施健 硕士学位论文答辩委员会成员名单

姓名	职称	单位	备注
欧阳树生	高级工程师	上海宝信软件股份有限公司	主席
刘爱玲	高级工程师	上海外高桥造船有限公司	
范昌琪	高级工程师	普华基础软件股份有限公司	

摘　要

现代社会软件开发技术的发展日新月异，不同领域的软件开发方式也是大相径庭。近些年世界各国都在大力发展和强化国防安全、轨交系统、工业及能源等方面的建设，以此增强各国的综合国力，而这些安全攸关领域的发展离不开计算机软件和应用。正因如此，在安全攸关领域，软件应用的开发方式也由传统的编码式开发转变为基于模型的开发。法国 Esterel 科技公司的 SCADE 软件套装就是一个用于开发安全攸关领域嵌入式控制软件的开发环境，该软件是模型驱动开发方式的代表，工程师只需关注软件系统的功能性建模，而无需将精力放在软件代码实现上。这大大提高了这类软件系统的开发效率。

安全攸关领域的软件安全性一直以来都备受关注，尽管软件应用的开发效率得到了提升，但依旧得保证软件系统的安全性和可靠性。毕竟在这些领域，软件发生错误将会带来严重的经济损失甚至会危及生命。传统的软件验证技术是对开发好的软件系统进行仿真模拟，进而发现错误并及时改正。不过 SCADE 还提供了形式化验证组件 Design Verifier（DV），它先对安全需求进行图形化建模，然后通过基于 SAT 的模型检查算法验证模型是否满足需求。但随着 SCADE 广泛的应用以及系统需求的多样性，DV 的表达能力已不够描述那些需求，尤其当涉及到时序相关的性质时。为了突破 SCADE 模型验证的这个限制，我们提出了一个 SCADE 模型时序性质验证方法，该方法将 SCADE 状态机转换到模型检查器 NuSMV 的输入模型上，引入线性时态逻辑（LTL）、计算树逻辑（CTL）等时态逻辑作为描述模型安全性需求的规范。与此同时，我们基于该方法开发了对应的验证工具 Scade2Nu，该工具的主要组件有 SCADE 文本模型解析器、符号表容器以及模型转换器，通过该工具工程师便可以使用 LTL 或 CTL 时态逻辑表达式来描述更多样的系统需求，也能在系统设计阶段发现更多潜在的设计错误，从而进一步提高软件系统的安全性与可靠性。最后本文使用 Scade2Nu 验证了实际案例中的安全性需求和性质，发现并改正了原模型的设计错误，从而也验证了文中方法的有效性和可行性。

关键词： 模型驱动；SCADE；形式化验证；模型检查；安全攸关系统

ABSTRACT

The development of software development technology in modern society is changing with each passing day, and the software development methods in different fields are also quite different. In recent years, in order to enhance the comprehensive national strength, countries around the world have been vigorously developing and strengthening national defence security, rail transit systems, industry and energy. And the development of these safety-critical fields is inseparable from computer software and applications.

For this reason, in the safety-critical fields, the development of software applications has also changed from traditional coding development to model-based development. The SCADE suite from Esterel Technologies of France is a development environment for developing embedded control software in the safety-critical fields. This software is representative of the model-driven development method. Engineers only need to pay attention to the functional modelling of software systems without having to focus on software code implementation. This greatly improves the development efficiency of such software systems.

Software security in the safety-critical fields has always been a concern. Although the development efficiency of software applications has been improved, the security and reliability of software systems are still guaranteed. After all, in these areas, software debugs will bring serious economic losses and even life-threatening. The traditional software verification technology simulates the developed software system, and then finds errors and corrects them in time. However, SCADE also provides a formal verification

component, Design Verifier (DV), which first graphically models safety requirements and then verifies that the model meets the requirements through an SAT-based model checking algorithm. But with the wide application of SCADE and the diversity of system requirements, its ability of expression is not enough to describe those needs, especially when it comes to temporal properties. In order to break this limitation of SCADE formal verification, we propose a new method to verify the temporal properties of SCADE state machine. This method transforms the SCADE state machine to the input model of the model checker NuSMV and introduces temporal logics such as linear temporal logic (LTL) and computational tree logic (CTL) as specifications for describing the system requirements. At the same time, we develop the corresponding verification tool called Scade2Nu based on this method. The main components of the tool are SCADE textual model parser, symbol table container and model translation engine. Through this tool, engineers can use LTL or CTL temporal logic to express more kinds of system requirements and then find more potential design debugs in the design phase. Thereby it can further improve the security and reliability of the software system. Finally, this paper uses Scade2Nu to verify the safety requirements and properties in the industrial case, finds and corrects the design errors of the original model, and also verifies the validity and feasibility of the proposed method.

Keywords: *Model-driven development; SCADE; Formal verification; Model checking; Safety-critical system*

目录

插图

表格

第一章　绪 论

1.1　研究背景及意义

随着计算机技术与应用的不断发展，软件系统的复杂性日益提高。在航空航天、轨道交通、汽车电子、医疗设备、工业与能源等安全攸关领域，嵌入式软件早已作为它们系统的主要部件发挥着重要作用，并控制着系统的运转。一旦这些软件在运行过程中出现错误，则可能引发灾难，造成严重的财产损失甚至危及生命。因此减少安全攸关领域的软件缺陷尤为重要。

模型驱动开发（MDD）通过其自身的优势成为了近年来软件系统开发的趋势之一，在模型驱动开发流程中，工程师可以将重点放在高层次的模型设计上，而可执行代码则能够通过强大的开发环境自动生成，从而提高系统的设计便捷度以及开发效率 [1]。另一方面，随着系统应用场景的多样化以及系统功能复杂性的提高，传统开发方式越来越难确保高层次系统模型设计与低层次系统实现的一致性，由于在安全攸关领域，嵌入式软件对安全性的要求十分严厉 [2]，高层次的模型设计必须满足系统的功能需求，若到低层次实现时才发现系统的错误，那么势必会影响到整个系统的研发进度，同时，改正这些错误所花费的时间与资源也会成倍增加。而模型驱动开发能够在系统设计前期就能及时发现潜在问题，从而能够大大减少系统开发的周期与成本。

正因以上特点，现今在安全攸关领域的嵌入式软件开发中，基于模型驱动开发过程得到了广泛应用。SCADE（Safety Critical Application Development Environment）便是一款模型驱动开发环境，专门用来设计安全攸关领域的软件，例如引擎控制系统、轨交控制系统、紧急制动系统、能源与燃料控制系统、核电站控制等工业控

制系统 [3]。SCADE 覆盖了系统开发的多个环节，包括需求管理、基于模型的设计、模型仿真、模型验证以及代码生成。使用 SCADE 设计开发的软件能够具有多种安全、质量认证，例如航空领域的 DO-178B/C 标准、工业控制领域的 IEC 61508 标准、汽车行业的 ISO 26262 标准以及铁路领域的 EN 50128 标准。此外，SCADE 的基于图形化模型开发方式也具有诸多优势，例如解决了控制系统设计中固有的复杂性，提供了软件复用的结构，以及可在代码生成之前验证系统的安全需求，从而可以更早地检测与消除与需求不符的设计错误。因此，SCADE 为高安全性软件开发者提供了完整的嵌入式开发解决方法，可以有效进行低成本高质量的开发，从而提高软件的安全性与可靠性 [4]。

安全攸关领域的嵌入式软件对安全性具有高标准与高要求，SCADE 使用形式化方法来验证系统的安全需求是否能够满足。为了使得用 SCADE 设计出来的软件系统达到这些要求，SCADE 可根据用户提供的安全需求来对系统进行安全性质分析，得到系统是否满足这些安全需求的反馈信息，然后根据结果修改系统设计。在 SCADE 中，工程师可通过图形化的方式设计系统的模型，由于 SCADE 同时还是一个模型语言，因此用它设计出来的模型拥有严格的形式化描述，其严格的数学理论可以确保能够清晰地、无歧义地表达系统功能需求，无二义性问题。另一方面，根据形式化验证流程，用自然语言描述的安全需求还需转换为某种形式化规范描述，之后才能基于模型对这些安全性质进行形式化验证。Design Verifier[5] (DV) 是 SCADE 的形式化验证组件，这个组件可以为系统的安全性质提供形式化的规范描述，经过对系统的验证得以确保系统设计的正确性。DV 使用基于 SAT 的模型检查方法 [6] 来进行系统的安全性与可达性分析，但随着模型驱动开发的发展，SCADE 的用户也越来越多，再加上系统场景的丰富与需求的多样，越来越苛刻、复杂的安全性质需要得到验证。Daskaya 等人 [7] 通过将 SCADE 模型转换为 UPPAAL[8] 时间自动机来分析工业案例中系统的活性（liveness property）。活性也可用来刻画系统的安全性需求，它不像不变性（invariant properties）那样用以确保某种性质在系统中一直不变，而是描述了那些不需要一直维持，但最终总能满足的性质。在

形式化验证领域，大多数这类性质都可以被诸如线性时态逻辑 (LTL)、计算树逻辑 (CTL) 这样的时态逻辑来描述 [9]。但是，DV 并不支持时态逻辑规范描述，它无法使用 LTL 或 CTL 来规范系统的安全性质，这就意味着 DV 描述安全需求的能力是有限的，尤其是那些与时序相关的性质。然而时序性质（temporal property）在安全攸关领域又是十分重要的，为了能够减少汽车嵌入式软件中的有限状态错误、时间错误以及其他错误，Lettnin 等人 [10] 基于仿真提出了两种验证时序性质的方法。对于航空领域的飞行器软件，验证其执行路径上的时序性质也是一个挑战 [11]。因此，工程师迫切需要 SCADE 拥有验证多种时序性质的能力。如果可以为 SCADE 模型提供一种验证方法，使得工程师能够使用 LTL 以及 CTL 这些时态逻辑规范来描述系统的安全需求，那么对于系统而言，不但可以验证 DV 能够验证的不变性，还能够验证用时态逻辑描述的有界时间、无界时间的时序性质。

引入时序性质验证方法的意义在于进一步完善 SCADE 高安全性软件开发过程的系统设计阶段。用户能够在系统设计阶段进一步减少模型潜在的设计错误，当发现系统中有不能满足需求的错误，工程师便可以根据反馈信息来调试和修改模型，进一步保证 SCADE 模型的正确性与可靠性。由于系统开发基于模型驱动，而当系统模型能够满足安全需求且设计错误也降至最低时，自动生成代码的安全性则会随之提高，从而可以有效降低在系统实现阶段才发现错误所造成的资源与时间成本，优化和普及以 SCADE 为代表的模型驱动开发过程。因此，该研究对高安全性嵌入式软件开发的正确性、安全性和可靠性具有非常重要的意义。

1.2 国内外研究现状

1.2.1 SCADE 模型的形式化验证

SCADE 是 Lustre 同步数据流语言 [12] 以及安全状态机的融合，Hangen 等人 [13, 14] 引入了一个基于 SMT 的方法来验证 Lustre 程序的不变性，并且为该语言的模型开发了一个模型检查器 KIND。SCADE 模型的形式化验证长期依赖于其组件 Design Verifier，不过由于 DV 的局限，Basold 等人 [15] 根据 Hangen 和 Tinelli 的研

究，提出了另一种方式来验证SCADE模型的不变性，这种方法将SCADE模型翻译到一个中间语言LAMA，然后将该语言进一步转换到一系列SMT公式，通过定义一些不变式来验证系统的一些安全性质。然而在SCADE形式化方法中，关于验证复杂的时序性质的研究则比较稀少。Daskaya等人[7]将SCADE模型转换到UPPAAL时间自动机来验证一个工业案例中系统的活性，除此之外的SCADE模型时序性质验证的研究则是少之又少。由于SCADE本身还拥有状态机的描述能力，并且在实际系统设计中它的状态机多是层次化的结构，但就国内外而言对这类SCADE模型时序性质验证方案的研究目前则是一块无人问津的领域。不过SCADE的状态机与Statecharts的联系[16]可以启发我们寻求为SCADE进行时序性质验证的新方法。

1.2.2 状态机与 Statecharts 的模型检查

模型检查[17]是形式化验证的一个重要技术，而其中系统模型的行为需要用一种精确而无歧义的方式来描述。大部分模型都会使用有限状态机来描述。SCADE中也结合了状态机来描述系统的行为，但它的状态机拥有嵌套性、层次化等特点，这给SCADE模型的验证带来了挑战。将层次化状态机展开是一种方法，Xavier等人对层次化状态机的展开技术做了十分详细的研究与评估，他们认为状态机展开技术可用于解决代码生成、模型检查、基于模型测试等问题，但却只是通往这些问题的其中一步；正因如此，展开（flattening）算法的描述通常没有提供准确的结构，也无法对这些算法的适用性做出合理的评估；另一方面，这些算法的有效性往往并不理想[18]。

Statecharts[19]状态图由David Harel在1987年提出，它是对传统的状态机的广泛扩展，其目标是使状态机能够更有效地运用在刻画复杂系统的实际工作中。Statecharts可视化地描述了复杂的反应系统[20]。反应系统是一类不断与其环境进行交互的系统，这里的环境可以是物理环境、人的操作等，而系统则是运行在特定硬件上和特定操作系统上的软件。SCADE便是用来开发这类系统的集成环境。状态图Statecharts的特点包括了层次化体系、并发以及并发组件间通过广播机制的通信，这些特点同样是验证状态图Statecharts的挑战所在。状态图Statecharts的模

型检查方法以转换居多。Latella 与 Lilius 等人 [21, 22] 提出了从 UML Statechart 状态图的子图到 PROMELA/SPIN[23] 的转换，这些转换方法支持了一部分状态图功能的转换，而 PROMELA 作为 SPIN 的语言，可以做异步进程的模型检查且使用线性时态逻辑 LTL 来描述需要检查的性质，它还能够拓展到验证一些通信协议及分布式系统中。另一思路则是将 Statecharts 状态图转换为 SMV[24]，Chan 等人 [25] 将状态图 Statecharts 的变体 Harel 状态图 RSML 作为输入模型，然后将其翻译成 SMV，接着通过对性质进行规范地描述使其能够分析鲁棒性以及安全攸关性质。除此之外，在工具 STATEMATE[26] 中，Clarke 等人提出了叫做 STP 的方法 [27]，通过时态语言 ETL 定义了从 Statechart 到 SMV 的模型转换框架，将 CTL 表达式应用到模型性质的验证中，这也启发了本文对 SCADE 状态机模型形式化验证的研究。

