果 c 中包含“赋值语句” $x_i' = p$，则 $e_i = p$；否则 $e_i = x_i$。
- φ_c 是 c 中除去“赋值语句”后的所有一阶公式的合取。

4.4　Ceagle-Term

基于第 4.3 小节的模型构造方法，我们在程序验证工具 Ceagle[172] 中实现了 C 程序终止性自动验证工具 Ceagle-Term。该工具接受 C 语言程序输入，并对其终止性进行自动验证。

4.4.1　工具组成

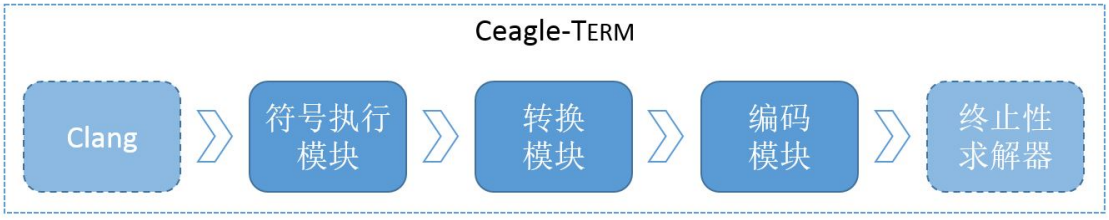


图 4.2　Ceagle-Term 组成

如图 4.2 所示，Ceagle-Term 主要由三部分构成：符号执行模块，转换模块和编码模块。前端接入第三方工具 Clang，将目标 C 程序编译成 LLVM IR 程序。符号执行模块接受 LLVM IR 代码输入，根据第 4.3.1 小节定义的符号执行规则及执行图构造策略，生成能够描述代码行为的符号执行图。根据第 4.3.2 小节所述方法，该符号执行图经转换模块转换成对应的整数变迁系统。基于第 4.3.3 小节，编码模块将变迁系统编码成适合后端终止性求解器验证的整数重写模型。后端接入第三

方重写模型终止性求解器（目前支持开源求解器 KITTeL[143]），求解输出终止性验证结果。

Ceagle-Term 的核心模块为其符号执行模块，它需要对每个抽象程序状态进行单步符号执行，以及对整个符号执行图进行构造。单步符号执行的具体规则已在第 4.3.1 小节列出，在具体的系统实现时，我们利用 Z3[173] 对符号执行规则中的约束进行求解。构建符号执行图的一般策略也已经进行介绍，其具体算法如算法 2 所示。

例 4.1： 给定以下 C 程序：

```
int main() {
  int i = 0;
  while (i < 10) i++;
  return 0;
}
```

Ceagle-Term 生成的符号执行图如图 4.3 所示①，其中包含一对精化类型的边（绿色），以及两条抽象类型的边（红色）。Ceagle-Term 对该程序给出的验证结果是“YES”，即具有终止性。

4.4.2　工具评估

本小节我们将对 Ceagle-Term 的验证能力进行评估。

评估测试集采用终止性问题数据库 TPDB[174]（Termination Problems Data Base）中的 C 程序分支。TPDB 作为每年一度的终止性竞赛[175]（Termination Competition）的竞赛题库，收录了许多经典且具有挑战性的终止性问题。本次测试选取其中两个以 C 程序为输入的子分支——C 程序分支与 C 整数程序分支，其区别在于 C 整数程序分支只包含 C 语言中涉及整数操作的子集。由于 Ceagle-Term 目前只实现了终止性验证技术，还无法证明程序不终止②，因此本次评估只选择 TPDB 中具有终止性的测试程序：C 整数程序分支含此类测试程序 136 个，C 程序分支含 168 个。

该评估运行在具有 24 核 CPU 型号为 Intel Xeon E5-2620 v3@2.40GHz 的计算机上，操作系统为 64 位 Ubuntu-16.04；使用 BenchExec[176] 作为评估运行平台，限制使用核心数为 8、内存为 15 GB，验证超时时间为 300s。

① 由于状态中信息过多，在此不详细展示。
② 程序不终止，指的是存在一条运行轨迹使程序无法终止。

Input: G：只包含初始状态顶点的符号执行图
Input: $path$：存放顶点的栈结构，初始状态只包含初始顶点
Output: G：构建完成的符号执行图
while $path$ 非空 **do**
    $v = peek(path)$;
    if $v.visited == false$ **then**
        if $v == END$ **then** // 如果 v 是结束状态
            $v.visited = true$;　　$pop(path)$;
            continue;
        end
        在 $path$ 中寻找程序位置与 v 相同的顶点，组成列表 $list_v$;
        $should = false$;　　// 判断是否应该进行抽象
        foreach $u \in list_v$ **do**
            if u 和 v 满足应该抽象的判断条件 **then**
                $should = true$;　　$target = u$;
                break;
            end
        end
        if $should$ **then**
            应用算法 3 进行抽象;
        else
            应用算法 4 进行符号执行;
        end
    else // v 已被访问过
        在 v 的后继顶点中寻找一个未被访问的 w;
        if w 不存在 **then**
            $pop(path)$;　　// 将 v 从栈中取出
        else
            $push(w, path)$;
        end
    end
end

算法 **2:** 构建符号执行图

Input: G：正在构建的符号执行图
Input: v：正在访问的顶点
Input: $target$：v 的抽象目标顶点
Input: $path$：当前搜索路径
Output: $G, path$
if 状态 $target$ 是 v 的抽象 **then**
    $v.visited = true$;
    在 G 中添加从 v 到 $target$ 的一条抽象类型的边;
    $pop(path)$;　　// 将 v 从栈中取出
else // 需要计算 v 和 $target$ 的抽象
    对 $path$ 进行出栈操作，直到 $target$ 成为栈顶元素;
    在 G 中删除 $target$ 的后继顶点;
    计算 v 和 $target$ 的抽象状态 c，并将其加入 G;
    if $path$ 中只含 $target$ 一个元素 **then**
        在 G 中添加从 $target$ 到 c 的一条抽象类型的边;
        $push(c, path)$;
    else
        $pop(path)$;　　// 将 $target$ 从栈中取出
        $p = peek(path)$;
        if 在 G 中存在从 p 到 $target$ 的抽象类型的边 **then**
            从 G 中删除顶点 $target$ 及其边;
            在 G 中添加从 p 到 c 的一条抽象类型的边;
            $push(c, path)$;
        else
            在 G 中添加从 $target$ 到 c 的一条抽象类型的边;
            $push(target, path)$;
            $push(c, path)$;
        end
    end
end

算法 **3:** 状态抽象

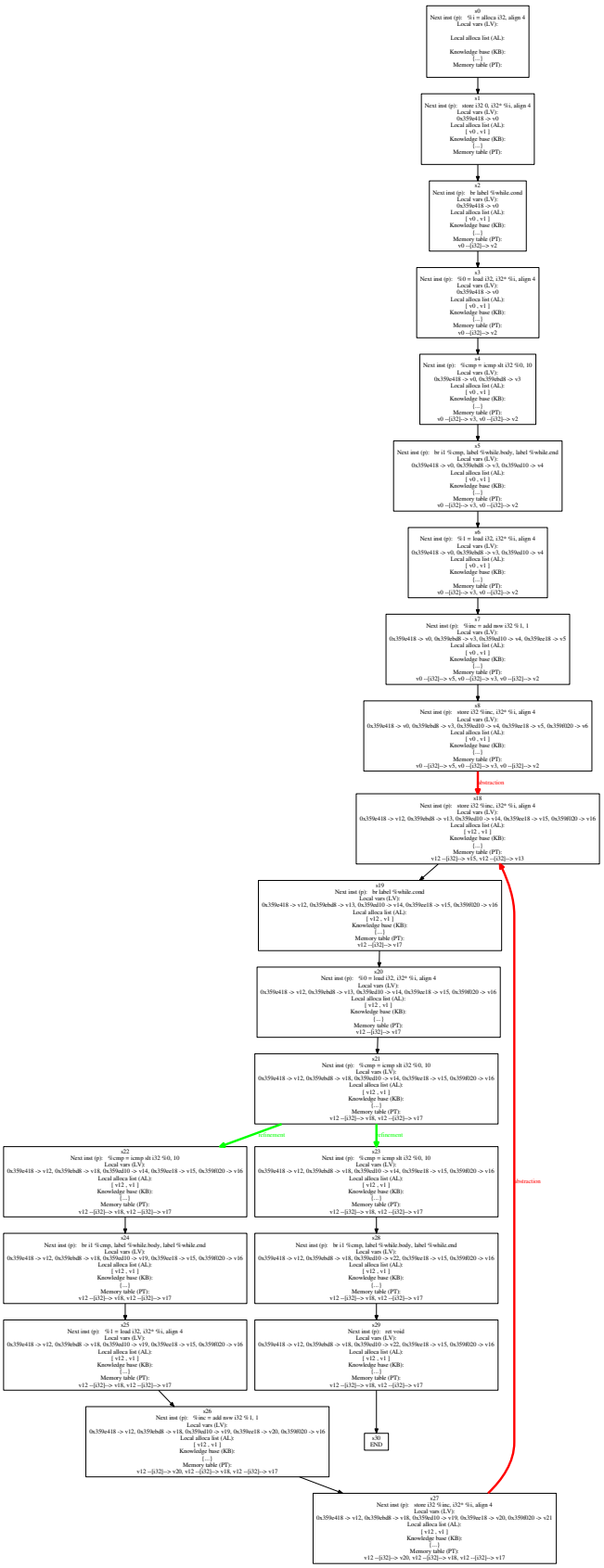


图 4.3　例 4.1 中 C 程序经 Ceagle-Term 生成的符号执行图

Input: G：正在构建的符号执行图
Input: v：正在访问的顶点
Input: $path$：当前搜索路径
Output: $G, path$
$v.visited = true$;
对状态 v 应用除抽象外的符号执行规则，获得新状态列表 $list_{new}$;
if $size(list_{new}) == 1$ **then** // 没有对 v 应用精化规则
    $w = list_{new}[0]$;
    在 G 中添加顶点 w 以及从 v 到 w 的一条求值类型的边;
    $push(w, path)$;
else // 精化规则被应用，产生 2 个新状态
    $w_1 = list_{new}[0]$;
    $w_2 = list_{new}[1]$;
    在 G 中添加顶点 w_1 以及从 v 到 w_1 的一条精化类型的边;
    在 G 中添加顶点 w_2 以及从 v 到 w_2 的一条精化类型的边;
    $push(w_1, path)$;
end