1.3 本文研究内容

本文主要研究 SCADE 状态机模型的时序性质验证方法。根据上文所述可知，要想为 SCADE 的模型提供更多安全相关性质的验证，尤其是时序性质方面的验证，则需要为它引入时序性质规范的描述能力。本文将结合 Statechart 状态图模型检查和 SCADE 模型形式化验证这两个领域，研究如何为 SCADE 模型引入新的安全性质验证方法。考虑到许多验证状态图的工作都将它们转换到一个规范严谨的形式化模型上，然后通过对应的模型检查器来进行模型检查，本文将研究如何将 SCADE 状态机转换成模型检查器 NuSMV[28] 的模型语言，从而验证 SCADE 模型的需求性质。模型转换具有诸多优势，首先是目标模型语言的正确性与完备性，其次是成熟且高效的模型检查算法，还有丰富的性质规范表达能力。而 NuSMV 的模型语言的特点之一则是其模块化（Module）的编写方式，用这种方式编写模型可以很容易构建出层次化状态机的结构，非常适合用于 SCADE 状态机模型。本文具体的研究内容和贡献如下：

1. 研究如何定义 SCADE 状态机到 NuSMV 输入模型的转换规则。基于 STP 方法，本文将定义 SCADE 到 NuSMV 的转换，包括初始状态、子状态机、状态

转移等模块化转换规则，使其能够适用于 SCADE 状态机。本文将改进 STP 方法中的变量监控机制，设计监控参数，在目标模型中构建 SCADE 的监控变量机制；通过该些规则，我们便可以将状态机主导的 SCADE 模型转换至 NuSMV 的输入模型上。然后，我们将自然语言描述的安全需求用 NuSMV 语义中定义的 LTL 或 CTL 规范重新描述，将它们和模型一同输入到 NuSMV 检查器中便可以验证 SCADE 模型的安全性质。

2. 研究如何形式化定义层次化的 SCADE 状态机结构。Statechart 中的状态是完全抽象的，而 SCADE 状态机里的状态会拥有许多数据流操作，更加具体。本文将研究并形式化定义 SCADE 层次化状态机结构，使其能够描述 SCADE 状态机中状态的行为。

3. 设计模型转换算法。本文将基于转换规则与 SCADE 状态机的形式化定义，设计从 SCADE 状态机模型到 NuSMV 输入模型的转换算法。

4. 设计并实现 SCADE 安全性质分析工具 Scade2Nu。本文基于以上研究内容，开发 SCADE 模型的安全性质验证工具 Scade2Nu，并应用在实际安全攸关领域的 SCADE 模型中。Scade2Nu 将 LTL 和 CTL 的性质表达能力引入到 SCADE 需求形式化描述中，该工具解析 SCADE 文本模型，生成中间结构，然后使用我们设计的转换算法得到目标模型。通过将 NuSMV 模型检查工具整合进 Scade2Nu，工程师便能进行完整的 SCADE 模型验证流程，从而为 SCADE 工程师提供时序性质的验证方案。

在这篇论文中，我们提出一个方法来确保安全攸关软件领域中 SCADE 状态机安全性，在 SCADE Design Verifier 的基础上进一步保证了 SCADE 模型的时序关系不出问题。该方法拓展了 SCADE 中的性质规范描述，通过引入 LTL 与 CTL 时序规范，为 SCADE 模型在时序性质方面增加了验证的手段。通过分析验证后返回的结果来修正模型，以确保模型的时序安全，从而减少在系统设计阶段发生错误的概率，提高安全攸关领域系统开发的时序安全性。

1.4 本文组织结构

本文共分六章，组织结构如下：

第一章绪论介绍了本文的研究背景与研究意义，介绍了 SCADE 模型的形式化验证、状态机与 Statecharts 的模型检查的国内外研究现状，并说明了本文的研究内容和相关贡献以及本文的组织结构。

第二章预备知识与概念将介绍必备的背景知识，这些预备知识将有助于对本文的理解。这一章将详细介绍 SCADE，包括它的建模语言、代码生成、模型仿真、以及形式化验证；除此之外，还会介绍模型检查器 NuSMV 的基础知识，包括它输入模型的形式化语言，性质规范的形式化过程，以及这些性质规范表达式的语义。

第三章将定义从 SCADE 状态机到 NuSMV 的输入模型的转换规则。这一章基于 STP 方法，从初始状态、子状态机、状态转移、监控变量机制等方面提出了详细的转换规则。针对监控变量机制，设计适合 SCADE 层次化状态机的监控参数，提出其转换规则。

第四章将介绍 SCADE 模型验证框架和工具 Scade2Nu，介绍本文提出的验证流程、定义的层次化 SCADE 状态机的形式化描述，以及基于转换规则的算法；同时介绍可视化工具 Scade2Nu 的架构设计以及实现。

第五章将介绍 Scade2Nu 的案例研究，从轨道交通、汽车电子以及航天航空三个安全攸关领域的应用场景中说明如何使用 Scade2Nu 来验证这些 SCADE 模型，同时给出对 Scade2Nu 的评价。

第六章总结与展望会对本文提出的方法进行总结，并且将讨论现阶段 Scade2Nu 的不足，并展望未来的改进之处。

1.5 本章小结

本章首先介绍了本文的研究背景与研究意义，讲述了模型驱动开发在安全攸关领域软件系统开发中的作用，以及模型驱动开发环境 SCADE 在该类软件开发中的重要性，并指出了 SCADE 模型在形式化验证方法还有提升空间，尤其在时序性

质这类安全需求上 SCADE 工具链的不足。接着介绍了 SCADE 模型形式化验证国内外研究现状，之后便说明了本文的研究内容旨在解决 SCADE 模型在时序安全性质上的验证问题，最后给出了本文的文章组织结构。

第二章　预备知识与概念

2.1　模型驱动软件开发平台与建模工具 SCADE

SCADE（Safety Critical Application Development Environment）是由法国爱斯特尔技术有限公司发布的一款高安全应用开发工具套装，为高安全性系统和软件开发人员提供完整的基于模型的嵌入式开发解决方案，可用来开发安全攸关的系统应用，从而降低系统软件的开发成本、减少开发风险并缩短验证时间。SCADE 服务于许多安全攸关领域，例如轨道交通行业、航空航天领域、国防安全行业、能源与核工业、工业与医疗等等行业。这些行业中的许多巨头公司都使用该产品来研发各自领域下的嵌入式应用。

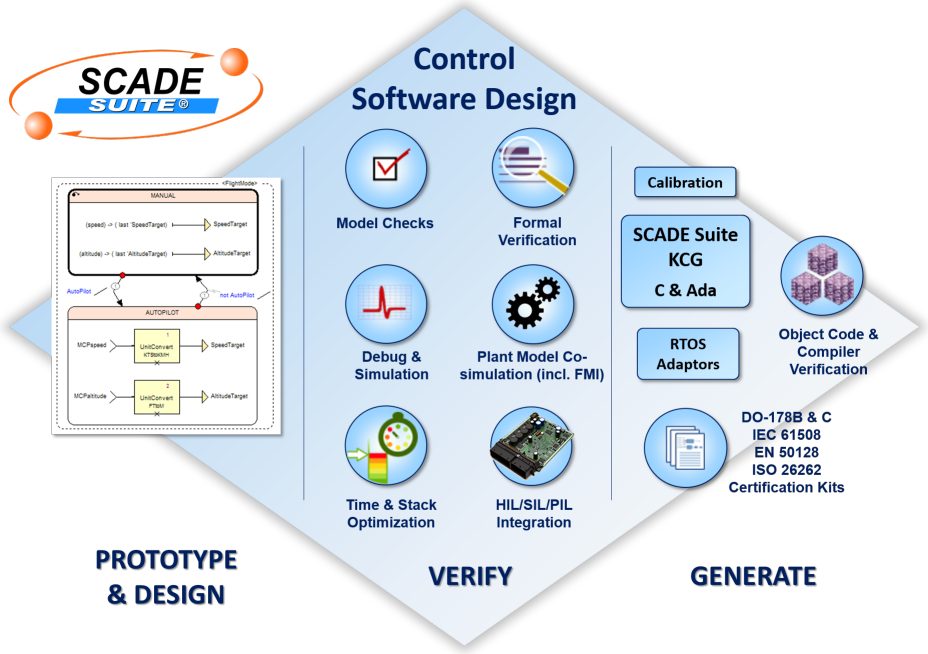


图 2.1: SCADE Suit 产品介绍图

2.1.1　SCADE 建模语言

SCADE 之所以能够被如此广泛的应用与它强大的嵌入式软件开发能力密不可分，它拥有一个同叫做 SCADE 的形式化建模语言。使用 SCADE 建模的优势在于使用模型驱动开发方法。它为用户提供了一个可视化的建模界面，该可视化建模开发方式可以让工程师从最基本的系统需求出发，控制系统中需要用到的复杂算法可以在新的模块中设计，然后在控制系统中调用它，这样也大大提高了一些模块函数的可复用性，提高了软件的开发效率。

SCADE 主要用来设计反应式系统，如图2.2所示，反应式系统的模型接收由传感器捕获的环境输入，通过系统控制命令进行运算后作出响应，输出到执行设备中。而这些输出也有可能会影响到环境，从而改变传感器的值。这种模型执行的方式是循环式的，在下一个周期，还会进行之前的流程。用 SCADE 建立的模型还遵循同步假设 [29]，也就是在每个周期内，假设当模型接受传感器的值并进行计算时，环境与它互不影响，并假设了反应式系统响应速度非常快。

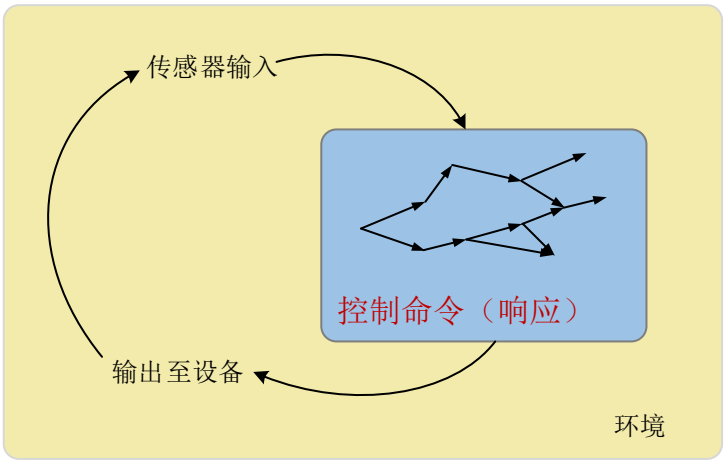
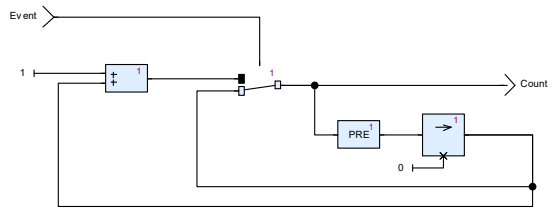


图 2.2: 反应式系统

SCADE 语言基于同步数据流语言 Lustre，是一种声明式、结构化数据流建模语言，因此可用于对反应式系统的建模。用户建立的图形化模型其背后就是用 SCADE 语言描述的模型，而图形化建模又包括了数据流图和安全状态机两种方式。通常 SCADE 中的模块节点可以是基本的操作符，也可以是用图形化或者文本表示的

SCADE 语句。一个控制系统的模型就是通过多个模块节点连接而成，这种方式类似于 Simulink 的模型 [30]。SCADE 语言能够支持用户自定义类型，支持在实数和自然数上的数学运算、在布尔类型上的逻辑运算、支持控制流操作以及时态操作。如图2.3(a)所示是一个用 SCADE 数据流图建立的简单的计数器模块节点。该节点的输入是叫做 Event 的一个布尔类型的输入变量，其中的 PRE 是一个时态操作符，保存了上一个周期的数据流结果。该节点通过加法运算操作符来计数，从而产生一个整型的名为 Count 的输出流。图2.3(b)则是该数据流节点所对应的 SCADE 文本语言模型。



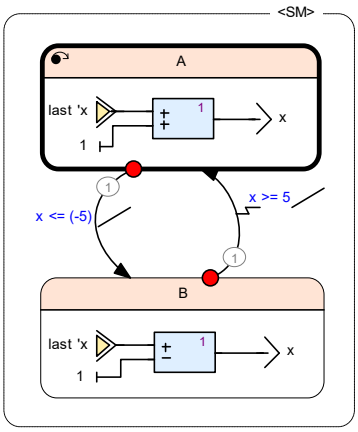
(a)

```
node Count(Event : bool)
returns(Count : int)
var
_L1 : int; _L2 : int;
let
_L1 = if Event then (1+_L2) else _L2;
_L2 = 0 -> pre _L1;
Count = _L2;
tel
```

(b)

图 2.3: 计数器 Count 模块节点

在 SCADE 建模中，安全状态机 [31] 是一种刻画控制系统的状态和它状态内部的逻辑功能的建模方式。用户可以通过将安全状态机的图标拖动到设计界面上来设计系统的各个状态。安全状态机是有限状态机的拓展，能够支持系统进行顺序、并发、层次化的控制。通常当系统应用有控制主导这方面要求时，基于状态机建模是更好的选择。使用安全状态机的建模方式能够让用户仅关注应用设计中的复杂系统之间的逻辑结构，从而更好地确保建模的正确性。如图2.4(a)所示，我们挑选了一个 SCADE 状态机，它拥有两个状态，分别是 A 与 B。A 为初始状态，该状态的边框略微加粗，状态 A 可以定义数据流操作来描述一些行为。其中 last ′x 记录了上一个时间单元下 x 的值，当 x<=-5，状态 B 被激活，那么在当 x>=5 之前，系统仅执行 B 中的操作。其 SCADE 语言表示的文本形式如图2.4(b)所展示。



(a)

```
node machine() returns(x : int)
let
automaton SM
   initial state A
   unless if x <= (-5)
          restart B;
   let
      x = last 'x + 1;
   tel
   state B
   unless if x >= 5
          restart A;
   let
      x = last 'x - 1;
   tel
returns x ;
tel
```

(b)