算法 **4:** 求值与精化

在评估中与 Ceagle-Term 进行对比的工具有 AProVE[148,162]、Ctrl[149] 和 KITTeL[143]，均为基于重写技术的 C 程序终止性验证工具，且这些工具目前仍存在开发者进行开发与维护。

表 4.12　Ceagle-Term 与相关工具在 TPDB 上的测试结果

验证结果 / 工具	C 整数程序分支			C 程序分支		
	YES	TIMEOUT	MAYBE	YES	TIMEOUT	MAYBE
AProVE	123	12	1	128	33	7
Ctrl	0	0	136	2	0	166
KITTeL	104	29	3	74	49	45
Ceagle-Term	59	56	21	77	30	61

表 4.12 展示了本次评估的运行结果。“YES”表示工具在该分支中验证成功的测试程序数量；“TIMEOUT”表示工具的验证过程无法在超时时间（300s）内给出验证结果；“MAYBE”表示该工具在时限内运行结束，但无法判断目标程序的终止性。从该表中可以看出，虽然 Ceagle-Term 的后端采用与 KITTeL 相同的重写

终止性求解器，但对于 C 整数程序，Ceagle-Term 的正确率却不如 KITTeL。其原因在于，相较于 KITTeL，Ceagle-Term 建模得到的重写模型带有大量在符号执行过程中产生的额外信息，使得模型规模变大，因此在传递到后端进行求解时容易超时。而另一方面，由于 KITTeL 的前端（llvm2KITTeL）无法像 Ceagle-Term 一样对内存操作进行建模，因此针对 C 程序分支，虽然模型变大引起的超时问题仍然存在，但 Ceagle-Term 的正确率仍然比 KITTeL 高。另外，AProVE 在两个分支都取得了最高的正确率（90.4% 与 76.2%）。同时注意到 Ctrl 在两个分支的正确率都接近 0，原因在于其前端 c2lctrs[163] 对 C 语言语法的限制非常严格，比如无法支持带副作用的语句；另一方面，我们发现，Ctrl 的前端输出的重写模型，需要手动进行调整才能输入后端进行终止性验证，使用起来十分不便。

那么，表 4.12 的数据是否说明了 AProVE 比其它工具都好呢？是否说明 Ceagle-Term 可以被其它工具替代，不具有实用价值呢？下面我们从另一个角度来分析这些数据。

表 4.13　Ceagle-Term 与相关工具的相对验证能力

对比数据 / 工具	C 整数程序分支				C 程序分支			
	△	□	✓	✗	△	□	✓	✗
AProVE	1	65	58	12	9	60	68	31
Ctrl	59	0	0	77	75	0	2	91
KITTeL	3	48	56	29	29	26	48	65

表 4.13 展示了 Ceagle-Term 与其它工具的验证能力比较。“△”表示在该分支中，Ceagle-Term 能成功验证但该工具无法验证的程序数量；“□”表示 Ceagle-Term 无法验证但该工具可以成功验证的程序数量；“✓”表示 Ceagle-Term 与该工具都能成功验证的程序数量；“✗”表示两者都无法验证的程序数量。从该表可以看出，虽然 Ceagle-Term 在两个分支的验证成功率都比 AProVE 低，但这不代表 AProVE 可以替代 Ceagle-Term，仍然存在测试程序可以被 Ceagle-Term 验证但无法被 AProVE 验证：在 C 整数程序分支存在 1 个，在 C 程序分支存在 9 个。类似地，在评估测试集中共有 32 个测试程序可以被 Ceagle-Term 验证但无法被 KITTeL 验证。同理也存在无法被 Ceagle-Term 验证但可以被 AProVE 或 KITTeL 验证的程序。这说明 Ceagle-Term 与其它工具存在互补关系。另外需要指出的是，在 C 整数程序分支及 C 程序分支中，分别存在 9 个和 21 个测试程序无法被此次评估中的任一工具所验证。

4.4.3　可扩展性

由于 Ceagle-Term 的核心模块——符号执行模块是针对 LLVM IR 实现的，得益于 LLVM 一般化的框架及丰富的工具集，通过接入不同的前端，Ceagle-Term 可以支持更多语言的程序终止性验证。比如利用 GCC 搭配 DragonEgg[177] 作为前端，则可以实现对 Ada 语言[178] 和 Fortran 语言[179] 程序的终止性验证。

从后端的角度看，由于 Ceagle-Term 输出的模型及模型格式由编码模块决定，因此通过对编码模块进行扩展，可以实现后端对更多重写模型终止性求解器的支持，比如 Ctrl[149] 和 T$_2$[150]。

4.5　本章小结

整数重写模型是规范化条件重写模型的一种特例。基于整数重写模型和符号执行图，我们开发了针对 C 语言程序终止性的自动验证工具 Ceagle-Term。该工具接受 C 语言程序输入，并自动对其进行模型构建与终止性求解，不需要人工参与，有效降低了形式化验证方法的使用成本，且为软件的完全正确性提供了必要的工具支持。

第 5 章　结束语

5.1　工作总结

针对现代嵌入式系统结构复杂、行为复杂所引发的形式化建模与验证问题，本文基于重写理论，完成了以下工作：

1. 针对嵌入式系统中硬件的并发行为与软件的顺序行为并存的异构性特征进行研究，提出了规范化条件重写模型。将硬件的并发行为抽象成不确定性行为，将软件的顺序行为抽象成确定性行为，基于模重写模型、条件重写模型以及 Nipkow 在高阶重写中使用的 $\beta\eta$ 规范化过程，本文提出的规范化条件重写模型，利用等式规则描述系统的结构特征及状态等价关系，利用条件约束描述系统的控制流特征，利用化简规则描述系统的确定性行为，利用重写规则描述系统的不确定性行为。本文对重写模型的规则应用策略进行扩展，定义了在多种规则共存的情况下，确定性行为与不确定性行为的协作方式，且保证了重写规则触发时条件约束的可判定性。规范化条件重写模型具有严格的形式化语法及语义定义。该模型是面向建模与验证的重写模型扩展，相比其它重写模型扩展，它的表达能力得到了提升。
2. 基于规范化条件重写模型，以嵌入式系统建模方法论为切入点就重写模型易用性低的问题进行研究。本文提出对嵌入式系统的结构层次性、行为异构性、结构动态性和实时性等特征的具体建模方法，对系统建模过程进行指导。基于语义映射的方式，将该建模方法在工具 Maude 中进行实现。通过对两个真实嵌入式系统的建模验证过程予以应用，验证了该建模方法在嵌入式系统中实际应用的可行性。其中，机车优化控制系统在本文的建模与验证过程中，成功定位了测试人员进行仿真测试未能发现的系统缺陷；该系统目前运行稳定，满足了铁路节能驾驶使用要求，并在沈阳铁路局通过了实车运用考核。速率单调调度系统的可调度性和正确性通过模型检测方法得到了验证，且本文证明了该结果具有可靠性和完备性；该调度系统目前在某工业级航天控制器中在线运行。
3. 基于整数重写模型这一规范化条件重写模型实例，针对重写模型易用性低的问题，以嵌入式系统软件的自动建模为切入点进行研究。本文设计、开发了针对 C 语言程序终止性的自动验证工具 Ceagle-Term。为增加工具的可扩展性，Ceagle-Term 采用 LLVM IR 作为程序的中间表示语言。参考已有工具，Ceagle-Term 采用符号执行技术生成符号执行图作为程序行为的中间模

型。它的后端可以接入多种重写模型终止性求解器，进一步提高了它的可扩展性。Ceagle-Term 接受 C 语言程序输入，自动生成终止性验证结果，有效降低了形式化验证方法的使用成本，且为软件的完全正确性提供了必要的支持。

5.2　研究展望

在本文的工作基础上，为了进一步推动规范化条件重写模型在嵌入式系统建模与验证工作中的应用，拟在以下方面开展更多研究工作：

1. 对条件模重写模型的终止性与合流性开展研究。在规范化条件重写模型 $\langle \mathcal{R}, \mathcal{S}, \mathcal{E} \rangle$ 的定义中，$\mathcal{S}$ 和 $\mathcal{E}$ 被要求使得条件模重写模型 $\mathcal{S}_{\mathcal{E}}$ 满足终止性和合流性。从建模的角度看，根据第 3.3 小节提出的建模方法，$\mathcal{S}$ 用于建模系统的确定性行为（如软件的局部顺序行为），$\mathcal{E}$ 用于描述项表达式的结构信息，因此 $\mathcal{S}_{\mathcal{E}}$ 的终止性和合流性可以由建模的方法得到保证。然而建模过程是由人主导的，人总是容易出错，这正是为什么需要对系统进行验证的原因。从理论和工具的角度对 $\mathcal{S}_{\mathcal{E}}$ 的终止性和合流性进行检查，不但可以保证模型的一致性（consistency），还可以发现建模过程引入的错误。虽然目前存在对模重写模型的终止性和合流性的研究[90–95]，但目前的结果对等价模型 $\mathcal{E}$ 具有较严格的约束。因此，要推动规范化条件重写模型在建模过程中的应用，需要对模重写模型的终止性和合流性进行更一般化的理论研究，以及相应检查工具的开发。
2. 进一步提高和完善基于规范化条件重写模型的建模方法。现代嵌入式系统种类繁多、行为复杂，本文提出的基于规范化条件重写模型的建模方法，需要在更多实际案例上进行应用，才能发现它在应对不同系统时产生的问题，对其进一步提高和完善。对该建模方法的另一种可行的完善方式，是针对不同的特定嵌入式系统，如中断系统、调度系统等，设计领域特定语言（Domain Specific Language，DSL）来对特定系统进行建模，并将 DSL 模型翻译成对应的规范化条件重写模型。
3. 完善规范化条件重写模型的支持工具集。如第 3.4.2 小节所述，工具 Maude 不能完全支持规范化条件重写模型的建模，其根本原因在于 Maude 的理论模型——重写逻辑的表达能力不如规范化条件重写模型。因此，对 Maude 的底层核心模块进行扩展，或开发专门针对规范化条件重写模型的建模验证工具集，可以更充分地发挥规范化条件重写模型的作用。
4. 提高和完善 Ceagle-Term 工具。一方面，由于 Ceagle-Term 采用符号执行技

术对模型进行构建，这导致在大规模程序上进行应用时要耗费大量的计算资源。因此，为了降低模型构建的资源占用问题，也为了减小模型规模，可以考虑使用诸如切片[180] 等程序分析变形技术，在构建模型前对程序中与终止性无关的代码进行削减。另一方面，在构造程序的符号执行图时，抽象规则会导致状态的信息丢失。而在丢失的状态信息中，可能存在某些对终止性判定有用的信息，比如循环不变式[137]。设计新的抽象规则，使更多的有用信息得到保留，同时尽量减少不必要的信息，是一个值得研究的方向。再者，虽然目前的内存模型以及对函数调用的执行规则可以保证符号执行图的构造过程在递归函数存在时得以终止，但这并没有完全解决对递归函数的建模问题。如何对递归函数的语义进行精确建模，这是一个从理论和应用层面都值得研究的问题。最后，将不同的终止性验证技术、非终止性验证技术与 Ceagle-Term 进行融合，是完善 Ceagle-Term 工具的另一个方向。

参考文献

[1] Heath S. Embedded systems design. Newnes, 2002.

[2] Guidelines for development of civil aircraft and systems. SAE International, December, 2010.

[3] Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment. SAE International, December, 1996.

[4] Clarke E M, Grumberg O, Peled D. Model checking. MIT press, 1999.

[5] Chang C, Lee R C T. Symbolic logic and mechanical theorem proving. Computer science classics, Academic Press, 1973.

[6] Koopman P. Embedded system design issues (the rest of the story). 1996 International Conference on Computer Design (ICCD '96), VLSI in Computers and Processors, October 7-9, 1996, Austin, TX, USA, Proceedings. IEEE Computer Society, 1996. 310–317.

[7] Shaw A C. Real-time systems and software. Wiley, 2001.

[8] Gerndt R, Ernst R. An event-driven multi-threading architecture for embedded systems. Proceedings of the Fifth International Workshop on Hardware/Software Codesign, CODES/CASHE 1997, March 24-26, 1997, Braunschweig, Germany. IEEE Computer Society, 1997. 29–34.

[9] White R L. Reconfigurable embedded control systems: applications for flexibility and agility by mohamed khalgui and han-michale hanisch. ACM SIGSOFT Software Engineering Notes, 2011, 36(5):53.

[10] Li X, Hanxleden R. Multithreaded reactive programming - the Kiel Esterel processor. IEEE Trans. Computers, 2012, 61(3):337–349.

[11] Valmari A. The state explosion problem. In: Reisig W, Rozenberg G, (eds.). Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets, held in Dagstuhl, September 1996, volume 1491 of *Lecture Notes in Computer Science*. Springer, 1996. 429–528.

[12] Woodcock J, Larsen P G, Bicarregui J, et al. Formal methods: Practice and experience. ACM Comput. Surv., 2009, 41(4):19:1–19:36.

[13] Bjørner D, Jones C B, (eds.). The Vienna Development Method: The Meta-Language, volume 61 of *Lecture Notes in Computer Science*. Springer, 1978.

[14] Spivey J M. The Z notation - a reference manual. Prentice Hall International Series in Computer Science, Prentice Hall, 1989.

[15] Murata T. Petri nets: Properties, analysis and applications. Proceedings of the IEEE, 1989, 77(4):541–580.

[16] Hopcroft J E, Ullman J D. Introduction to Automata Theory, Languages and Computation. Addison-Wesley, 1979.

[17] Minsky M L. Computation: finite and infinite machines. Prentice-Hall, Inc., 1967.

[18] Harel D. Statecharts: A visual formalism for complex systems. Sci. Comput. Program., 1987, 8(3):231–274.

[19] Alur R, Dill D L. A theory of timed automata. Theor. Comput. Sci., 1994, 126(2):183–235.

[20] Alur R. Timed automata. In: Halbwachs N, Peled D A, (eds.). Computer Aided Verification, 11th International Conference, CAV '99, Trento, Italy, July 6-10, 1999, Proceedings, volume 1633 of *Lecture Notes in Computer Science*. Springer, 1999. 8–22.

[21] Larsen K G, Pettersson P. Uppaal2k. Bulletin of the European Association for Theoretical Computer Science, 1999..

[22] Larsen K G, Pettersson P, Yi W. UPPAAL in a nutshell. STTT, 1997, 1(1-2):134–152.

[23] Boucheneb H, Gardey G, Roux O H. TCTL model checking of time Petri nets. J. Log. Comput., 2009, 19(6):1509–1540.

[24] Kheldoun A, Barkaoui K, Ioualalen M. Formal verification of complex business processes based on high-level petri nets. Inf. Sci., 2017, 385:39–54.

[25] Xu N, Peng S, Wang Z. Verifying soundness of geodata web service composition based on Petri nets. J. Web Eng., 2017, 16(1&2):145–160.

[26] Zhang X, Ansari J, Yang G, et al. TRUMP: efficient and flexible realization of medium access control protocols for wireless networks. IEEE Trans. Mob. Comput., 2016, 15(10):2614–2626.

[27] Salah H B, Benzina A, Khalgui M. Petri nets-based design of real-time reconfigurable networks on chips. In: Ito T, Kim Y, Fukuta N, (eds.). 14th IEEE/ACIS International Conference on Computer and Information Science, ICIS 2015, Las Vegas, NV, USA, June 28 - July 1, 2015. IEEE Computer Society, 2015. 597–604.

[28] Jensen K. Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use - Volume 3. Monographs in Theoretical Computer Science. An EATCS Series, Springer, 1997.

[29] Lakos C. From coloured Petri nets to object Petri nets. In: Michelis G D, Diaz M, (eds.). Application and Theory of Petri Nets 1995, 16th International Conference, Turin, Italy, June 26-30, 1995, Proceedings, volume 935 of *Lecture Notes in Computer Science*. Springer, 1995. 278–297.

[30] David R, Alla H. Discrete, continuous, and hybrid Petri nets. Springer Science & Business Media, 2010.

[31] Berthomieu B, Diaz M. Modeling and verification of time dependent systems using time Petri nets. IEEE Trans. Software Eng., 1991, 17(3):259–273.

[32] Zuberek W M. Timed Petri nets and preliminary performance evaluation. In: Lenfant J, Borgerson B R, Atkins D E, et al., (eds.). Proceedings of the 7th Annual Symposium on Computer Architecture, May 1980. ACM, 1980. 88–96.

[33] Tsai J J P, Yang S J, Chang Y. Timing constraint Petri nets and their application to schedulability analysis of real-time system specifications. IEEE Trans. Software Eng., 1995, 21(1):32–49.

[34] Berthomieu B, Vernadat F. Time Petri nets analysis with TINA. Third International Conference on the Quantitative Evaluation of Systems (QEST 2006), 11-14 September 2006, Riverside, California, USA. IEEE Computer Society, 2006. 123–124.

[35] Vardi M Y. An automata-theoretic approach to linear temporal logic. In: Moller F, Birtwistle G M, (eds.). Logics for Concurrency - Structure versus Automata (8th Banff Higher Order Workshop, August 27 - September 3, 1995, Proceedings), volume 1043 of *Lecture Notes in Computer Science*. Springer, 1995. 238–266.