图 2.4: SCADE 状态机示例

2.1.2　SCADE 自动代码生成器 KCG

SCADE 为开发者提供了一套可执行代码自动生成工具 KCG[32]。用 SCADE 设计完后的模型可以通过 KCG 得到目标平台的可执行嵌入式 C 代码。与此同时，自动代码生成器 KCG 生成的代码满足多种安全标准，例如航空航天领域中的 DO-178B、DO-178C 认证、轨道交通领域的 EN 50128 标准，核能源领域的 EN 60880 标准，以及工业领域的 IEC 61508 标准等等。KCG 根据输入的 SCADE 模型以及用户设定的一些参数，通过目标代码编译器生成最终的可执行 C 代码程序。

2.1.3　SCADE 模型仿真

与此同时 SCADE 还提供了一个模型仿真环境。SCADE 模型仿真并不是在模型上的真正仿真，而是通过 KCG 转换成可执行的 C 代码来模拟的，这样仿真器通过载入场景输入得以仿真调试由这些 C 语言代码形成的 SCADE 模型程序。用 SCADE 设计的任何一个模块或者子系统都能够使用它提供的仿真器进行仿真模拟。当进入选定的子系统的仿真器后，便能够看到它的输入、输出等变量。仿真器还能够监控变量值和变量走势，用户可以选择感兴趣的变量，将这些变量在每一个周期里的值都显现出来，从而直观地分析数据的变化。

2.1.4　SCADE 形式化验证

为保证 SCADE 模型设计的正确性，SCADE 开发过程使用了形式化验证方案来减少软件设计中的错误，保证需求安全 [33]。现代工业系统中对信息技术的需求不断增加，安全和关键任务系统也越来越多。而这些系统的可靠性越来越依赖于软硬件的正常运作，一旦发生错误通常代价高昂甚至致命，因此对系统验证方面的要求也就越来越高。

传统的验证技术通常采用系统规范的检查，包括运用一些算法审查，但会留有不明确的部分。仿真与测试当然也是验证手段，但由于需要设计测试用例后才能执行并检查输出。因此工作量巨大，且不一定能将错误完全覆盖。即使发现错误，也依旧无法证明设计的软件不存在其他错误。因此，传统验证阶段所花费的成本在不断增加，甚至可能比系统构建所花费的时间还要多，而对于系统的设计错误，越早发现越好。

形式化验证的引入能够达到降低验证成本，提高系统可靠性的目的。SCADE 形式化验证在验证过程中加入了数学推理，用数学模型来表示系统的行为与性质，将需求表示成逻辑表达式，然后在数学模型中检查这些表达式成立与否。Design Verifier（DV）[5] 则是 SCADE 形式化验证的模块组件，它是系统安全性质的模型检查器。在 Design Verifier 中，系统的安全需求使用 SCADE 图形化模型来表示，一般称这些实现安全性质的节点为观察器（Observer），安全需求的验证是要确保该性质在该 SCADE 模型的任意周期和环境下都是满足的。DV 将需要验证的模型和观察器结合，通过输入、输出流自动验证 SCADE 模型是否满足安全需求，其内部的核心算法基于 Stalmarck 的 SAT 求解算法，用于处理布尔公式，通过归纳法处理时态行为和状态空间搜索 [4]。如果得到正向的结果，则说明这个安全需求所具有的性质是能够被系统满足的，而若被证伪，则 DV 会生成一个反例来给用户继续调试原模型。图2.5表示的就是 SCADE 形式化验证阶段的流程。

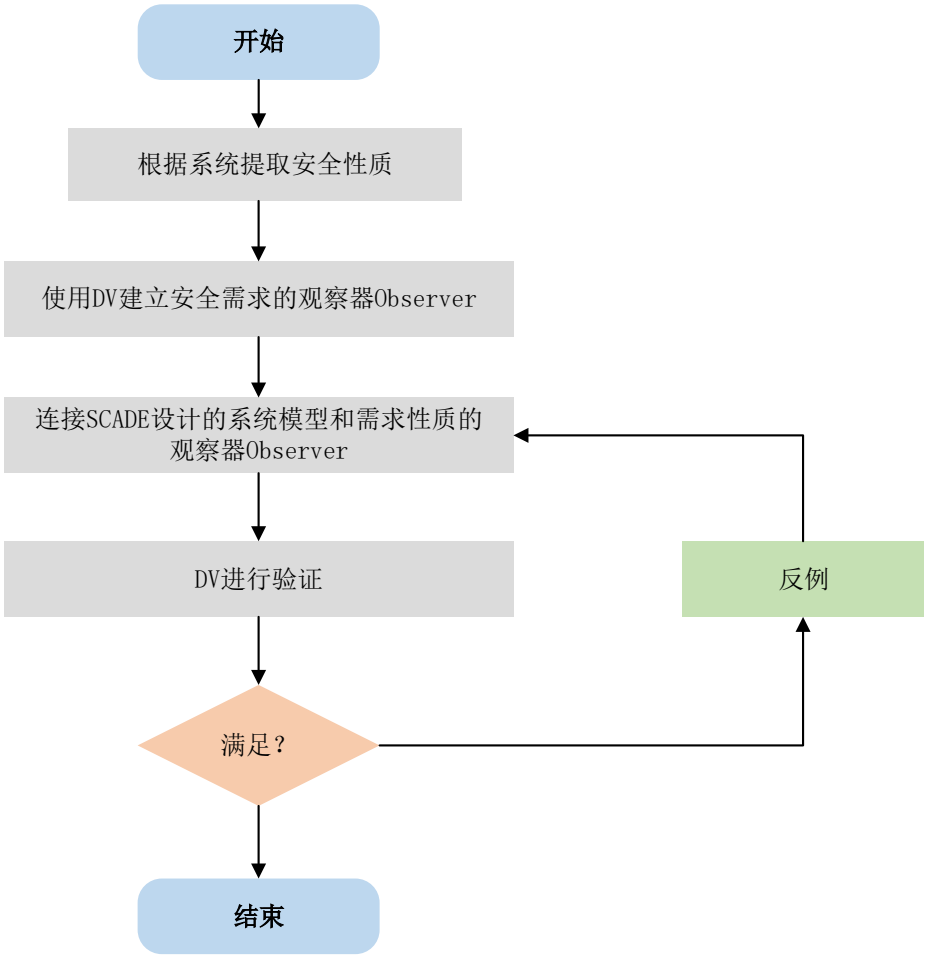


图 2.5: Design Verifier 验证过程

2.2　模型检查器 NuSMV

NuSMV 是一个符号模型检查器，它拓展并重新实现了原先由 CMU 开发的基于 BDD[34] 的模型检查器 CMU SMV[35]。而最新的 NuSMV 更是整合了基于 SAT 的模型检查算法。基于 BDD 和基于 SAT 的模型检查通常能够解决不同类别的问题，因此这两个技术可以互补。NuSMV 也是一个非常流行的适用于时态逻辑的模型检查器，它能够用来描述有限状态机，不管是异步的还是同步的，与系统相关的时序性质可以用计算树逻辑（CTL）与线性时态逻辑（LTL）描述。通过 NUSMV 基于 BDD 与 SAT 技术的模型检查算法，便能够验证模型是否满足这些性质规范。

在 SCADE 中，用户会经常使用状态机来设计控制系统，而从抽象层次来看，

SCADE 状态机的结构与 StateChart 是类似的，它们都支持层次化状态机；由于 NuSMV 是一个成熟的模型检查工具，它的模型语言也是准确的，这就保证了转换的目标模型拥有严格正确的语义；另外，NuSMV 内置了高效实现的模型检查算法；最重要的是它能够通过模块化的建模方式来表达和描述层次化状态机结构的模型。这些因素也促使了本文选择将 NuSMV 的输入语言作为 SCADE 状态机模型的目标模型。

图 2.6: NuSMV 模型描述的有限状态机示例

2.2.1 NuSMV 输入模型

NuSMV 的模型语言用一个有限状态迁移系统 [36] 来描述要验证的模型。该系统可以描述成一个 Kripke 结构 $M = (S, I, \rightarrow, L)$，其中集合 S 是一个有限的状态集合；$I \subseteq S$ 是初始状态的集合；$\rightarrow \subseteq S \times S$ 是转移关系，描述了状态之间可能存在的转移；L 则是标记函数，标记了给定状态下原子命题是否成立。NuSMV 的模型语言提供的数据类型有布尔类型、有界整型、符号枚举类型、以及有界数组等等。如图2.6所示是一个 NuSMV 模型的例子，一个 NuSMV 模型通常具有下面两个代码块：`VAR` 代码块和 `ASSIGN` 代码块。`VAR` 代码块主要声明了该状态机中涉及到的变量，包括状态名的集合以及原子命题变量；`ASSIGN` 代码块又由三部分构成，第一部分是状态变量的初始化，第二部分是状态之间的转移关系，第三部分是特定状态下原子命题的赋值操作。复杂的模型系统可以将它分解成多个模块（`MODULE`）。每个模块定义一个有限状态机，并且能够在其他模块中被实例化。这种特点能够使 NuSMV 模型非常灵活且可复用。

2.2.2 NuSMV 时序规范与验证

在上一小节中，我们已经知道了 SCADE Design Verifier 中的形式化验证过程，但 DV 的性质描述能力要弱于 CTL 或 LTL。NuSMV 能够描述计算树逻辑（CTL）、线性时态逻辑（LTL）以及过去的线性时态逻辑（Past-LTL）的规范，NuSMV 进行模型检查的流程如图2.7所示，在 NuSMV 中，性质规范可以通过关键字 LTLSPEC 和 CTLSPEC 来声明。例如在 LTL 逻辑表达式中，能够使用 G（globally）、F（finally）、X（next）、Y（previous）、U（until）等表达式，除此之外，还有命题逻辑表达式 !（not）、& （and）、|（or）、xor（exclusive or）、-> （implies）和 <->（equivalence）。对于 CTL 逻辑，能够使用以下表达式：EG （exists globally），EX （exists next state），EF （exists finally），AG （forall globally），AX （forall next state），AF （forall finally），E[U] （exists until），A[U] （forall until）。

值得一提的是，NuSMV 还分别为 LTL 与 CTL 拓展了带时序边界描述的表达能

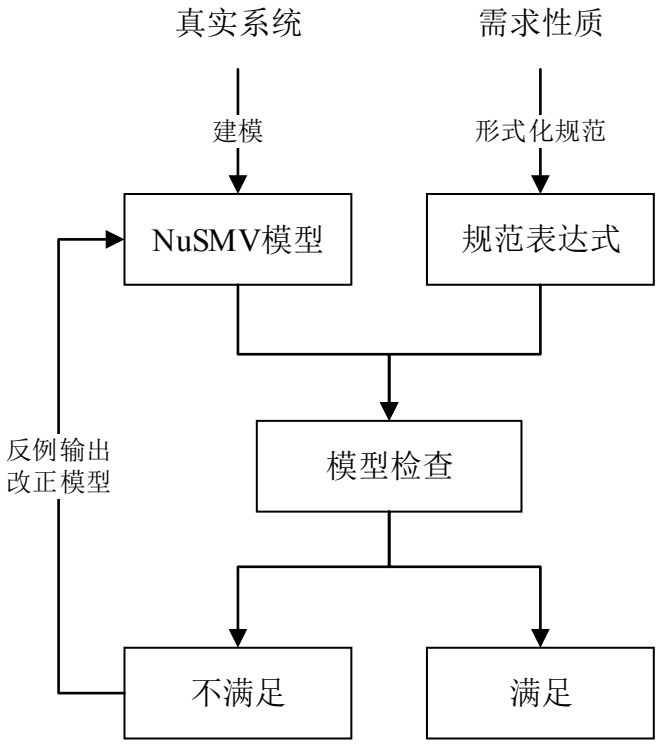


图 2.7: NuSMV 模型检查过程

力。除了 LTL 中的 G（globally）、F（finally），LTL 还拓展了 G[bound]、F[bound] 用来描述在一定时间区间中的性质，其中 bound 就是一个区间。而对于 CTL，NuSMV 拓展了实时的 CTL 表达式，同样支持在 CTL 表达式中加上时间区间，允许描述在某一段时间中需要满足的性质。在 NuSMV 的使用手册中有对这方面详细的规范说明 [37]。

最终将需要验证的时序性质用这些时态逻辑表达式表示后，通过 NuSMV 模型检查器就能够在系统模型上进行模型检查，若系统模型不满足性质，那么根据输出的反例改正原模型。

2.3　本章小结

本章主要介绍了一些预备知识与概念，主要包括对模型驱动软件开发与建模工具 SCADE 和模型检查器 NuSMV 的基本知识的介绍。与此同时，本章也讲述了

选择 NuSMV 的模型语言作为转换目标语言的原因，从而更好地述说下文的转换原理和算法。

第三章　SCADE 状态机到 NuSMV 模型的转换

本章将详细介绍从 SCADE 状态机转换到 NuSMV 输入模型的转换规则。该规则基于 STP 方法，为了使其能够适用于 SCADE 状态机，本章针对 SCADE 的层次化结构及状态机特点重新定义了转换规则，并改进了它的监控变量机制，从而创建 SCADE 状态机到 NuSMV 输入模型的转换框架。如图3.1所示，是一个控制节点（Node）中的 SCADE 状态机，该状态机具有层次化的结构。状态机 SM1 有两个状态 A 与 B，而状态机 SM2 拥有两个状态 C 和 D，并且 SM2 是状态 B 的子状态机。当状态转移条件 $g1$ 满足时，状态 A 会转移到状态 B；因此图中的 $g1, g2, ...$ 代表了状态之间的转移条件。在 SCADE 的状态里有具体的运算操作，并且在每个周期下，每个状态机有且只有一个状态是处于激活的。