[36] Gardey G, Lime D, Magnin M, et al. Romeo: A tool for analyzing time Petri nets. In: Etessami K, Rajamani S K, (eds.). Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings, volume 3576 of *Lecture Notes in Computer Science*. Springer, 2005. 418–423.

[37] Jensen K, Kristensen L M, Wells L. Coloured Petri nets and CPN tools for modelling and validation of concurrent systems. STTT, 2007, 9(3-4):213–254.

[38] Syme D. Reasoning with the formal definition of standard ML in HOL. In: Joyce J J, Seger C H, (eds.). Higher Order Logic Theorem Proving and its Applications, 6th International Workshop, HUG '93, Vancouver, BC, Canada, August 11-13, 1993, Proceedings, volume 780 of *Lecture Notes in Computer Science*. Springer, 1993. 43–60.

[39] Andrews P B. An introduction to mathematical logic and type theory - to truth through proof. Computer science and applied mathematics, Academic Press, 1986.

[40] Parent C. Synthesizing proofs from programs in the calculus of inductive constructions. In: Möller B, (eds.). Mathematics of Program Construction, MPC'95, Kloster Irsee, Germany, July 17-21, 1995, Proceedings, volume 947 of *Lecture Notes in Computer Science*. Springer, 1995. 351–379.

[41] Van Benthem J, Doets K. Higher-order logic. Handbook of philosophical logic. Springer, 1983: 275–329.

[42] Bertot Y, Castéran P. Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series, Springer, 2004.

[43] Nipkow T, Paulson L C, Wenzel M. Isabelle/HOL - A Proof Assistant for Higher-Order Logic, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[44] Gonthier G. The four colour theorem: Engineering of a formal proof. In: Kapur D, (eds.). Computer Mathematics, 8th Asian Symposium, ASCM 2007, Singapore, December 15-17, 2007. Revised and Invited Papers, volume 5081 of *Lecture Notes in Computer Science*. Springer, 2007. 333.

[45] Paulson L C. A mechanised proof of Gödel's incompleteness theorems using nominal Isabelle. J. Autom. Reasoning, 2015, 55(1):1–37.

[46] Leroy X. Formal verification of a realistic compiler. Commun. ACM, 2009, 52(7):107–115.

[47] Stewart G, Beringer L, Cuellar S, et al. Compositional CompCert. In: Rajamani S K, Walker D, (eds.). Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015. ACM, 2015. 275–287.

[48] Klein G, Elphinstone K, Heiser G, et al. seL4: formal verification of an OS kernel. Proc. ACM Symposium on Operating Systems Principles (SOSP '09), 2009. 207–220.

[49] Gu L, Vaynberg A, Ford B, et al. CertiKOS: a certified kernel for secure cloud computing. In: Chen H, Zhang Z, Moon S, et al., (eds.). APSys '11 Asia Pacific Workshop on Systems, Shanghai, China, July 11-12, 2011. ACM, 2011. 3.

[50] Gu R, Shao Z, Chen H, et al. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In: Keeton K, Roscoe T, (eds.). 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016. USENIX Association, 2016. 653–669.

[51] Dershowitz N, Jouannaud J. Rewrite systems. Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B). 1990: 243–320.

[52] Terese. Term rewriting systems. Cambridge Tracts in Theoretical Computer Science (Jan Willem Klop et al editors). Cambridge University Press, 2003:.

[53] Peterson G E, Stickel M E. Complete sets of reductions for some equational theories. J. ACM, 1981, 28(2):233–264.

[54] Gramlich B. On termination and confluence of conditional rewrite systems. In: Dershowitz N, Lindenstrauss N, (eds.). Conditional and Typed Rewriting Systems, 4th International Workshop, CTRS-94, Jerusalem, Israel, July 13-15, 1994, Proceedings, volume 968 of *Lecture Notes in Computer Science*. Springer, 1994. 166–185.

[55] Nikhil R S. Bluespec: A general-purpose approach to high-level synthesis based on parallel atomic transactions. High-Level Synthesis. Springer, 2008: 129–146.

[56] Thompson S J. Haskell - The Craft of Functional Programming, 3rd Edition. Addison-Wesley, 2011.

[57] Nikhil R S. Bluespec System Verilog: efficient, correct RTL from high level specifications. 2nd ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2004), 23-25 June 2004, San Diego, California, USA, Proceedings. IEEE, 2004. 69–70.

[58] Singh G, Shukla S K. Model checking Bluespec specified hardware designs. In: Abadir M S, Wang L, Bhadra J, (eds.). Eighth International Workshop on Microprocessor Test and Verification (MTV 2007), Common Challenges and Solutions, 5-6 December 2007, Austin, Texas, USA. IEEE Computer Society, 2007. 39–43.

[59] Martí-Oliet N, Meseguer J. Rewriting logic: roadmap and bibliography. Theor. Comput. Sci., 2002, 285(2):121–154.

[60] Meseguer J. Twenty years of rewriting logic. J. Log. Algebr. Program., 2012, 81(7):721–781.

[61] Clavel M, Durán F, Eker S, et al. Maude: specification and programming in rewriting logic. Theor. Comput. Sci., 2002, 285(2):187–243.

[62] Ölveczky P C, Meseguer J. Semantics and pragmatics of Real-Time Maude. Higher-Order and Symbolic Computation, 2007, 20(1):161–196.

[63] Clavel M, Palomino M, Riesco A. Introducing the ITP tool: a tutorial. J. UCS, 2006, 12(11):1618–1650.

[64] Meseguer J, Rosu G. The rewriting logic semantics project: A progress report. Inf. Comput., 2013, 231:38–69.

[65] Nipkow T. Higher-order critical pairs. In *Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15-18, 1991*[181], 342–349.

[66] Jouannaud J, Okada M. A computation model for executable higher-order algebraic specification languages. In *Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15-18, 1991*[181], 350–361.

[67] Hughes J. Why functional programming matters. Comput. J., 1989, 32(2):98–107.

[68] Assaf A, Dowek G, Jouannaud J P, et al. Encoding Proofs in Dedukti: the case of Coq proofs. Proceedings Hammers for Type Theories, Coimbra, Portugal: Easy Chair, 2016.

[69] Book R V. Thue systems as rewriting systems. J. Symb. Comput., 1987, 3(1/2):39–68.

[70] Raoult J. On graph rewritings. Theor. Comput. Sci., 1984, 32:1–24.

[71] Lankford D S, Ballantyne A M. Decision procedures for simple equational theories with commutative-associative axioms: Complete sets of commutative-associative reductions. Memo ATP-39, University of Texas, Austin, August, 1977.

[72] Jouannaud J, Li J. Church-Rosser properties of normal rewriting. In: Cégielski P, Durand A, (eds.). Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France, volume 16 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012. 350–365.

[73] Kaye R. Models of Peano Arithmetic. Clarendon Press, 1991.

[74] Dauchet M. Simulation of turning machines by a left-linear rewrite rule. In Dershowitz N[182], 109–120.

[75] Huet G, Lankford D. On the uniform halting problem for term rewriting systems. Rapport Laboria 283, IRIA, Rocquencourt, France, 1978.

[76] Hsiang J, (eds.). Rewriting Techniques and Applications, 6th International Conference, RTA-95, Kaiserslautern, Germany, April 5-7, 1995, Proceedings, volume 914 of *Lecture Notes in Computer Science*. Springer, 1995.

[77] Geser A, Hofbauer D, Waldmann J, et al. On tree automata that certify termination of left-linear term rewriting systems. Inf. Comput., 2007, 205(4):512–534.

[78] Payet É. Loop detection in term rewriting using the eliminating unfoldings. Theor. Comput. Sci., 2008, 403(2-3):307–327.

[79] Lescanne P. On termination of one rule rewrite systems. Theor. Comput. Sci., 1994, 132(2):395–401.

[80] Jouannaud J, Li J. Termination of dependently typed rewrite rules. In: Altenkirch T, (eds.). 13th International Conference on Typed Lambda Calculi and Applications, TLCA 2015, July 1-3, 2015, Warsaw, Poland, volume 38 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015. 257–272.

[81] Jacquemard F. Reachability and confluence are undecidable for flat term rewriting systems. Inf. Process. Lett., 2003, 87(5):265–270.

[82] Newman M H A. On theories with a combinatorial definition of 'equivalence'. Ann. Math., 1942, 43(2):223–243.

[83] Hindley J R. The Church-Rosser property and a result in combinatory logic[D]. University of Newcastle upon Tyne, 1964.

[84] Rosen B K. Tree-manipulating systems and Church-Rosser theorems. J. ACM, 1973, 20(1):160–187.

[85] Knuth D E, Bendix P B. Simple word problems in universal algebras. In: Leech J, (eds.). Computational Problems in Abstract Algebra. Elsevier, 1970: 263–297.

[86] Oostrom V. Confluence by decreasing diagrams converted. In: Voronkov A, (eds.). RTA, volume 5117 of *LNCS*. Springer, 2008. 306–320.

[87] Zankl H, Felgenhauer B, Middeldorp A. Labelings for decreasing diagrams. J. Autom. Reasoning, 2015, 54(2):101–133.

[88] Liu J, Jouannaud J, Ogawa M. Confluence of layered rewrite systems. In: Kreutzer S, (eds.). 24th EACSL Annual Conference on Computer Science Logic, CSL 2015, September 7-10, 2015, Berlin, Germany, volume 41 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015. 423–440.