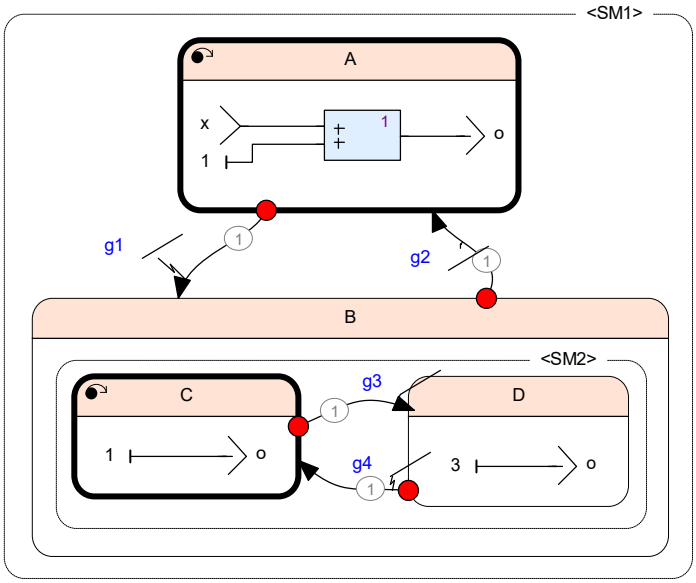


图 3.1: 层次化 SCADE 状态机示例

3.1　SCADE 层次化状态机结构的转换

3.1.1　触发器参数 active 与 default

SCADE 层次化状态机拥有 StateChart 的特点，因此与 STP 方法类似，我们为转换规则引入两个布尔类型的触发器参数 active 和 default。这两个触发器参数可以帮助目标模型构建出 SCADE 状态机中的层次结构，从而正确地用 NuSMV 模型语言描述出来。

触发器参数 active

如图3.2所示，它是仿真下状态机的情况，红色加粗代表着状态目前处于被激活状态。SCADE 状态机中的状态转移发生在同一层上，若目标状态为 B 的状态转移条件满足，则状态机会激活该转移，SM1 的状态 B 也被激活，与此同时状态机 SM2 中的状态 C 也激活。图中代表了在仿真下的状态机情况，红色加粗代表着状态目前处于被激活状态。

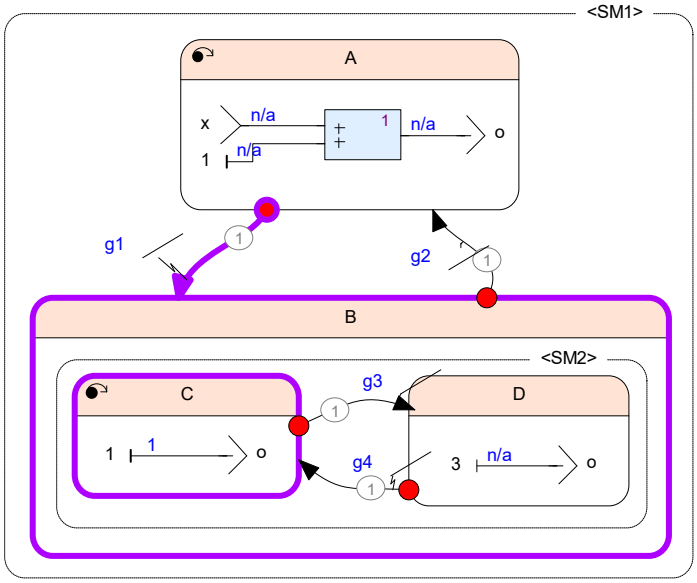


图 3.2: 仿真中状态 B 与状态机 SM2 被激活

换句话说，在 SCADE 层次化状态机中，当父状态激活时，其子状态机才会被激活。而直接将这样的结构转向 NuSMV 输入模型时就会出现问题：NuSMV 程序

中的所有模块（MODULE）都会在一开始时进行初始化，并在之后一直处于激活状态。显然不能直接以这样的方式进行转换，当对应的 MODULE 不应该在某些周期中被激活时，目标 NuSMV 模型必须以某种方式明确指出，以防止在 NuSMV 模型对变量造成不正确的操作，这也是为什么需要引入 active 触发器参数，通过它来处理 SCADE 状态机与 NuSMV 模型启动时的冲突，标记出对应的 MODULE 是否能被认为是激活的。

在转换框架中，每一个子状态机最终都会转换成 NuSMV 的一个 MODULE。假设状态机 S_{sub} 是状态 S 的子状态机，active 代表状态 S 所在的状态机是否是激活的，令 SM_active 代表子状态机 S_{sub} 是否激活，如**规则 1** 所示，它的值由赋值符号右侧的表达式确定，即当状态 S 的父状态激活，且目前状态处于 S 时，那么 S_{sub} 状态机也被激活。

规则 1：子状态机S_{sub}的触发器参数 active
`SM_active := (state = S) & active;`

触发器参数 default

每一个状态机都有它自己的初始状态，触发器参数 active 能告诉我们某个子状态机模块是否被触发。而一个状态可能会发生两种不同的转移，分别为，

(1) 默认转移（default transition）：当状态机被触发的时候，它的初始状态会首先被激活，我们称这为默认转移。

(2) 常规转移（regular transition）：该状态机在其所在层状态之间发生的转移，我们称它们为常规转移。

在 SCADE 中，假设某个子状态机被激活并执行了常规转移，若干周期后，该子状态机的父状态因迁移条件变化发生状态迁移，这就意味着父状态会从激活状态变为非激活状态。而当该子状态机的父状态下一次被激活时，子状态机需要发

生默认转移，即子状态机的初始状态要被激活。而当我们将 SCADE 状态机转换到 NuSMV 输入模型时，仅有 active 触发器参数依旧无法保证子状态机的 MODULE 会进行默认转移，这就是为什么需要引入另一个触发器参数 default。通过设置参数 default 为真（True），目标模型就能确保每次进入子状态机时会发生默认转移，激活初始状态。

令 S 为当前状态，$P_1, ..., P_n$ 是状态 S 的前继（predecessors）。当状态 $P_1, ..., P_n$ 中有一个到状态 S 的转移条件满足时，触发器参数 default 需设置为真（True）。与此同时，只有当状态 S 的所在的状态机处于激活状态时才有意义。如**规则 2** 所示，赋值语句必须与触发器 active 合取（&）。假设 Def_S 代表了 S 状态下的子状态机 S_{sub} 的触发器 default 参数，那么首先我们初始化该参数为 False，代表着 S_{sub} 状态机还未初始化；然后运用以上的转换规则来定义 Def_S：

规则 2：子状态机S_{sub}的触发器参数 default

```
init(Def_S) := FALSE;
  next(Def_S) := (active & (state = P_1|...|P_n)
   & next(active)
   & next(state) = S) ? TRUE : FALSE;
```

除此之外，还有一个特殊情况，也就是最顶层的状态机。SCADE 状态机模型中的所有状态机都是最顶层的状态机的子状态机，例如图3.1中 SM1 则是最顶层状态机 SM 的子状态机。这个状态机对应于 NuSMV 目标模型中的主模块（MODULE main）。最顶层的状态机实际上可以不需要触发器参数 default，因为它总是激活的。因此我们选择在第一个时间单元将 default 设置为 True，然后在以后的时间单元设置为 False，这样就能保证最顶层的状态机的默认转移仅在最开始发生，并且只持续一个周期。根据**规则 3**，我们可以在最终 NuSMV 程序中的主模块中定义它的参数 default：

规则 3：顶层状态机的触发器参数 default
init(default) := **TRUE**; **next**(default) := **FALSE**;

3.1.2　NuSMV 目标模型中层次化状态机的结构

现在，通过触发器参数 active 与 defalt 的转换规则就能规范出 NuSMV 目标模型中的层次化状态机结构。假设 S 是一个子状态机 SMi 的一个状态，SMj 是状态 S 的子状态机，并且该子状态机 SMj 有状态 $s_1, ..., s_n$。NuSMV 目标模型中，子状态机 SMi 与 SMj 的层次化结构如图3.3所示，在 SMi 中参数 active 代表着 SMi 是否处于激活状态；由于 SMj 是状态 S 的子状态机，SMj_active 代表着子状态机 SMj 是否激活。该结构结合了规则 1 和规则 2：

```
MODULE SMi(..., default, active)
VAR
   state : {..., S, ...};
VAR
   Def_S : boolean;
ASSIGN
   init(Def_S) := FALSE;
   next(Def_S) := (active & (state = P_1|...|P_n)
      & next(active)
      & next(state) = S) ? TRUE : FALSE;
DEFINE
   SMj_active := (state = S) & active;
VAR
   S_sub : SMj(..., Def_S, SMj_active);
   ...

MODULE SMj(...,default,active)
VAR
   state : {s_1,...,s_n};
   ...
```

图 3.3: NuSMV 目标模型中层次化状态机的结构

3.1.3　状态转移规则

在 SCADE 层次化状态机存在状态之间的转移关系，回顾图3.1，子状态机 SM1 中有两个状态 A 和 B，当转移条件 $g1$ 满足时，状态 A 到状态 B 的转移就会发生。以此类推，当 $g3$ 满足时，子状态机 SM2 中从状态 C 到状态 D 的转移就会发生。每个子状态机在 NuSMV 的目标模型中都以模块（`MODULE`）的形式出现，SCADE 的子状态机里的状态名都会在对应模块中的枚举变量 state 中声明，并且拥有两个触发器参数 active 与 default。参数 active 表示该模块是否真正激活，default 代表了是否发生默认的初始转移。因此，只有当触发器参数 active 为 True 时，该模块中的状态名变量 state 才是可见的。

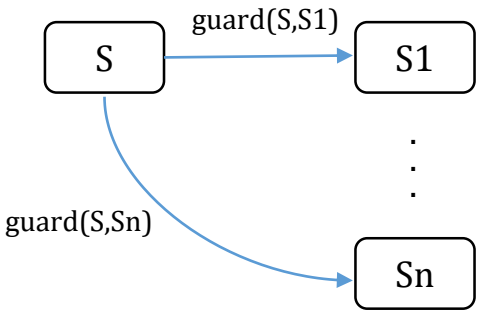


图 3.4: 状态 S 的转移图

根据 SCADE 状态机的特点，我们知道状态转移仅发生在同一层次的状态机中。如图3.4所示，子状态机中 $S_1, ..., S_n$ 是状态 S 的目标状态，它们分别通过转移条件 $guard(S, S_1), ..., guard(S, S_n)$ 进行状态转移。通过 NuSMV 的 next 关键词，定义常规状态转移的转换**规则 4**。

从中我们也能看出，当某个周期发生状态转移时，状态名会发生变化；若所有对应于特定状态的转移条件都未满足，那么下一个状态依旧是自身。这与 SCADE 状态转移的情形是一致的，当 SCADE 状态机未发生转移时，所在状态的操作会再一次执行。这些都建立在所在状态机处于激活的条件下，因此需要将触发器参数 active 与它们做合取。

而当子状态机要发生默认转移时使用以下的状态转移**规则 5** 激活初始状态：

规则 4：状态S到其目标状态S_i的转移

```
next(state) :=
  case
     active & state = S & guard(S,Si) : Si;
     …
     TRUE : state;
  esac;
```

规则 5：默认转移情况下的状态转移

```
next(state) := next(active) & next(default): S0;
```

3.2 SCADE 变量监控机制

反应式系统根据环境的输入作出响应，输出到特定设备并更新原环境的传感值，而 SCADE 模型决定了系统如何响应。因此在 SCADE 设计的控制系统中必然拥有对输出变量产生影响的操作与行为。再次回顾图3.1，可以发现状态 A 中对输入变量 x 进行了数值运算操作，并将其输出到了输出变量 o 中。这也应证了 SCADE 状态中的行为会更新和改变声明变量的值。在 NuSMV 目标模型中，SCADE 里的环境输入会定义在 NuSMV 的模型的主模块（`MODULE main`）中，并将这些输入声明为全局变量，在每一个子状态机模块中根据环境输入作出响应和操作，更新全局变量的值。

NuSMV 中，每个模块能够读取到主模块中全局变量的值，然而 NuSMV 并不支持这些子状态机模块同时更新一个全局变量。STP 方法中的监控式机制（monitor-like mechanism）可以使模型能够读写受监控的变量。监控式机制意味着对于 StateChart 中的事件或条件变量 Var，总是拥有一个对应的模块叫做 Set_Var，这个模

块能够通过监控参数来操纵变量 Var 的值。

但问题是，STP 方法中的事件与条件变量都抽象成了布尔类型的变量，而 SCADE 中的变量数据类型会更复杂。并且在 SCADE 中，状态转移条件与状态中的事件行为都是十分具体的数据流操作。例如在 SCADE 状态机里，一个状态 S 会使用到一些运算操作符来对特定的变量做出响应。

为了解决 STP 方法对于 SCADE 状态机的不足，本文提出了面向 SCADE 模型的变量监控的机制，从而为子状态机与全局变量建立一个通讯机制。SCADE 变量监控机制首先将 STP 方法中的两类变量监控参数 set_m 和 $reset_m$ 精化为 $set_{(var,s)}$ 和 $reset_{(var,s)}$，其次设计 SCADE 变量监控模块，最后提出了两类变量监控参数在各个子状态机里的赋值规则。

SCADE 变量监控参数

由于在 SCADE 中，事件行为与转移条件都是含有输入、输出变量的表达式，而输入变量又由环境决定，因此仅需对输出变量进行监控。这些监控参数都会作为全局变量声明在 NuSMV 目标模型的主模块中。

定义 3.2.1. *在 SCADE 状态机中，令一个状态 s 中存在一个操作行为，该操作行为更新输出变量 var 的值，则有：*

- *变量监控参数 $set_{(var,s)}$ 是一个布尔类型的参数，表示当所处状态为 s 时，该状态中影响变量 var 的行为操作是否需要被执行；*
- *变量监控参数 $reset_{(var,s)}$ 是一个布尔类型的参数，表示当目标状态为 s 的状态转移发生时，目标状态 s 中影响变量 var 的行为操作是否需要被执行。*

SCADE 变量监控模块的转换规则

我们还需要定义变量监控模块的转换规则。设某个输出变量 Var，对该变量的监控模块被命名成 Set_Var。这类模块负责根据监控参数的值来改变变量 Var 的值，当对应的监控参数为 True 时，则能执行对变量 Var 的赋值操作。令 $v_1, ..., v_i, ..., v_n$

为 SCADE 状态机中的所有变量，状态 $s_1, ..., s_m$ 中都存在对输出变量 v_i 的操作，**规则 6** 就是关于变量 v_i 的监控模块的转换规则。其中 $set\_v_i\_s_1, ..., reset\_v_i\_s_m$ 代表了相关的监控参数 $set_{(var,s)}$ 和 $reset_{(var,s)}$。具体在特定状态 s_k 中，对变量 v_i 的赋值操作的表达式由 $set\_action(s_k)$ 和 $reset\_action(s_k)$ 表示。这些赋值操作是在状态中的数据流操作，我们可以直接将它们转换成对应的表达式。转换规则也支持这类赋值操作调用其它用户定义的函数节点，因此如果需要的话，还得声明调用的函数名 $function_1, ..., function_k$。