[89] Liu J, Dershowitz N, Jouannaud J. Confluence by critical pair analysis. In: Dowek G, (eds.). Rewriting and Typed Lambda Calculi - Joint International Conference, RTA-TLCA 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings, volume 8560 of *Lecture Notes in Computer Science*. Springer, 2014. 287–302.

[90] Jouannaud J, Muñoz M. Termination of a set of rules modulo a set of equations. In: Shostak R E, (eds.). 7th International Conference on Automated Deduction, Napa, California, USA, May 14-16, 1984, Proceedings, volume 170 of *Lecture Notes in Computer Science*. Springer, 1984. 175–193.

[91] Jouannaud J, Marché C. Termination and completion modulo associativity, commutativity and identity. Theor. Comput. Sci., 1992, 104(1):29–51.

[92] Jouannaud J, Toyama Y. Modular Church-Rosser modulo: The complete picture. Int. J. Software and Informatics, 2008, 2(1):61–75.

[93] Jouannaud J. Modular Church-Rosser modulo. In: Pfenning F, (eds.). Term Rewriting and Applications, 17th International Conference, RTA 2006, Seattle, WA, USA, August 12-14, 2006, Proceedings, volume 4098 of *Lecture Notes in Computer Science*. Springer, 2006. 96–107.

[94] Jouannaud J, Liu J. From diagrammatic confluence to modularity. Theoretical Computer Science, 2012, 464:20–34.

[95] Jouannaud J, Kirchner H. Completion of a set of rules modulo a set of equations. SIAM J. Comput., 1986, 15(4):1155–1194.

[96] Brand D, Darringer J A, Joyner W H. Completeness of conditional reductions. IBM Research Division, 1978.

[97] Bergstra J A, Klop J W. Conditional rewrite rules: Confluence and termination. J. Comput. Syst. Sci., 1986, 32(3):323–362.

[98] Dershowitz N, Okada M, Sivakumar G. Canonical conditional rewrite systems. In: Lusk E L, Overbeek R A, (eds.). 9th International Conference on Automated Deduction, Argonne, Illinois, USA, May 23-26, 1988, Proceedings, volume 310 of *Lecture Notes in Computer Science*. Springer, 1988. 538–549.

[99] Dershowitz N, Okada M. A rationale for conditional equational programming. Theor. Comput. Sci., 1990, 75(1&2):111–138.

[100] Bertling H, Ganzinger H. Completion-time optimization of rewrite-time goal solving. In Dershowitz N[182], 45–58.

[101] Nadel A. Bit-vector rewriting with automatic rule generation. In Biere A and Bloem R[183], 663–679.

[102] Holzmann G J. Design and validation of protocols: A tutorial. Computer Networks and ISDN Systems, 1993, 25(9):981–1017.

[103] Holzmann G J. The model checker SPIN. IEEE Trans. Softw. Eng., 1997, 23(5):279–295.

[104] Braibant T, Chlipala A. Formal verification of hardware synthesis. In: Sharygina N, Veith H, (eds.). Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings, volume 8044 of *Lecture Notes in Computer Science*. Springer, 2013. 213–228.

[105] Clavel M, Durán F, Eker S, et al., (eds.). All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[106] Bae K, Krisiloff J, Meseguer J, et al. Designing and verifying distributed cyber-physical systems using multirate PALS: an airplane turning control system case study. Sci. Comput. Program., 2015, 103:13–50.

[107] Huet G P. Confluent reductions: Abstract properties and applications to term rewriting systems. J. ACM, 1980, 27(4):797–821.

[108] Goguen J A, Meseguer J. Order-sorted algebra I: equational deduction for multiple inheritance, overloading, exceptions and partial operations. Theor. Comput. Sci., 1992, 105(2):217–273.

[109] Bouhoula A, Jouannaud J, Meseguer J. Specification and proof in membership equational logic. Theor. Comput. Sci., 2000, 236(1-2):35–132.

[110] Eker S, Meseguer J, Sridharanarayanan A. The Maude LTL model checker. Electr. Notes Theor. Comput. Sci., 2002, 71:162–187.

[111] Manna Z, Pnueli A. The temporal logic of reactive and concurrent systems - specification. Springer, 1992.

[112] Huang J, Deng Y, Yang Q, et al. An energy-efficient train control framework for smart railway transportation. IEEE Trans. Computers, 2016, 65(5):1407–1417.

[113] Liu C L, Layland J W. Scheduling algorithms for multiprogramming in a hard-real-time environment. J. ACM, 1973, 20(1):46–61.

[114] Lehoczky J P, Sha L, Strosnider J K. Enhanced aperiodic responsiveness in hard real-time environments. Proc. Real-Time Systems Symposium (RTSS '87), 1987. 261–270.

[115] Sprunt B, Sha L, Lehoczky J P. Aperiodic task scheduling for hard real-time systems. Real-Time Systems, 1989, 1(1):27–60.

[116] Lehoczky J P, Ramos-Thuel S. An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems. Proc. Real-Time Systems Symposium (RTSS '92), 1992. 110–123.

[117] Strosnider J K, Lehoczky J P, Sha L. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. IEEE Trans. Computers, 1995, 44(1):73–91.

[118] Leung J Y, Whitehead J. On the complexity of fixed-priority scheduling of periodic, real-time tasks. Perform. Eval., 1982, 2(4):237–250.

[119] Audsley N C, Burns A, Wellings A J. Deadline monotonic scheduling theory and application. Control Engineering Practice, 1993, 1(1):71–78.

[120] Sha L, Rajkumar R, Lehoczky J P. Priority inheritance protocols: An approach to real-time synchronization. IEEE Trans. Computers, 1990, 39(9):1175–1185.

[121] Dhall S K, Liu C L. On a real-time scheduling problem. Operations Research, 1978, 26(1):127–140.

[122] López J M, García M, Díaz J L, et al. Utilization bounds for multiprocessor rate-monotonic scheduling. Real-Time Systems, 2003, 24(1):5–28.

[123] López J M, Díaz J L, García D F. Minimum and maximum utilization bounds for multiprocessor rate monotonic scheduling. IEEE Trans. Parallel Distrib. Syst., 2004, 15(7):642–653.

[124] Baruah S K, Goossens J. Rate-monotonic scheduling on uniform multiprocessors. IEEE Trans. Comput., 2003, 52(7):966–970.

[125] Oh Y, Son S H. Enhancing fault-tolerance in rate-monotonic scheduling. Real-Time Systems, 1994, 7(3):315–329.

[126] Ghosh S, Melhem R G, Mossé D, et al. Fault-tolerant rate-monotonic scheduling. Real-Time Systems, 1998, 15(2):149–181.

[127] Bertossi A A, Mancini L V, Rossini F. Fault-tolerant rate-monotonic first-fit scheduling in hard-real-time systems. IEEE Trans. Parallel Distrib. Syst., 1999, 10(9):934–945.

[128] Lehoczky J P, Sha L, Ding Y. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. Proc. Real-Time Systems Symposium (RTSS '89), 1989. 166–171.

[129] Kuo T, Mok A K. Load adjustment in adaptive real-time systems. Proc. Real-Time Systems Symposium (RTSS '91), 1991. 160–170.

[130] Bini E, Buttazzo G C, Buttazzo G M. Rate monotonic analysis: The hyperbolic bound. IEEE Trans. Comput., 2003, 52(7):933–942.

[131] Jiang Y, Zhang H, Li Z, et al. Design and optimization of multiclocked embedded systems using formal techniques. IEEE Trans. Ind. Electron., 2015, 62(2):1270–1278.

[132] Tian C, Duan Z. Model checking rate monotonic scheduling algorithm based on propositional projection temporal logic. Journal of Software, 2011, 22(2):211–221. In Chinese.

[133] Cui J, Duan Z, Tian C. Model checking rate-monotonic scheduler with TMSVL. Proc. International Conference on Engineering of Complex Computer Systems (ICECCS '14), 2014. 202–205.

[134] Ölveczky P C, Meseguer J. Abstraction and completeness for Real-Time Maude. Electr. Notes Theor. Comput. Sci., 2007, 176(4):5–27.

[135] Katcher D I, Arakawa H, Strosnider J K. Engineering and analysis of fixed priority schedulers. IEEE Trans. Softw. Eng., 1993, 19(9):920–934.

[136] Han M, Duan Z, Wang X. Time constraints with temporal logic programming. Formal Methods and Software Engineering - Proc. International Conference on Formal Engineering Methods (ICFEM '12), volume 7635 of *Lecture Notes in Computer Science*, 2012. 266–282.

[137] Hoare C A R. An axiomatic basis for computer programming. Commun. ACM, 1969, 12(10):576–580.

[138] Clarke E M, Emerson E A. Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Kozen D, (eds.). Logics of Programs, Workshop, Yorktown Heights, New York, May 1981, volume 131 of *Lecture Notes in Computer Science*. Springer, 1981. 52–71.

[139] Emerson E A, Halpern J Y. "sometimes" and "not never" revisited: on branching versus linear time temporal logic. J. ACM, 1986, 33(1):151–178.

[140] Emerson E A, Halpern J Y. Decision procedures and expressiveness in the temporal logic of branching time. J. Comput. Syst. Sci., 1985, 30(1):1–24.

[141] Lamport L. Proving the correctness of multiprocess programs. IEEE Trans. Software Eng., 1977, 3(2):125–143.

[142] Brockschmidt M, Cook B, Ishtiaq S, et al. T2: temporal property verification. In Chechik M and Raskin J[184], 387–393.

[143] Falke S, Kapur D, Sinz C. Termination analysis of C programs using compiler intermediate languages. In: Schmidt-Schauß M, (eds.). Proceedings of the 22nd International Conference on Rewriting Techniques and Applications, RTA 2011, May 30 - June 1, 2011, Novi Sad, Serbia, volume 10 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011. 41–50.

[144] Dershowitz N. Termination of rewriting. J. Symb. Comput., 1987, 3(1/2):69–116.

[145] Knuth D E, Bendix P B. Simple word problems in universal algebras. Automation of Reasoning. Springer, 1983: 342–376.

[146] Arts T, Giesl J. Termination of term rewriting using dependency pairs. Theor. Comput. Sci., 2000, 236(1-2):133–178.

[147] Giesl J, Arts T, Ohlebusch E. Modular termination proofs for rewriting using dependency pairs. J. Symb. Comput., 2002, 34(1):21–58.

[148] Giesl J, Thiemann R, Schneider-Kamp P, et al. Automated termination proofs with AProVE. In: Oostrom V, (eds.). Rewriting Techniques and Applications, 15th International Conference, RTA 2004, Aachen, Germany, June 3-5, 2004, Proceedings, volume 3091 of *Lecture Notes in Computer Science*. Springer, 2004. 210–220.

[149] Kop C, Nishida N. Constrained term rewriting tool. In: Davis M, Fehnker A, McIver A, et al., (eds.). Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings, volume 9450 of *Lecture Notes in Computer Science*. Springer, 2015. 549–557.

[150] Korp M, Sternagel C, Zankl H, et al. Tyrolean Termination Tool 2. In: Treinen R, (eds.). Rewriting Techniques and Applications, 20th International Conference, RTA 2009, Brasília, Brazil, June 29 - July 1, 2009, Proceedings, volume 5595 of *Lecture Notes in Computer Science*. Springer, 2009. 295–304.

[151] Podelski A, Rybalchenko A. Transition invariants. 19th IEEE Symposium on Logic in Computer Science (LICS 2004), 14-17 July 2004, Turku, Finland, Proceedings. IEEE Computer Society, 2004. 32–41.

[152] Berdine J, Chawdhary A, Cook B, et al. Variance analyses from invariance analyses. In: Hofmann M, Felleisen M, (eds.). Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007. ACM, 2007. 211–224.

[153] Urban C. The abstract domain of segmented ranking functions. In: Logozzo F, Fähndrich M, (eds.). Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings, volume 7935 of *Lecture Notes in Computer Science*. Springer, 2013. 43–62.

[154] Urban C. FuncTion: An abstract domain functor for termination - (competition contribution). In: Baier C, Tinelli C, (eds.). Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings, volume 9035 of *Lecture Notes in Computer Science*. Springer, 2015. 464–466.

[155] Alias C, Darte A, Feautrier P, et al. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In: Cousot R, Martel M, (eds.). Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings, volume 6337 of *Lecture Notes in Computer Science*. Springer, 2010. 117–133.

[156] Podelski A, Rybalchenko A. ARMC: the logical choice for software model checking with abstraction refinement. In: Hanus M, (eds.). Practical Aspects of Declarative Languages, 9th International Symposium, PADL 2007, Nice, France, January 14-15, 2007., volume 4354 of *Lecture Notes in Computer Science*. Springer, 2007. 245–259.

[157] Kroening D, Sharygina N, Tsitovich A, et al. Termination analysis with compositional transition invariants. In: Touili T, Cook B, Jackson P B, (eds.). Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings, volume 6174 of *Lecture Notes in Computer Science*. Springer, 2010. 89–103.

[158] Tsitovich A, Sharygina N, Wintersteiger C M, et al. Loop summarization and termination analysis. In: Abdulla P A, Leino K R M, (eds.). Tools and Algorithms for the Construction and Analysis of Systems - 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings, volume 6605 of *Lecture Notes in Computer Science*. Springer, 2011. 81–95.

[159] Cook B, Podelski A, Rybalchenko A. Termination proofs for systems code. In: Schwartzbach M I, Ball T, (eds.). Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006. ACM, 2006. 415–426.

[160] Urban C, Gurfinkel A, Kahsai T. Synthesizing ranking functions from bits and pieces. In Chechik M and Raskin J[184], 54–70.

[161] Chen H, David C, Kroening D, et al. Synthesising interprocedural bit-precise termination proofs. In: Cohen M B, Grunske L, Whalen M, (eds.). 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015. IEEE Computer Society, 2015. 53–64.

[162] Giesl J, Brockschmidt M, Emmes F, et al. Proving termination of programs automatically with AProVE. In Demri S et al.[185], 184–191.

[163] Kop C, Nishida N. Converting C to LCTRSs. Available online: http://www.trs.cm.is.nagoya-u.ac.jp/c2lctrs/formal.pdf, December, 2015.

[164] Le T C, Qin S, Chin W. Termination and non-termination specification inference. In: Grove D, Blackburn S, (eds.). Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015. ACM, 2015. 489–498.

[165] Heizmann M, Hoenicke J, Podelski A. Termination analysis by learning terminating programs. In Biere A and Bloem R[183], 797–813.

[166] Ströder T, Giesl J, Brockschmidt M, et al. Automatically proving termination and memory safety for programs with pointer arithmetic. J. Autom. Reasoning, 2017, 58(1):33–65.

[167] Lattner C, Adve V S. LLVM: A compilation framework for lifelong program analysis & transformation. 2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA. IEEE Computer Society, 2004. 75–88.

[168] King J C. Symbolic execution and program testing. Commun. ACM, 1976, 19(7):385–394.

[169] Giesl J, Ströder T, Schneider-Kamp P, et al. Symbolic evaluation graphs and term rewriting - A general methodology for analyzing logic programs. In: Albert E, (eds.). Logic-Based Program Synthesis and Transformation, 22nd International Symposium, LOPSTR 2012, Leuven, Belgium, September 18-20, 2012, Revised Selected Papers, volume 7844 of *Lecture Notes in Computer Science*. Springer, 2012. 1.

[170] Keller R M. Formal verification of parallel programs. Commun. ACM, 1976, 19(7):371–384.

[171] Falke S, Kapur D. A term rewriting approach to the automated termination analysis of imperative programs. In: Schmidt R A, (eds.). Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings, volume 5663 of *Lecture Notes in Computer Science*. Springer, 2009. 277–293.

[172] Wang D, Zhang C, Chen G, et al. C code verification based on the extended labeled transition system model. In: Lara J, Clarke P J, Sabetzadeh M, (eds.). Proceedings of the MoDELS 2016 Demo and Poster Sessions co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2016), Saint-Malo, France, October 2-7, 2016., volume 1725 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2016. 48–55.

[173] Moura L M, Bjørner N. Z3: an efficient SMT solver. In: Ramakrishnan C R, Rehof J, (eds.). Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings, volume 4963 of *Lecture Notes in Computer Science*. Springer, 2008. 337–340.

[174] Termination Problems Data Base. http://cl2-informatik.uibk.ac.at/mercurial.cgi/TPDB.

[175] Termination Competition. http://termination-portal.org/wiki/Termination_Competition.

[176] Beyer D. Reliable and reproducible competition results with BenchExec and witnesses (report on SV-COMP 2016). In Chechik M and Raskin J[184], 887–904.

[177] DragonEgg. http://dragonegg.llvm.org/.

[178] Wolfe M I, Babich W A, Simpson R, et al. The Ada language system. IEEE Computer, 1981, 14(6):37–45.

[179] Chivers I D, Sleightholme J. Introduction to Programming with Fortran - With Coverage of Fortran 90, 95, 2003, 2008 and 77, Third Edition. Springer, 2015.

[180] Weiser M. Program slicing. IEEE Trans. Software Eng., 1984, 10(4):352–357.

[181] Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15-18, 1991. IEEE Computer Society, 1991.

[182] Dershowitz N, (eds.). Rewriting Techniques and Applications, 3rd International Conference, RTA-89, Chapel Hill, North Carolina, USA, April 3-5, 1989, Proceedings, volume 355 of *Lecture Notes in Computer Science*. Springer, 1989.

[183] Biere A, Bloem R, (eds.). Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings, volume 8559 of *Lecture Notes in Computer Science*. Springer, 2014.

[184] Chechik M, Raskin J, (eds.). Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings, volume 9636 of *Lecture Notes in Computer Science*. Springer, 2016.

[185] Demri S, Kapur D, Weidenbach C, (eds.). Automated Reasoning - 7th International Joint Conference, IJCAR 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 19-22, 2014. Proceedings, volume 8562 of *Lecture Notes in Computer Science*. Springer, 2014.