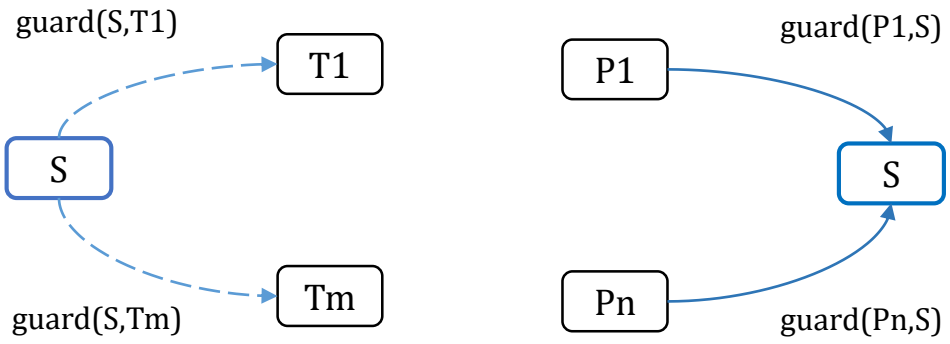
规则 6：变量监控 Module

```
MODULE Set_Vi(v1,…,vi,…,vn,
set_vi_s1, reset_vi_s1, …, set_vi_sm, reset_vi_sm,
function_1,…,function_k)
INIT
   vi = default value;
ASSIGN
   next(vi) :=
    case
      next(set_vi_s1)   : set_action(s1);
      next(reset_vi_s1) : reset_action(s1);
       …
      next(set_vi_sm)   : set_action(sm);
     next(reset_vi_sm) : reset_action(sm);
      TRUE              : vi;
    esac;
```

SCADE 变量监控参数的赋值规则

通过以上规则，以及之前引入的触发器参数 active 和 default，我们已经能够在 NuSMV 的模型上建立起 SCADE 层次化状态机的结构了。但是只有当变量监控参数处于正确的值，整个层次化状态机才能够正确地运转。这些监控参数就似状态行为的开关。当其中一个开关打开，那么这个开关控制的变量就会得到更新。因

此我们需要动态地控制这些开关，也就是在每一个子状态机对应的模块中控制这些监控参数。



(a) 状态 s 为起始状态的转移

(b) 状态 s 为目标状态的转移

图 3.5: 状态转移关系

- 监控参数 $set_{(var,s)}$ 的赋值规则

令当前起始状态为 s，如图3.5(a)所示 $T_1, ..., Tm$ 是状态 s 的后继状态，$guard(s, T_i)$ 代表从状态 s 到 T_i 的转移条件，其中 $i \in \{1, ..., m\}$。在 SCADE 状态机中，若状态 s 的所有转移条件都没满足，那么下一个状态依旧是状态 s，并在下个周期执行状态中的操作。因此，如**规则 7** 所示，状态 s 所在的状态机一直处于激活，那么若它的所有转移条件都没满足时，设置 $set_{(var,s)}$ 的值为 True，否则为 False。其中 set_var_s 等同于 $set_{(var,s)}$。

- 监控参数 $reset_{(var,s)}$ 的赋值规则

如图3.5(b)，状态 s 是状态 $P_1, ..., P_n$ 的目标状态，从状态 P_j 到状态 s 的转移条件记作 $guard(P_j, s)$，其中 $j \in \{1, ..., n\}$。当 $guard(P_j, s)$ 中的其中一个转移条件满足时，系统会到达状态 s，那么监控参数 $reset_{(var,s)}$ 就会被设置。**规则 8** 则是 $reset_{(var,s)}$ 的赋值规则，需要注意的是，若目标状态 s 是某个子状态机的初始状态，那么也得将该监控参数设置为 True。其中 $reset\_var\_s$ 等同于 $reset_{(var,s)}$。

规则 7：监控参数 $set_{(var,s)}$

```
next(set_var_s) :=
   active & next(active)
   & case
       state = s & !guard(s,T1)
                 …
                  & !guard(s,Tm) : TRUE;
       TRUE : FALSE;
     esac;
```

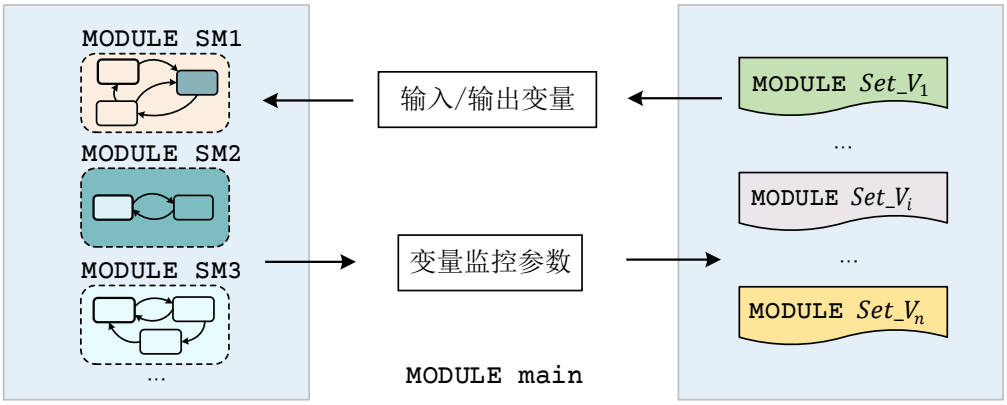


图 3.6: SCADE 变量监控机制

经过精化后的 SCADE 变量监控机制能够通过监控参数来间接操纵输出变量的取值。如图3.6所示是 SCADE 参数监控机制的示意图，该机制能够建立子状态机模块和输出变量监控模块之间的通信。之前也提到，系统的环境变量对应了输入变量，而输入变量会影响状态中其他变量的值，以及转移条件中表达式的值，以此使得各个状态机进行状态迁移。然后每个子状态机 SMi 就会读取变量，进而改变监控参数的取值。而监控参数则能够决定该不该执行特定状态下对某个输出变量的操作，如果监控参数为 True，则变量监控模块则能够更新对应输出变量的值。这就是该机制如何通过监控参数来改变各个变量的取值的。如此循环，构建成 SCADE

规则 8：监控参数 $reset_{(var,s)}$

```
next(reset_var_s) :=
   next(active)
   & case
      !active & !default & next(default) : TRUE;
       active & state = P1 & guard(P1,s) : TRUE;
       …
       active & state = Pn & guard(Pn,s) : TRUE;
       TRUE : FALSE;
     esac;
```

的变量监控机制。

3.3　SCADE 顶层状态机的生成

NuSMV 目标模型中的主模块（MODULE main）就是用来描述 SCADE 最顶层的状态机的，这个状态机只有一个状态，并且总是激活的。主模块主要处理一些变量声明，包括输入变量、输出变量、状态变量、监控变量等；除此之外还需处理一些实例化，包括子状态机模块的实例化、变量监控模块的实例化、用户自定义节点函数模块的实例化，以及对监控变量的初始化。

3.4　本章小结

本章详细介绍了 SCADE 状态机到 NuSMV 输入模型的转换规则，分别讲述了 SCADE 层次化状态机结构的转换、SCADE 变量监控机制以及其主模块的生成。这些规则为下一章转换算法及 SCADE 状态机验证工具 Scade2Nu 的设计提供了理论基础。

第四章　Scade2Nu 验证框架及其实现

本文上一章中阐述了从 SCADE 状态机模型到 NuSMV 输入模型的转换规则。本章节将阐述完整的 SCADE 状态机模型的时序性质验证框架，包括我们研发的验证工具 Scade2Nu。Scade2Nu 在 SCADE 开发环境和 NuSMV 模型检查器之间建立了桥梁，将模型检查中的时序性质验证技术带进了 SCADE 需求形式化验证中。

系统建模师首先会根据系统的基本需求设计系统模型，同时也会用自然语言描述出需要系统软件满足的安全需求和性质。如图4.1所示是 SCADE 状态机的验证流程和 Scade2Nu 的设计架构，该过程接受一个系统的 SCADE 模型和它的安全需求。SCADE 软件支持用户将模型导出为一个 SCADE 文本模型，该文本模型用 SCADE 语言描述，拥有完整的语法和语义。Scade2Nu 接收到系统的文本模型后便开始转换过程，通过核心组件，将模型转换成 NuSMV 的输入模型。NuSMV 是一个命令行式的模型检查工具，为了良好的用户体验，Scade2Nu 将 NuSMV 的可视化工具 NuSeen[38] 整合进来，通过软件后台调用 NuSMV，简化用户操作，使用户不用再写繁琐的执行命令。后台的 NuSMV 会接收转换好的输入模型，同时输入用户提前用时态逻辑 LTL 或 CTL 重写后的需求性质规范，便可进行模型检查，最后将验证结果反馈到 Scade2Nu 的可视化界面上。用户通过分析验证结果，对原 SCADE 模型进行调试，然后重复该过程继续验证，直到系统的安全需求全部满足。

下文将重点介绍 Scade2Nu 的架构设计，它由三个组件构成，分别是 Scade2Nu 文本模型解析器、符号表容器以及模型转换器。

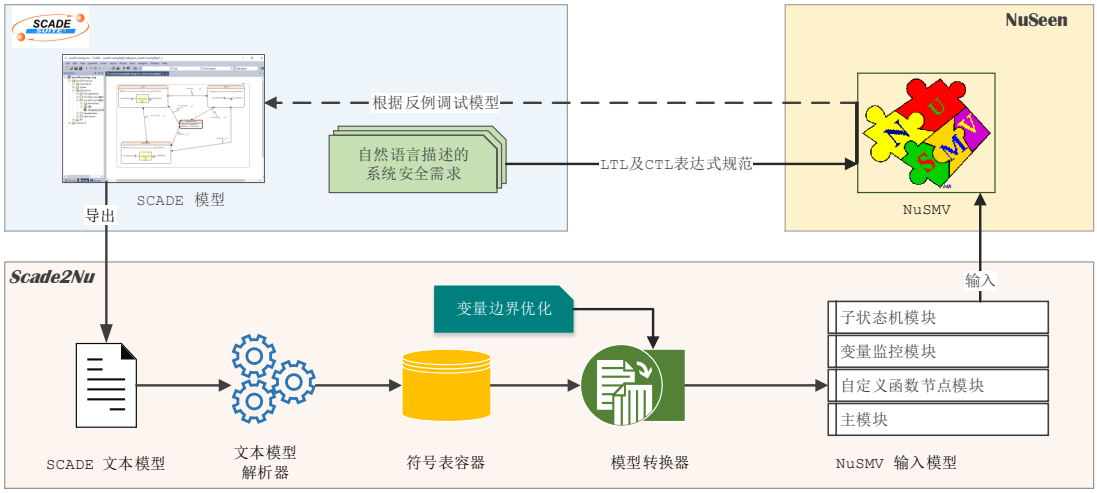


图 4.1: Scade2Nu 验证 SCADE 模型的验证框架及其软件架构

4.1　Scade2Nu 文本模型解析器

SCADE 既是一个基于模型驱动的嵌入式系统软件开发环境，同时也是一个建模语言。SCADE 文本模型解析器主要负责解析导出的文本模型，并生成 SCADE 抽象语法树，然后装载到下文会提到的符号表容器中。SCADE 语言的数据流由同步数据流 Lustre 语言发展而来，控制状态机基于 StateCharts[19, 39]。我们可以从上文的 SCADE 状态机中看出，大部分 SCADE 模型都是状态机与数据流两种形态的结合 [40, 41]。

SCADE 语言发展以来，拥有了成熟的词法以及语义，它的 SCADE 语言参考官方手册 [42] 中定义了它的语言结构，包括了它的类型系统、命名空间结构、词法和语法定义、静态语义和动态语义等。Scade2Nu 中的文本模型解析器使用了 ANTLR 解析工具 [43, 44]，由于 ANTLR 生态中还没有 SCADE 语言的语法文件，因此我们根据 SCADE 语言参考手册将 SCADE 语言的静态语义写进 ANTLR 特定格式的语法文件中。

4.1.1　Scade2Nu 中的 ANTLR 语法文件

Scade2Nu 支持并可对包含以下 SCADE 语法的模型进行解析操作：

SCADE 语言的顶层结构定义

顶层结构定义了 SCADE 语法中的程序（Program）语义和声明（Declaration）语义。Program 由一系列 Declaration 块构成，这些块的顺序互不影响，一般依赖于 SCADE 文本模型的顺序，同时 Program 会是 SCADE 抽象语法树的根节点。Declaration 可以包含一些必要的声明，包括数据类型声明、常数声明、用户自定义操作符的声明等。如图4.2所示是 ANTLR 语法文件的中顶层结构的语义。

```
program     : decls*
            ;

decls       : type_block
            | const_block
            | user_op_decl
            ;
```

图 4.2: 顶层结构语义

类型声明和常量声明语义

SCADE 语言中的类型（Type）用于建立数据类型。一个类型块（type_block）以关键词 type 开始，支持常用的数据类型以及用户自定义的枚举类型。预定义的类型有布尔类型（Boolean），整型（Integer），实数型（Real）以及字符类型（Char）。常量声明以关键词 const 开始，后跟一个标志符（ID）和‘:’以及含类性表达式或值。如图4.3所示是 Scade2Nu 解析器中的类型和常量声明的语义。

自定义操作符节点声明语义

自定义操作符节点（User-Defined Operators）声明 SCADE 模型的核心部分，也就是用来定义用户建立的操作符节点，这些操作符可以是功能函数（Function），也可以是状态机或数据流节点（Node）。模型的输入、输出变量也会在这一部分声明。如图4.4所示，变量声明中一般包含了变量名（identifier）及其类型（type_expr）。

这些变量作为参数（params）声明在节点名（ID）的后面，如图4.5所示，操作主体则由 opt_bdy 定义。主体（opt_bdy）中主要是一些局部变量以及用‘let’关键词

包围的等式。

```
type_block      : type (type_decl ';')*
                ;
type_decl       : ID ('=' type_def)?
                ;
type_def        : type_expr
                | 'enum' '{' ID (',' ID)* '}'
                ;
type_expr       : BOOL
                | INT
                | REAL
                | CHAR
                | ID
                ;
const_block     : 'const' (const_decl SEMI)*
                ;
const_decl      : ID ':' type_expr (= expr)?
                ;
```