致　谢

首先衷心感谢我的导师顾明教授和联合导师孙家广院士对我的悉心教导。他们为人处世的作风以及思考问题的方式，都使我获益良多。每每我踌躇徘徊时，顾老师总会鼓励我坚定前行。顾老师从生活上、科研上对我的关心，让我十分感动。感谢顾老师给予了我坚持到最后的信心。而孙老师“做成一件事”的纯粹想法和实际行动，也让我耳濡目染，渐渐成为了我做事做人的目标。同时还要感谢两位老师给我提供机会到法国交流学习。

感谢我的合作导师 Jean-Pierre Jouannaud 教授，感谢他带我踏入重写的世界，让我接触到这个简单又复杂的理论，也感谢他在我到法国交流的一年间对我的照顾和帮助。同时也非常感谢中法联合实验室 LIAMA 的 Jean-François Monin 教授和 Frédéric Blanqui 博士，以及 FORMES 研究组的荔建琦博士，是他们带领我在形式化的世界里从零开始探索。

还要感谢宋晓宇教授、贺飞教授和周旻博士在建模验证方面给予我意见和指导，同时对我的论文选题与撰写提供了很多帮助。

感谢实验室的小伙伴们，特别是得希、陈光、张超、天池在工具开发上对我的帮助，以及特别靠谱的华枫和苏神对我全方位的帮助。

感谢软件学院羽毛球队的小伙伴，感谢有你们陪我一起在羽毛球场上挥洒汗水，也感谢大家在最后带我走上马杯领奖台。

感谢我的健身小伙伴阿谷，使我的健身时光更加轻松有趣。

感谢我在法国认识的小伙伴们，感谢你们不仅让我的郊区生活提高了质量，也让我的伙食提高了质量。

最后我由衷感谢我的家人对我的支持，谢谢我的父母、我的妹妹、我的外婆以及所有亲人们，对我的理解和关心，你们的支持对我很重要。也感谢我的挚友们，钟凌戈、黄佳迦、陈沭，还有许多，感谢你们陪我一起哭一起笑，终于等到这一天。

7 年的博士生涯如白驹过隙，然而回想起来，心中也是收获满满。感谢来到过我生命中的你们。

声　明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。

签　名：_____ 日　期：_____

附录 A　定理 3.2 的证明

本附录给出定理 3.2 的详细证明。

为了证明定理 3.2，我们需要先介绍更多关于重写逻辑的背景知识[134]。如果一个项表达式 t 不含变量，那么 t 称作是封闭的（ground）。给定原子命题的一个子集 $P \subseteq AP$ 和封闭的项表达式 t 和 t'，如果 t 满足的 P 中的命题和 t' 所满足的 P 的命题完全一致，则记作 $t \simeq_P t'$（如果 P 可从上下文推断，则简记为 $t \simeq t'$）。

时间鲁棒性是我们希望一个实时重写逻辑模型所具有的一系列性质的集合。为了简化概念，我们在这里避免给出时间鲁棒性的准确定义，因为它对本附录的证明无关重要。我们给出以下引理用于证明时间鲁棒性：

引理 A.1 ([134])：　给定一个面向对象风格的模型 $\mathcal{R}^{\mathcal{L}}$，它只包含一条标准的单元计时规则，且满足时间域中只包含一个非正常的时间值为 `INF`。如果 $\mathcal{R}^{\mathcal{L}}$ 对所有封闭的项表达式 t、r 和 r' 满足以下条件，则 $\mathcal{R}^{\mathcal{L}}$ 具有时间鲁棒性：

(i) 对所有 $r \leq$ `mte(`t`)` 满足 `mte(delta(`t`,`r`)) = mte(`t`) monus` r，其中 `monus` 是内置类型 `Time` 的减法操作；

(ii) `delta(`t`,`0`)` $= t$；

(iii) 对所有 $r + r' \leq$ `mte(`t`)` 满足 `delta(delta(`t`,`r`),`r'`)` $=$ `delta(`t`,`$r + r'$`)`；

(iv) 对任意瞬时重写规则的左项表达式 l 的所有封闭实例 $\sigma(l)$ 满足 `mte(`$\sigma(l)$`)` $= 0$。

任意一步重写可以被分类，在此基础上，我们可以定义单元计时不变性。

定义 A.1 ([134])：　给定重写步骤 $t \rightarrow^{r} t'$，假定它应用了单元计时规则，且其上标 r 代表该重写的时间推移为 r：

- 如果不存在时间值 $r' > r$ 使得对于某个 t'' 满足 $t \rightarrow^{r'} t''$，则该重写步骤被称为一个极大的单元计时步骤（maximal tick step），记作 $t \rightarrow^{r}_{max} t'$；
- 如果对任意时间值 $r' > 0$，都存在某个 t'' 满足 $t \rightarrow^{r'} t''$，则该重写步骤被称作一个 ∞ 单元计时步骤（∞ tick step），记作 $t \rightarrow^{r}_{\infty} t'$；
- 如果存在一个极大的单元计时步骤 $t \rightarrow^{r'}_{max} t''$ 满足 $r' > r$，则该重写步骤被称为一个非极大的单元计时步骤（non-maximal tick step）。

定义 A.2 ([134])：　一个时间鲁棒的模型 $\mathcal{R}^{\mathcal{L}}$ 被称作根据一个命题集合 P 保持单元计时不变，当且仅当，对任意非极大的单元计时步骤或 ∞ 单元计时步骤 $t \rightarrow^{r} t'$ 有 $t \simeq_P t'$ 成立。

我们需要以下引理证明我们的模型根据我们定义的命题具有单元计时不变性：

引理 A.2 ([134])：　给定一个时间鲁棒的、面向对象风格的模型 $\mathcal{R}^{\mathcal{L}}$，它只包含一条标准的单元计时规则，且满足时间域中只包含一个非正常的时间值为 INF。如果对于所有满足 $r < \texttt{mte}(t)$ 的 t 和 r，都有 $\{t\} \simeq_P \{\texttt{delta}(t, r)\}$ 成立，那么 $\mathcal{R}^{\mathcal{L}}$ 是根据原子命题集合 P 保持单元计时不变的。

我们先证明以下中间引理：

引理 A.3：　给定分别具有类型 MaybeNat 和 TaskList 的项表达式 ID 和 L，对所有 $r \leq \texttt{mteTask}(ID, L)$，有 $\texttt{mteTask}(ID, \texttt{deltaTask}(ID, L, r)) = \texttt{mteTask}(ID, L) \texttt{ monus } r$ 成立。

证明　如果 $ID = \texttt{none}$，则此情况显而易见。根据定义，有

$$\begin{aligned} & \texttt{mteTask}(\texttt{none}, \texttt{deltaTask}(\texttt{none}, L, r)) \\ = \ & \texttt{mteTask}(\texttt{none}, L) \\ = \ & \texttt{INF} \\ = \ & \texttt{mteTask}(\texttt{none}, L) \texttt{ monus } r \quad 。 \end{aligned}$$

否则，存在具有类型 Nat 的项表达式 N，使 $ID = \texttt{some } N$。假设 L 的第 N 个任务的 cnt 值为 $[r_e/C]$。则根据定义，$\texttt{deltaTask}(ID, L, r) = L'$ 成立，其中 L' 的第 N 个任务的 cnt 值等于 $[(r_e + r)/C]$。因此，

$$\begin{aligned} & \texttt{mteTask}(ID, \texttt{deltaTask}(ID, L, r)) \\ = \ & \texttt{mteTask}(ID, L') \\ = \ & C \texttt{ monus } (r_e + r) \\ = \ & (C \texttt{ monus } r_e) \texttt{ monus } r \\ = \ & \texttt{mteTask}(ID, L) \texttt{ monus } r \quad 。 \end{aligned}$$

以上，引理得证。　□

引理 A.4：　给定具有类型 IntSrc 的项表达式 $ISRC$，它表示我们的模型中中断源的一个可达状态。则对所有 $r \leq \texttt{mteIS}(ISRC)$，$\texttt{mteIS}(\texttt{deltaIS}(ISRC, r)) = \texttt{mteIS}(ISRC) \texttt{ monus } r$ 成立。

证明 任意可达状态 *ISRC* 必然具有以下形式 <*O*:IntSrc|val:*v*,cycle:*T*>，其中 $v \leq T$。因此，

$$\begin{aligned} & \texttt{mteIS(deltaIS(}\mathit{ISRC}\texttt{,}r\texttt{))} \\ = \ & \texttt{mteIS(<}\mathit{O}\texttt{:IntSrc|val:(}v\ \texttt{monus}\ r\texttt{),cycle:}\mathit{T}\texttt{>)} \\ = \ & v\ \texttt{monus}\ r \\ = \ & \texttt{mteIS(}\mathit{ISRC}\texttt{)}\ \texttt{monus}\ r \quad 。 \end{aligned}$$

以上，引理得证。 □

引理 A.5： 给定具有类型 Hardware 的项表达式 *HW*，则对所有项表达式 *r*，如果满足 $r \leq \texttt{mteIr(}\mathit{HW}\texttt{)}$，那么 $\texttt{mteIr(}\mathit{HW}\texttt{)} = \texttt{mteIr(}\mathit{HW}\texttt{)}\ \texttt{monus}\ r$ 成立。

证明 对项表达式 *HW* 中是否存在被检测到的中断请求进行分情况讨论，则引理可证。 □

定理 3.2 的详细证明如下：

证明 (定理 3.2 的证明) 我们先利用引理 A.1 证明我们建立的模型满足时间鲁棒性。由于我们的模型选择了内置的连续时间域 POSRAT-TIME-DOMAIN-WITH-INF 进行实例化，因此 INF 是模型中唯一的非正常时间值。如第 3.6.3.6 小节所述，在模型中我们只定义了一条标准单元计时规则。因此，根据引理 A.1，如果条件 (i–iv) 成立，那么我们的模型满足时间鲁棒性。