图 4.3: 类型与常量声明语义

```
var_decls       : identifer (',' identifer)* ':' type_expr
                ;
```

图 4.4: 变量声明语义

等式及表达式的语义

等式（equations）分为两类，一类是常规的简单等式（simple_equation），主要是以数据流方式主导；另一类则是由控制流（control_block）主导，控制类等式一般包含 SCADE 状态机（state_machine）。等式又由表达式构成，如图4.6所示也包含了 Scade2Nu 中可解析的表达式（expr），基本表达式（basic expression）包括了变量名以及常量，其他表达式主要分为基本的数值运算操作、逻辑运算操作、时态运算操作、比较操作以及分支操作等。

SCADE 状态机语义

SCADE 状态机的声明以关键词 automaton 开始，主体在状态声明（state_decl）部分。SCADE 状态机会有多个状态，这些状态也会被关键词标记，例如标记该状态

```
user_op_decl
: op_kind ID params 'returns' params
  opt_body
;

op_kind
: 'function'
| 'node'
;

opt_body
: equations ';'
| (local_block)?
  'let' (equations ';')* 'tel' (';')?
;
```

图 4.5: 自定义操作符节点声明语义

```
equations
: simple_equations
| control_block returna
;

expr
: basic expression
| expr '/' expr
| expr '*' expr
| expr '+' expr
| expr '-' expr
| expr 'mod' expr
| expr '=' expr
| expr '<>' expr
| expr '<' expr
| expr '>' expr
| expr '<=' expr
| expr '>=' expr
| '!' expr
| expr 'and' expr
| expr 'or' expr
| expr 'xor' expr
| 'if' expr 'then' expr 'else' expr
| 'pre' expr
| expr 'times' expr
| expr '->' expr
| 'fby' '(' (expr (',' expr)*)? ';'
  expr ';' (expr (',' expr)*)? ')'
| ID '(' (expr (',' expr)*)? ')'
...
```

图 4.6: 等式与表达式语义

是否为初始状态（initial）。转移条件（transitions）会在 unless 或 until 中进行声明，转移条件是布尔类型的表达式。一个状态中的操作定义在数据声明（data_def）中，这些数据声明有许多等式构成，由于状态机是控制流等式（control_block），因此当这些等式中存在另外的控制流等式时，该状态机则具有层次化的特点。如图4.7所示是 SCADE 状态机的主要语义。

```
state_machine
    : 'automaton' ID state_decl+
    ;

state_decl
    : 'initial'? 'final'? 'state' ID
      ('unless' (transitions ';')+)?
      data_def
      ('until' (transitions ';')*)?
    ;
```

图 4.7: SCADE 状态机的语义

4.1.2　SCADE 模型语法树遍历

由于 SCADE 开发环境是一个成熟的商业工具，对 SCADE 语言的解析并没有开源工具可使用。而 ANTLR Runtime API 可以为我们减少开发时间，提高开发效率。通过 ANTLR 工具生成的基本访问接口，Scade2Nu 文本模型解析器实现了遍历 SCADE 语法树的监听器，从而供之后符号表容器的装载程序调用。

4.2　Scade2Nu 符号表容器

转换规则需要有一个特定的输入，该输入必须包含 SCADE 状态机模型的所有信息。通常在编译领域，符号表（symbol table）被用来作为解析程序后存放信息的容器，也是一种数据结构。因此，我们在本章定义 SCADE 状态机的符号表容器。该容器接收 SCADE 文本模型的抽象语法树，通过构建装载函数，调用我们实现的语法树监听器将信息装载到符号表中。这样 Scade2Nu 便可在其之上实现上一章中定义的转换规则，从而将 SCADE 状态机转换到 NuSMV 输入模型上。由于 SCADE 状态机支持层次化结构，因此符号表结构也需支持层次化状态机。

定义 4.2.1. *一个层次化 SCADE 状态机（SSM）是一个 5 元组 $\langle I, O, SM, Op, Fun\rangle$，其中：*

- *I 是输入变量的集合，*

- *O 是输出变量的集合，*

- *SM 是集合 $\{SM_1, ..., SM_n\}$，其中每个 SM_i 代表着一个子状态机结构，*

- Op 是该状态机中使用到的运算操作符块的集合，

- Fun 该状态机调用的用户自定义的函数功能节点集合。

层次化状态机 [45, 46] 包含许多子状态机，SSM 中的 SM 是子状态机的集合，因此 SSM 可以看作是最顶层的状态机。同时 SCADE 状态机还拥有输入变量和输出变量，这些变量时刻影响着状态机的运行。状态中可以存在一些数据流操作，而这些操作都会使用到一些运算操作符块。该状态机 SSM 用到的所有操作符块组成集合 Op。SCADE 支持用户将一些比较复杂的运算单独建立在一个函数节点中，可供建模时复用。SSM 的状态中就能够调用这些函数节点，而这些用户自定义的函数节点组成集合 Fun。我们还需继续定义子状态机的结构，

定义 4.2.2. 令一个子状态机结构 $SM_i \in SM$ 也是一个元组 $\langle S, s_0, Sub, \hookrightarrow, Act\rangle$，其中：

- S 是子状态机 SM_i 中的状态的集合，

- s_0 是它的初始状态，

- Sub: $S \to \mathcal{P}(S)$ 是关于它们子状态机关系的一个函数映射，

- $\hookrightarrow \subseteq S \times G(Var) \times S$ 代表了状态迁移关系的集合，

- $Act \subseteq S \times O \times 2^{Op} \times 2^{Fun}$ 代表了在该子状态机中各个状态中各种事件、行为操作的集合。

函数 Sub 定义了一个状态和其子状态的映射关系，$\mathcal{P}(S)$ 代表了状态 $s \in S$ 的子状态机集合。而 $G(Var)$ 则是由变量组成的转移条件表达式，当该表达式为真则发生状态转移，并且对于一个子状态机，转移只会发生在同一层次的状态之间。集合 Act 中的元素用来刻画特定状态中对特定输出变量的操作行为，这些状态中有时会使用到运算操作符或者用户自定义的函数节点。

以图4.8中的状态机为例，该 SCADE 状态机实例有一个输入变量 x 和一个输出变量 o，它拥有两个子状态机 SM_1 与 SM_2，该 SSM 的状态用到了加法运算操作符模块($plus$)、乘法运算操作符模块($multi$)以及一个用户自定义的函数节点(fun)，通过以上定义可以将其表示为 $SSM = \langle\{x\},\{o\},\{SM_1, SM_2\},\{plus, multi\},\{fun\}\rangle$。而对于子状态机 SM_1 而言，它的状态集合表示为 $S = \{A, B\}$，初始状态 $s_0 = A$；由于状态 B 的子状态机是 SM_2，所以子状态关系 $(B, SM_2) \in Sub$。Act 的元素代表了某状态中对特定输出变量的操作，因此 SM_1 中有 $(A, o, \{plus, multi\}, \varnothing) \in Act$，而对于 SM_2 中则有 $(D, o, \varnothing, \{fun\}) \in Act$。

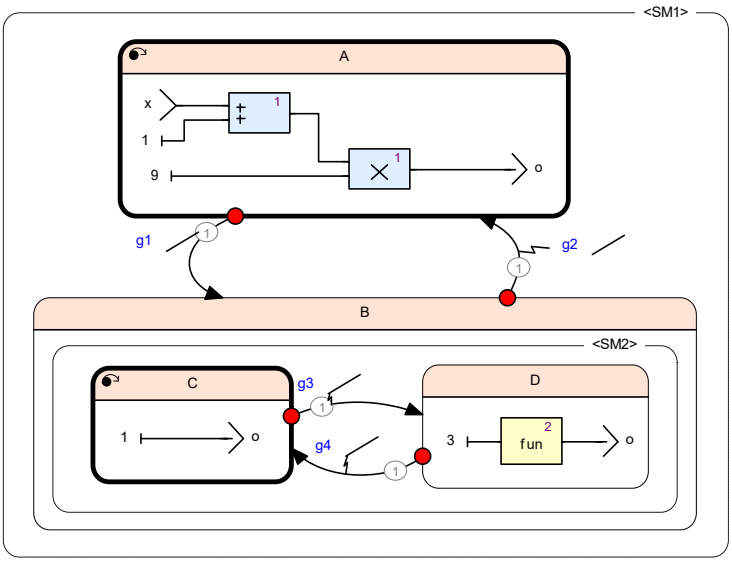


图 4.8: 一个 SCADE 状态机（SSM）的符号表实例

符号表定义确定后便可以将其构建为数据结构，然后将 Scade2Nu 文本解析器中得到的 SCADE 状态机的语法树信息通过写好的遍历器载入到符号表中，得到 SCADE 状态机的实例。不过，在将它放入到转换器模块来生成最终的 NuSMV 模型之前，还需对 SCADE 模型中的变量进行一些优化。

4.3　Scade2Nu 模型变量边界优化

SCADE 用来设计与开发安全攸关领域的反应式控制系统，系统的规模也会对 Scade2Nu 的验证增加挑战。而状态爆炸一直是模型检查中的一个重要问题 [47]。许

多抽象优化方法用来减少状态爆炸发生的概率，现在版本的 NuSMV 由于同时支持基于 BDD 和基于 SAT 的模型检查，大大提高了它解决状态爆炸的能力 [28]。但实际 SCADE 应用的变量区间可能较大，因此即使像 NuSMV 这样使用符号模型检查技术，当变量规模太大也会很难验证。Scade2Nu 提供了对变量边界的优化选项，从而减弱模型中的状态爆炸。该技术也属于一种简单的抽象技术 [48]，通过设置变量的边界來降低 NuSMV 模型中状态的复杂性，减少状态空间，提高验证效率。后面本文会在案例分析中阐述在使用 Scade2Nu 过程中关于变量边界设置的经验教训。

4.4　Scade2Nu 模型转换器

Scade2Nu 模型转换器的工作是将 SCADE 状态机符号表 SSM 的实例转换到最终的 NuSMV 输入模型。从架构图4.1中可以看到转换器通过转换算法生成 4 类 NuSMV 模型中的模块（`MODULE`）代码，分别是：

1. 子状态机 `MODULE SMi`：这部分的每一个模块都对应于 SCADE 状态机中的一个子状态机 SMi，包括了它所含的状态、状态转移关系、触发器参数、监控参数（monitor parameters）赋值以及对它拥有的子状态机的实例化；

2. 变量监控 `MODULE Set_Var`：这部分每一个模块都通过监控参数控制着对应的输出变量 Var 的赋值操作，是 SCADE 变量监控机制的重要部分；

3. 自定义函数节点 `MODULE Function`：SCADE 模型中若使用到了用户自定义的数据流函数节点，Scade2Nu 会将它们生成为 NuSMV 中的函数模块；

4. 顶层状态机模块 `MODULE main`：该模块代表着 SCADE 状态机中最顶层的状态机，负责模型变量和参数声明、子状态机实例化、变量与参数初始化、函数节点实例化等工作，它有且只有一个，主导着整个生成的模型。

接下来阐述 Scade2Nu 用到的转换算法，该算法基于上一章中定义的转换规则，生成以上 4 类的目标模型代码：

4.4.1 变量监控模块

SCADE 变量监控机制需要该类模块来对输出变量进行赋值与操作，这些模块在每个周期都会读取监控参数的值，然后进行相应的赋值操作。算法1描述了Scade2Nu 如何得到变量监控模块，它的输入是一个 SCADE 状态机（SSM）的符号表容器，算法通过加载该 SSM 得到 SCADE 状态机的信息。每个子状态机的状态行为存储在集合 $\mathcal{A}$ 中，因此算法 1 在一开始初始化了该集合，并随后遍历 SM 中的子状态机元素，将每一个 SM_i 中的 Act 与集合 $\mathcal{A}$ 取并操作。继续遍历每一个输出变量 v，为它们生成对应的变量监控 `MODULE Set_v`。算法会依次生成 NuSMV 模型的关键词，以及转换规则中定义的模块中的参数，包括所有的变量、与该输出变量相关的监控参数以及调用到的函数。最后从 $\mathcal{A}$ 中找到对应于该变量 v 的赋值操作，基于转换规则的定义生成对 v 的赋值表达式。

Algorithm 1 变量监控模块的转换算法

Input: $SSM = (I, O, SM, Op, Fun)$

1: $\mathcal{A} \leftarrow \emptyset$;
2: **for** each $SM_i \in \{SM_1, ..., SM_n\}$ **do**
3: $Act_i \leftarrow Act$;
4: $\mathcal{A} \leftarrow \mathcal{A} \cup Act_i$;
5: **end for**
6: **for** each $v \in O$ **do**
7: 生成 `MODULE Set_v(variables, monitor parameter,`
8: `function parameters)`;
9: 生成关键词 `INIT`;
10: 初始化输出变量 v;
11: 生成关键词 `ASSIGN`;
12: **for** each $(s, v', op, fun) \in \mathcal{A}$ that $v' = v$ **do**
13: 根据转换规则生成对应的赋值表达式语句;
14: **end for**
15: **end for**

4.4.2　子状态机模块

Scade2Nu 的转换引擎根据层次化结构转换规则和 SCADE 变量监控机制这两类规则来生成 NuSMV 目标模型中的各个子状态机模块。算法2中展示了生成它们的算法，Scade2Nu 转换器将 SSM 作为输入，遍历每一个状态机 SMi。`MODULE SMi` 中会声明一个变量 state 用来存放该状态机的所有状态名，当该状态机有子状态时，就会实例化它的子状态机。首先算法通过每一个子状态机关系确定该声明哪些子状态机的触发器参数 active 与 default。然后通过规则来为每一个子状态机 SM_{Sub} 生成它们的触发器参数赋值等式，并实例化该子状态机。然后再基于规则来生成子状态机的状态转移。

SCADE 变量监控机制要求在一个子状态机模块中还需生成对监控参数的赋值语句。我们需要从状态转移集合 $\hookrightarrow$ 中筛选出监控参数赋值所需要的那些转移，然后基于规则为它们赋值。

4.4.3　自定义函数节点模块

自定义函数节点模块生成依赖于 SSM 中的集合 Fun，Scade2Nu 支持转换那些用数据流方式建立的函数节点。Fun 中的每个节点都有各自的输入变量、输出变量以及等式，因此可以直接将它们转向 NuSMV 中的等式。对于节点中的输入输出变量，Scade2Nu 直接将它们生成在模块名后的参数表中。对于节点中声明的有关输出变量的等式，Scade2Nu 将其生成在函数模块中的 `ASSIGN` 代码块中，并将其余的等式生成在 `DEFINE` 代码块中。

4.4.4　顶层状态机模块

最顶层的状态机的工作将在 `MODULE main` 中完成，该模块需要对许多变量作声明，并要生成监控参数。对各变量和参数初始化的工作也在这部分完成。算法3给出了生成 `MODULE main` 的全部过程，它接收一个状态机实例 SSM，当 SM 不为空时进行主模块生成。

Algorithm 2 子状态机模块的转换算法

Input: $SSM = (I, O, SM, Op, Fun)$

for each $SM_i \in \{SM_1, ..., SM_n\}$ **do**
  生成 `MODULE SMi(variables,`
  `monitor parameters, default, active);`
  生成关键词 `VAR;`
  if $S \neq \emptyset$ **then**
    声明状态变量 `state;`
  end if
  if $Sub \neq \emptyset$ **then**
    for each $(s, \mathcal{P}(s)) \in Sub$ **do**
      $SM_{sub} \leftarrow \mathcal{P}(s)$
      for each $sub_{sm} \in SM_{sub}$ **do**
        生成关键词 `VAR;`
        根据规则 1 对 `Def_s` 进行声明和赋值;
        生成关键词 `DEFINE;`
        根据规则 2 定义 sub_{sm}`_active;`
        生成关键词 `VAR;`
        实例化 sub_{sm};
      end for
    end for
  end if
  生成关键词 `ASSIGN;`
  根据 s_0 初始化 SM_i 的状态变量 `state;`
  for each $(s, g, t) \in \hookrightarrow$ **do**
    根据规则生成状态转移赋值语句;
  end for
  生成关键词 `ASSIGN;`
  for each $(s, v, op, fun) \in Act$ **do**
    若存在 $(s', g, t) \in \hookrightarrow$ 并满足 $s' = s$ ，则使用参数赋值规则;
    若存在 $(p, g, s'') \in \hookrightarrow$ 并满足 $s'' = s$ ，则使用参数赋值规则 4;
  end for
end for

Algorithm 3 顶层状态机模块的转换算法

Input: $SSM = (I, O, SM, Op, Fun)$

生成语句 `MODULE main`;
if $n > 0$ **then**
  $\mathcal{M}_{para} \leftarrow \emptyset$; $\mathcal{A} \leftarrow \emptyset$;
  生成关键词 `VAR`;
  声明顶层状态机的状态变量 $state = \{SM\}$;
  声明触发器参数 `default` 和 `active`;
  生成关键词 `VAR`;
  for each $v \in I$ **do**
    声明输入变量;
  end for
  for each $v \in O$ **do**
    声明输出变量;
  end for
  for each $SM_i \in \{SM_1, ..., SM_n\}$ **do**
    $Act_i \leftarrow Act$;
    $\mathcal{A} \leftarrow \mathcal{A} \cup Act_i$;
  end for
  for each $(s, v, op, fun) \in \mathcal{A}$ **do**
    $\mathcal{M}_{para} \leftarrow \mathcal{M}_{para} \cup \{set_{(v,s)}, reset_{(v,s)}\}$
  end for
  声明集合 $\mathcal{M}_{para}$ 中的监控参数;
  生成关键词 `VAR`;
  实例化变量监控 module、函数节点 module 以及子状态机 module
  生成关键词 `ASSIGN`;
  初始化状态变量 $state$;
  基于规则对顶层状态机的触发器参数赋值;
  for each $m \in \mathcal{M}_{para}$ **do**
    初始化监控参数 m;
  end for
else
  return ;
end if

算法3使用 $\mathcal{M}_{para}$ 来存放监控参数，顶层的状态变量有且仅有一个状态，并将其命名为 SM，然后再声明一些变量，包括顶层状态机的触发器参数、输入变量以及输出变量。由于监控参数的个数由输出变量和对影响输出变量的状态决定，因此这里依旧需要用到该 SSM 所有的状态行为集合 $\mathcal{A}$，然后对 $\mathcal{A}$ 中的行为逐个遍历，对于每一个行为元素，生成一组监控参数，并与集合 $\mathcal{M}_{para}$ 取并集，得到完整的监控参数集合并将它们逐个声明。变量监控模块、函数节点模块以及子状态机模块都会在主模块中被实例化，之后再进行初始化和赋值操作。首先初始化状态变量 $state$，然后基于规则对顶层状态机的触发器参数 $active$ 与 $default$ 进行赋值，最后初始化监控参数。

附录A中给出了一个简单的案例模型，从该模型可以看出一个完整的 NuSMV 目标模型由以上四类模块的组成，Scade2Nu 主页[1]中也有更多的目标模型。

4.5　本章小结

本章详细讲述了 SCADE 状态机验证工具 Scade2Nu 的架构设计，并且详细介绍了它的主要组件：文本模型解析器、符号表容器以及模型转换器。

[1]Scade2Nu 主页, 见 https://sites.google.com/view/scade2nu/

第五章　Scade2Nu 的案例研究与评估

本章讲述如何使用 Scade2Nu 来验证实际安全攸关领域中的系统案例，与此同时说明本文验证方案的正确性与可行性。案例是用 SCADE 设计的模型，模型的设计以控制流为主导，使用 SCADE 状态机来对系统进行描述。通过对案例系统安全需求性质进行验证，来检验和完善 SCADE 设计的模型。

5.1　Scade2Nu 在实际领域中的应用

5.1.1　平稳驾驶系统

平稳驾驶系统（Cruise Control System）是一个能够自动控制机动车驾驶速度的系统。驾驶员通过设置该速度能使机动车自己接管油门来维持平稳的行驶车速。Esterel 公司提供了该系统的案例，包括模型的详细的参数信息 [49, 50]，如下表5.1所示是该系统的输入与输出变量的描述。

如图5.1所示是 Esterel 公司设计提供的平稳驾驶控制系统的状态机模型，在顶层状态机下有三个子状态机 SM1、SM2 和 SM3。SM1 有两种状态分别代表系统的开启（On）和关闭（Off）。当系统处于关闭状态时，油门指令的值完全由加速踏板传感器的值决定。当机动车驾驶员启用平稳驾驶控制系统时，状态机将变为开启状态并且系统开始运行。子状态机 SM2 执行平稳速度的控制，它有两种状态分别是调节（Regulation）和中断（Interrupt）。如果驾驶员刹车，系统将被中断，之后若想恢复需要在松开刹车后重启。在调节状态下，驾驶员可以设置、增加和降低平稳驾驶速度。SM3 是调节状态的子状态机，当它处于托管（RegulOn）状态时，机动车的速度和油门指令将完全由平稳驾驶系统控制。在 SM3 中，驾驶员可以加速以

表 5.1: 平稳驾驶系统变量描述

变量名	描述	数据类型	变量类别
On	启动平稳驾驶控制	布尔类型	输入
off	关闭平稳驾驶控制	布尔类型	输入
CurSpeed	机动车速度传感器	整型	输入
Resume	在刹车后重启平稳驾驶控制	布尔类型	输入
Set	设置当前行驶速度为新的平稳驾驶速度	布尔类型	输入
Accel	油门踏板压力传感器	整型	输入
Break	刹车压力传感器	整型	输入
QuickDecel	降低平稳驾驶速度	布尔类型	输入
QuickAccel	增加平稳驾驶速度	布尔类型	输入
CruiseSpeed	平稳驾驶速度	整型	输出
ThrottleCmd	油门指令值	整型	输出

提高汽车的速度，而状态也转向待机状态（StandBy），在加速完后返回托管状态。当然，当汽车的行驶速度超出范围时，它也将转移到待机状态，并将油门控制权交给驾驶员。

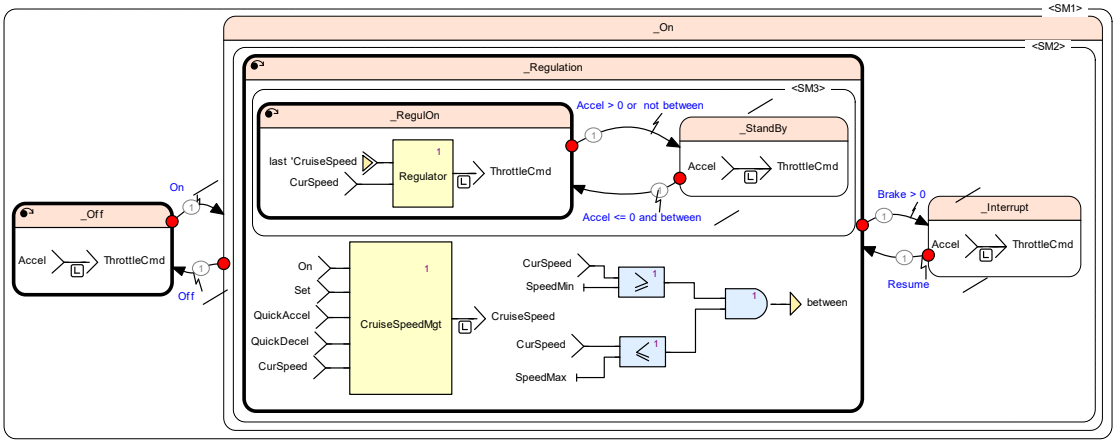


图 5.1: 平稳驾驶控制系统的 SCADE 状态机模型

我们从系统的安全需求中抽取三条，使用 Scade2Nu 来验证它们。下表列出了这三个需要验证的安全性需求：

- 平稳驾驶控制系统总是处于关闭状态，那么油门指令的值也总是受油门传感

器控制；

- 当系统处于平稳驾驶控制中，机动车刹车后系统总会进入中断或关闭状态；
- 对于所有的执行路径而言，若驾驶员关闭平稳驾驶控制，则下个周期系统必然进入关闭状态。

使用 Scade2Nu 对 SCADE 状态机进行模型检查

通过以上系统的描述，我们使用 Scade2Nu 来验证系统的三个性质，从而检验本文方案的有效性和可行性。SCADE 支持用户将模型导出为一个文本模型文件。在 Scade2Nu 中新建完验证项目后，我们便可以导入该文本模型。在此之前我们还需要设置模型的变量边界，从而优化在 NuSMV 模型检查器中的状态规模，边界值如表5.2所示。之后通过转换菜单便能够在对应目录下生成 NuSMV 的模型文件。与此同时，还需将需要验证的需求性质用 LTL 或 CTL 时序性质规范来描述，然后通过 Scade2Nu 后台调用的 NuSMV 模型检查器就能够验证系统是否满足这些性质。表5.3与表5.4分别是性质的形式化规范描述以及对应的验证结果。验证结果中若是 False 的表中会给出反例，由于反例较长，这里仅显示能够反馈出模型错误的状态及其变量值。

验证结果分析

NuSMV 给出的验证结果表明性质 1 是正确的，也就是说该系统满足需求 1 的描述。而 NuSMV 验证出性质 2 与性质 3 并不能被满足，并分别给出了反例。我们通过分析反例来验证结果的有效性，并调试模型。

从需求 2 的反例中可以看出，驾驶员在平稳驾驶系统正在运行过程中踩下了刹车，因此系统进入了中断状态。紧接着驾驶员重启了系统，使其系统进入了调节状态，但由于 Break 的值并没有改变，驾驶员依旧踩着刹车。需求 2 要求刹车后系统得中断或者关闭的，反例给出的情况表明，驾驶员刹车未结束但是系统却恢复到了调节状态，这确实不能够满足需求 2 的要求。回看 SCADE 状态机模型，我们

表 5.2: 平稳驾驶系统变量边界优化设置

变量名	左边界	有边界	默认值
On	-	-	false
off	-	-	false
CurSpeed	0	15	0
Resume	-	-	false
Set	-	-	false
Accel	0	10	0
Break	0	10	0
QuickDecel	-	-	false
QuickAccel	-	-	false
CruiseSpeed	0	15	0
ThrottleCmd	0	10	0

知道需求 2 主要描述的是调节状态（Regulation）与中断状态（Interrupt）之间的迁移关系，并发现中断状态到调节状态的迁移条件仅仅是要求重启（Resume）。根据反例我们得知该转移的发生还需保证刹车停止才能满足需求 2，因此可以将该转移条件修改为 `Resume and not (Break > 0)`。

通过分析需求 3 的反例，我们发现系统从关闭状态到开启状态的迁移过程竟然可以在变量 Off 为 True 时发生，该执行路径显然违背了需求 3 的性质，因此需要修改关闭状态到开启状态的转移条件。为了确保每一个执行路径，驾驶员关闭平稳驾驶系统都能进入关闭状态，该转移条件应修改为 `On and not Off`。

表 5.3: 平稳驾驶系统需求的时态逻辑规范描述

序号	时态逻辑表达式
1	`G((SM_SM1.state = _Off & G !On) -> Accel = next(ThrottleCmd))`
2	`G(SM_SM1.state = _On & X Brake > 0 ->` `X (SM_SM1.Sub_SM2.state = _Interrupt | SM_SM1.state = _Off))`
3	`AG(Off -> AX SM_SM1.state = _Off)`

表 5.4: 平稳驾驶系统性质验证结果

序号	验证结果	反例
1	True	
2	False	-> State: 6.3 <- Brake = 1 SM_SM1.Sub_SM2.state = _Interrupt -> State: 6.4 <- Resume = TRUE SM_SM1.Sub_SM2.state = _Regulation ...
3	False	-> State: 4.1 <- Off = FALSE On = FALSE SM_SM1.state = _Off -> State: 4.2 <- Off = TRUE On = TRUE -> State: 4.3 <- Off = FALSE On = FALSE SM_SM1.state = _On ...