条件 (i)。我们需要证明，对所有 $r \leq \texttt{mte(}s\texttt{)}$，均有 $\texttt{mte(delta(}s\texttt{,}r\texttt{))} = \texttt{mte(}s\texttt{)}\ \texttt{monus}\ r$ 成立，其中 *s* 是具有类型 System 的系统状态项表达式。假设 *s* 具有形式 (*L T STS HW ISRC*)，且 *ID* = (*HW*).getPc。在此我们详细讨论 *ID*::MaybeNat 成立的情况，另一情况 *ID*::Oid 的证明过程类似。根据 mte 和 delta 的定义，

$$\begin{aligned} & \texttt{mte(delta(}s\texttt{,}r\texttt{))} \\ = \ & \texttt{mte(deltaTask(}\mathit{ID}\texttt{,}L\texttt{,}r\texttt{)}\ \ T\ \mathit{STS}\ \mathit{HW}\ \texttt{deltaIS(}\mathit{ISRC}\texttt{,}r\texttt{))} \\ = \ & \texttt{minimum(mteTask(}\mathit{ID}\texttt{,deltaTask(}\mathit{ID}\texttt{,}L\texttt{,}r\texttt{)),} \\ & \qquad \texttt{mteIS(deltaIS(}\mathit{ISRC}\texttt{,}r\texttt{)),}\ \texttt{mteIr(}\mathit{HW}\texttt{))} \quad 。 \end{aligned}$$

根据引理 A.3、引理 A.4 和引理 A.5，以上表达式可以进一步化简：

$$
\begin{aligned}
& \texttt{mte(delta(}s,r\texttt{))} \\
=\ & \texttt{minimum(mteTask(}ID,L\texttt{) monus }r\texttt{ ,} \\
& \qquad \texttt{mteIS(}ISRC\texttt{) monus }r\texttt{ ,} \\
& \qquad \texttt{mteIr(}HW\texttt{) monus }r\texttt{)} \\
=\ & \texttt{minimum(mteTask(}ID,L\texttt{),} \\
& \qquad \texttt{mteIS(}ISRC\texttt{),} \\
& \qquad \texttt{mteIr(}HW\texttt{)) monus }r \\
=\ & \texttt{mte(}s\texttt{) monus }r\ \ 。
\end{aligned}
$$

条件 (ii)。由于对所有 r 都有 $r+0=r$ 和 $r \texttt{ monus } 0 = r$ 成立，因此条件 (ii) 成立。

条件 (iii)。我们需要证明 $\texttt{delta(delta(}s,r\texttt{),}r'\texttt{) = delta(}s,r+r'\texttt{)}$ 对所有 $r+r' \leq \texttt{mte(}s\texttt{)}$ 成立，其中 s 是具有类型 `System` 的系统状态项表达式。采用与条件 (i) 相同的符号，我们给出 $ID\texttt{::MaybeNat}$ 成立时的详细证明。根据定义，待证等式的左边

$$
\begin{aligned}
& \texttt{delta(delta(}s,r\texttt{),}r'\texttt{)} \\
=\ & \texttt{(deltaTask(}ID,\texttt{deltaTask(}ID,L,r\texttt{),}r'\texttt{)} \\
& \quad T\ STS\ HW\ \texttt{deltaIS(deltaIS(}ISRC,r\texttt{),}r'\texttt{))}\ ,
\end{aligned}
$$

而待证等式的右边

$$
\begin{aligned}
& \texttt{delta(}s,r+r'\texttt{)} \\
=\ & \texttt{(deltaTask(}ID,L,r+r'\texttt{)}\ T\ STS\ HW\ \texttt{deltaIS(}ISRC,r+r'\texttt{))}\ 。
\end{aligned}
$$

由于 + 具有结合律，因此

$$
\texttt{deltaTask(}ID,\texttt{deltaTask(}ID,L,r\texttt{),}r'\texttt{)} \ = \ \texttt{deltaTask(}ID,L,r+r'\texttt{)}
$$

成立。又因为 $(v \texttt{ monus } r) \texttt{ monus } r' = v \texttt{ monus } (r+r')$ 对所有 `Time` 类型的项表

达式 v 成立，于是

$$\texttt{deltaIS(deltaIS(}\mathit{ISRC}\texttt{,}r\texttt{),}r'\texttt{)} \;=\; \texttt{deltaIS(}\mathit{ISRC}\texttt{,}r+r'\texttt{)}$$

成立。综上，条件 (iii) 成立。

条件 (iv)。我们证明任意瞬时规则的左项实例的 mte 值等于 0。比如考虑规则 interrupt-request，根据其条件约束 (*ISRC*).timeout，可得知 *ISRC* 的 val 值等于 0。所以，

$$\begin{aligned}
&\texttt{mte(}L\ T\ \mathit{STS}\ \mathit{HW}\ \mathit{ISRC}\texttt{)}\\
=\;&\texttt{minimum(mteTask(}\mathit{ID}\texttt{,}L\texttt{),mteIS(}\mathit{ISRC}\texttt{),mteIr(}\mathit{HW}\texttt{))}\\
=\;&\texttt{minimum(mteTask(}\mathit{ID}\texttt{,}L\texttt{), 0 ,mteIr(}\mathit{HW}\texttt{))}\\
=\;&0 \quad 。
\end{aligned}$$

以此类推，其它规则也可根据其条件约束得到证明。综上所述，根据引理 A.1 可证我们的模型满足时间鲁棒性。

最后，我们证明用于分析模型的命题（即 taskTimeout 和 correct）满足单元计时不变性。根据引理 A.2，我们必须证明 $\{s\} \simeq_P \{\texttt{delta(}s\texttt{,}r\texttt{)}\}$ 对所有 System 类型的项表达式 s 和 $r < \texttt{mte(}s\texttt{)}$ 成立。也就是说，对系统状态项表达式应用单元计时规则使得时间往前推移 r 个时间单位，并不会对任意命题的真值产生影响。假设 s 的形式为 ($L\ T\ \mathit{STS}\ \mathit{HW}\ \mathit{ISRC}$) 且 $(\mathit{HW})\texttt{.getPc} = \mathit{ID}$。

- 命题 taskTimeout 成立当且仅当 L 中包含 error。由于 delta 并不会在 L 中产生或减少 error，因此命题 taskTimeout 针对状态 s 成立，当且仅当，taskTimeout 针对状态 $\texttt{delta(}s\texttt{,}r\texttt{)}$ 成立，其中 $r < \texttt{mte(}s\texttt{)}$。这意味着我们的模型根据 taskTimeout 保持单元计时不变性。
- 命题 correct 的真值取决于 *ID* 以及 L 中所有任务的状态。与上述情况类似，delta 无法改变 *ID* 或 L 中任意任务的状态，所以命题 correct 在状态 s 成立，当且仅当，correct 在状态 $\texttt{delta(}s\texttt{,}r\texttt{)}$ 也成立，其中 $r < \texttt{mte(}s\texttt{)}$。模型根据命题 correct 的单元计时不变性得证。

综上所述，根据定理 3.1，我们应用非时控模型检测对可调度性及系统正确性进行验证的方法具有完备性。 □

个人简历、在学期间发表的学术论文与研究成果

个人简历

1987 年 9 月 24 日出生于广东省惠州市。

2006 年 9 月考入清华大学软件学院计算机软件专业，2010 年 7 月本科毕业并获得工学学士学位。

2010 年 9 月免试进入清华大学计算机科学与技术系攻读博士学位至今。

发表的学术论文

[1] **Liu Jiaxiang**, Zhou Min, Song Xiaoyu, Gu Ming, Sun Jiaguang. Formal modeling and verification of a rate-monotonic scheduling implementation with Real-Time Maude. IEEE Trans. Industrial Electronics, 2017, 64(4):3239–3249. (SCI 源刊, JCR-1 区期刊 TIE, 影响因子 6.383)

[2] **Liu Jiaxiang**, Jouannaud Jean-Pierre, Ogawa Mizuhito. Confluence of layered rewrite systems. In: Kreutzer S, (eds.). 24th EACSL Annual Conference on Computer Science Logic, CSL 2015, September 7–10, 2015, Berlin, Germany, volume 41 of LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015. 423–440. (EI 收录, 检索号 20161002065898, CCF-C 类会议 CSL)

[3] **Liu Jiaxiang**, Dershowitz Nachum, Jouannaud Jean-Pierre. Confluence by critical pair analysis. In: Dowek G, (eds.). Rewriting and Typed Lambda Calculi - Joint International Conference, RTA-TLCA 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14–17, 2014. Proceedings, volume 8560 of Lecture Notes in Computer Science. Springer, 2014. 287–302. (EI 收录, 检索号 20143118004586, CCF-C 类会议 RTA)

[4] **Liu Jiaxiang**, Jouannaud Jean-Pierre. Confluence: The unifying, expressive power of locality. In: Iida S, Meseguer J, Ogata K, (eds.). Specification, Algebra, and Software - Essays Dedicated to Kokichi Futatsugi, volume 8373 of Lecture Notes in Computer Science. Springer, 2014. 337–358. (EI 收录, 检索号 20143117998944)

[5] Jouannaud Jean-Pierre, **Liu Jiaxiang**. From diagrammatic confluence to modularity. Theoretical Computer Science, 2012, 464:20–34. (SCI 收录, 检索号 WOS:000311985000003, CCF-B 类期刊 TCS, 影响因子 0.643)