5.1.2 轨交信号灯系统

轨交系统是安全攸关领域中一个非常接近我们现实生活的实际应用。而与普通的交通场景相比，火车与公路的交界处则更需要严格把控。交叉路口信号灯控制系统（Level Crossing System）则是一个能够自动控制信号灯和道闸升降的系统，当接收到远处有列车来时，系统则会使信号灯变红，并放下道闸。这也是保证交通安全避免意外的最基本的措施。如图5.2所示，Esterel 公司提供了该系统的 SCADE 模型，它是一个非层次化的状态机模型，由一个子状态机构成，该状态机有 4 个状态。状态 RedWithoutTrain 代表交叉路口轨道区域附近没有火车且信号灯显示为红灯；该状态是初始状态，并且此时的道闸是完全打开的。当火车来后，系统则转移到状态 RedWithTrain，这时控制系统需要放下道闸，阻止机动车通过十字路口。当火车离开后，系统切换信号灯至绿色，并抬起道闸。状态中道闸的放下与抬起操作通过调用用户自定义的数据流函数节点来完成。如果火车在绿灯时过来，则需要

先显示黄灯提醒机动车司机，随后切换到红灯，具体的状态转移可在图中看到。

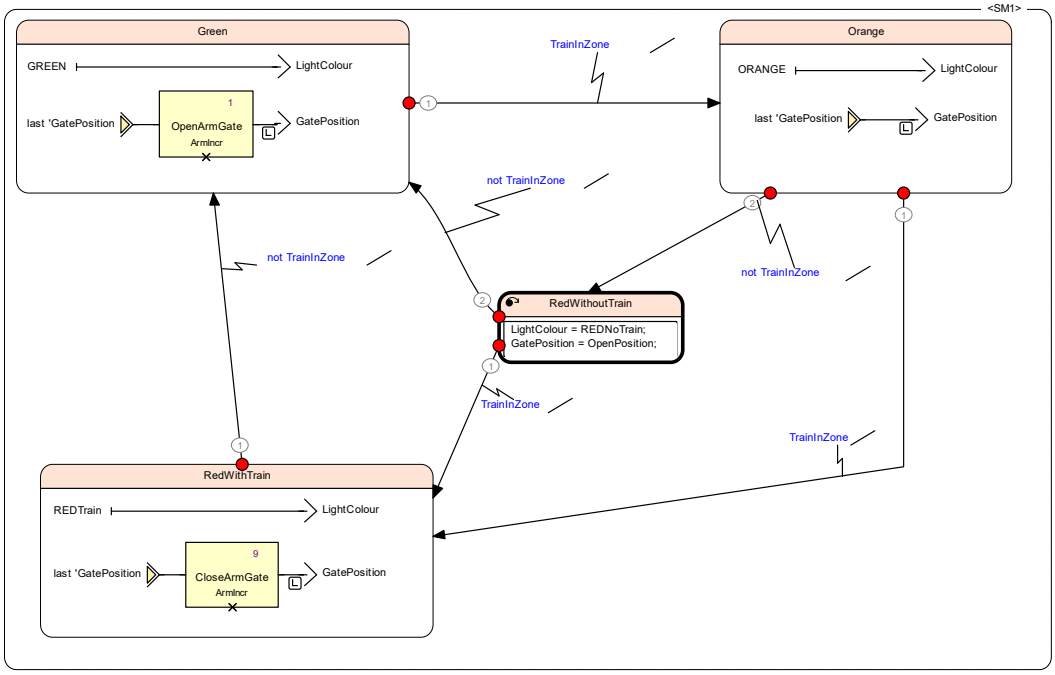


图 5.2: 交叉路口信号灯控制系统的 SCADE 状态机模型

SCADE 模型拥有它对应的输入变量和输出变量，该系统中变量的描述如下表5.5所示：

表 5.5: 交叉路口信号灯系统的变量描述

变量名	描述	数据类型	变量类别
TrainInZone	火车是否在附近区域	布尔类型	输入
LightColour	信号灯的颜色	枚举类型	输出
GatePosition	道闸角度	整型	输出
OpenPosition	道闸全开的角度值 100	整型	常量
ClosePosition	道闸关闭的角度值 0	整型	常量

我们需要验证这个案例的以下两个安全需求，用自然语言描述为：

- 当火车在附近区域时，交叉路口信号灯控制系统总是能够确保信号灯不显示绿灯；

- 当信号灯为绿灯时，火车在未来某个时刻进入附近区域，那么系统必须确保信号灯会切换到黄灯以更加醒目地提醒驾驶员。

使用 Scade2Nu 对 SCADE 状态机进行模型检查

按照之前的方法，我们需要设置变量边界。由于该模型中变量 LightColour 的数据类型是一个用户自定义的枚举类型，因此在设置变量时只需设置它的默认值。与此同时，该模型还定义了常量，常量名可直接用于边界值的设定，表5.6中是该模型变量的边界值。

表 5.6: 交叉路口信号灯系统变量边界的优化值设置

变量名	左边界	右边界	默认值
TrainInZone	-	-	false
LightColour	OpenPosition	ClosePosition	OpenPosition
GatePosition	-	-	RedNoTrain

将需求性质用 LTL 规范公式描述后如表5.7所示，然后通过 Scade2Nu 将转换后的模型同规范性质一起输入到 NuSMV 中得到验证结果。该系统能够满足这两条性质，因此这次不用再修改 SCADE 模型。

表 5.7: 交叉路口信号灯系统需求的时态逻辑规范及验证结果

序号	时态逻辑表达式	验证结果
1	`G(TrainInZone -> X LightColour != GREEN)`	True
2	`G((LightColour = GREEN) & (F TrainInZone)) -> (F LightColour = ORANGE)`	True

5.1.3　卫星控制系统

在航空航天领域中会有各种各样的控制系统，这些系统的错误会导致巨大的财产损失，因此这些控制系统的共性之一则是需要确保设计的系统满足它们的安

全需求。本小节的卫星控制系统由上海航天控制技术研究所提供，该系统是一个子控制系统，用来控制卫星的各个不同工作模式的切换，覆盖了卫星与火箭的分离过程以及分离后卫星的工作过程。

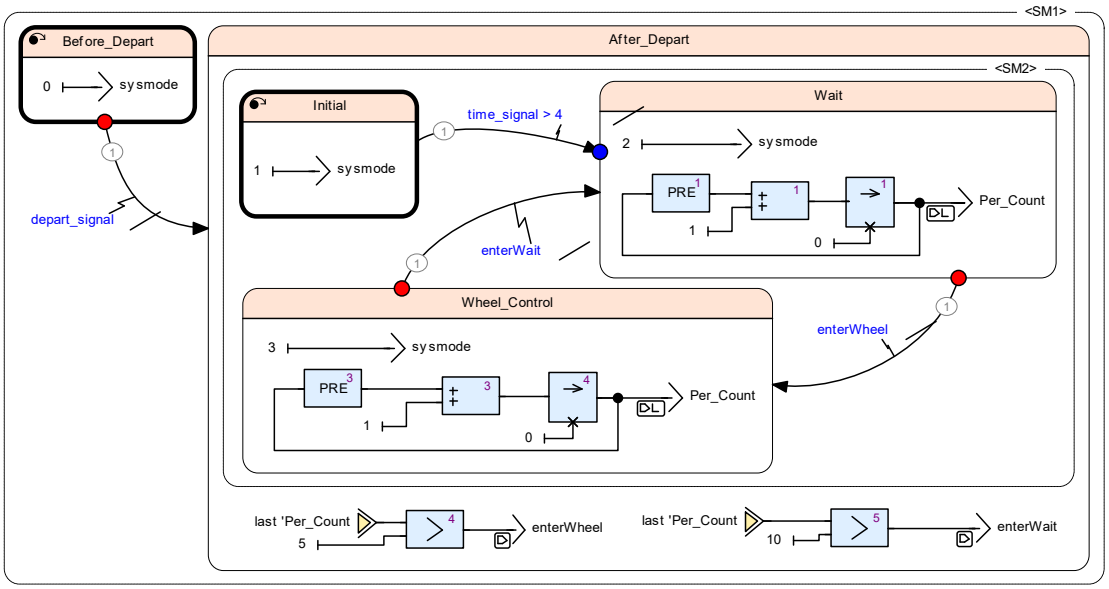


图 5.3: 卫星控制系统的 SCADE 状态机模型

如图5.3所示是该子系统的 SCADE 状态机模型，它有两个子状态机 SM1 与 SM2，分别用来描述星箭分离过程与星箭分离后卫星的工作模式。因此，SM1 拥有两个状态，即未分离状态（BeforeDepart）与分离后状态（AfterDepart）；SM2 拥有三个状态，分别代表了速度阻尼模式（Initial）、等待模式（Wait）以及飞轮控制模式（WheelControl）。每一个子状态机的状态都会为系统输出一个工作模式信号，用来判断控制系统处于哪个工作模式下。卫星一开始会接收到一个星箭分离信号，根据星箭分离信号进入分离后状态，激活速度阻尼模式。速度阻尼模式下会监听一个时间信号，该时间信号描述的是环境监测到的分离后持续的时间，时间信号超过一段规定值后会使卫星进入等待模式。进入等待模式会激活一个节拍计数器，该计数器用来记录切换到该工作模式后持续的时间，当满足一定条件后便正式进入卫星飞轮控制模式。飞轮控制模式是大多数卫星长期运行和进行业务工作的模式，该模式也拥有一个节拍计数器，待到相应时间后继续回到等待模式，从而节省卫

星能源。如下表5.8所示是该状态机拥有的输入变量和输出变量：

表 5.8: 卫星控制系统的变量描述

变量名	描述	数据类型	变量类别
depart_signal	星箭分离信号	布尔类型	输入
time_signal	分离后持续的时间信号	整型	输入
sysmode	卫星系统的工作模式信号	整型	输出
Per_Count	节拍计数器	整型	输出
enterWheel	确定何种切换条件才能进入飞轮控制模式	布尔类型	输出
enterWait	确定何种切换条件才能进入等待模式	布尔类型	输出

这里我们使用 Scade2Nu 验证两条安全性质，分别描述如下：

- 卫星控制系统在星箭分离后总是会先进入速度阻尼状态，当接收到的时间信号大于 4 个时间单元时再进入等待状态；
- 星箭分离后，卫星控制系统进入等待模式，随后再经过 5 个时间单元进入飞轮控制模式开始工作，再经过 10 个时间单元回到等待模式。

使用 Scade2Nu 对 SCADE 状态机进行模型检查

Scade2Nu 能够将 SCADE 模型转换成 NuSMV 的输入模型，当然工程师需要优化并设置该系统中的变量的边界，将其写入变量配置文件 VarBound 中。VarBound 的书写格式很简单，将下表5.9的内容直接填入文件即可。

在该步骤工程师还需要做的工作便是将以上自然语言描述的需求用 NuSMV 中的逻辑规范表达式表示出来。从需求中可以看出，卫星控制系统对时序的要求要更大。之前的系统需求仅仅只描述了状态之间的前后关系，而对状态之间转换的时间间隔不做要求。但在卫星控制系统中需要根据各个工作模式的时间节拍来进行模式的切换，因此这里我们会使用到带时间边界的时态逻辑来规范系统的安全需求。下表5.10是卫星控制系统需求的时态逻辑规范描述，使用 Scade2Nu 我们能够得到验证信息：系统能够满足需求 1；系统无法满足性质 2。

表 5.9: 卫星控制系统变量边界的优化值设置

变量名	左边界	有边界	默认值
depart_signal	-	-	false
time_signal	0	10	0
sysmode	0	3	0
Per_Count	0	12	0
enterWheel	-	-	false
enterWait	-	-	false

表 5.10: 卫星控制系统需求的时态逻辑规范描述

序号	时态逻辑表达式	验证结果
1	G((sysmode = 0 & departSignal -> (X sysmode = 1)) & (sysmode = 1 & X timeSignal > 4 -> X sysmode = 2))	True
2	G((SM_SM1.state = AfterDepart & SM_SM1.Sub_SM2.state = Initial & X SM_SM1.Sub_SM2.state = Wait) -> (G[1,6] SM_SM1.Sub_SM2.state = Wait & G[7,17] SM_SM1.Sub_SM2.state = WheelControl))	False

验证结果分析

NuSMV 给出的验证结果表明性质 1 是正确的，然而系统不能够满足性质 2，因此这里产生了可供工程师分析的反例。如图5.4所示是 Scade2Nu 提供的另一种观察反例的方式，以表格可视化的形式将反例的状态路径显示出来。每一行对应了路径中的一个状态，每一列对应了该状态下变量的取值。分析需求 2 的描述我们可以得知，卫星系统必须满足以下条件：切换到等待模式后（花费 1 个时间单元），还需继续持续 5 个时间单元，之后切换到飞控控制模式（花费 1 个时间单元）并工作 10 个时间单元。这些时间单元通过输出变量 Per_Count 记录，图中也给出了反例执行路径中该值的变化，分析可视化反例表格，我们发现当前的模型在系统模式

（sysmode）切换到等待模式（sysmode = 2）后，节拍计数器又记录了 6 个时间单元，同样地当前系统在系统模式切换到飞轮控制模式（sysmode = 3）后，节拍计数器又经过了 11 个时间单元。而导致该路径发生的原因则是切换条件的设计错误。从原 SCADE 模型中看到切换条件变量 enterWheel 与 enterWait 的赋值操作中都使用了严格大于操作符（>），我们将其修改为大于等于操作符（≥）。再次验证后，系统则能满足性质 2 的需求。

LTL： G (((SM_SM1.state = AfterDepart & SM_SM1.Sub_S...

PerCount	sysmode	enterWait	enterWheel
0	0	FALSE	FALSE
0	0	FALSE	FALSE
0	1	FALSE	FALSE
0	2	FALSE	FALSE
1	2	FALSE	FALSE
2	2	FALSE	FALSE
3	2	FALSE	FALSE
4	2	FALSE	FALSE
5	2	FALSE	FALSE
6	2	FALSE	FALSE
0	3	FALSE	TRUE
1	3	FALSE	FALSE
2	3	FALSE	FALSE
3	3	FALSE	FALSE
4	3	FALSE	FALSE
5	3	FALSE	FALSE
6	3	FALSE	FALSE
7	3	FALSE	TRUE
8	3	FALSE	TRUE
9	3	FALSE	TRUE
10	3	FALSE	TRUE
11	3	FALSE	TRUE
0	2	TRUE	TRUE

图 5.4: 卫星控制系统性质 2 的可视化反例

5.2　Scade2Nu 可视化验证平台及其评估

使用 Scade2Nu，用户可以验证 SCADE 状态机模型的安全需求，从而找出模型的设计错误，直到需求能够被系统全部满足。而在安全攸关领域，再小的错误也有可能造成极严重的后果，因此在这些领域的系统再安全也不为过。Scade2Nu 的核心模块是两种模型之间的转换器，为了使用户在使用时可以直观地看到模型，

我们开发了可视化的交互界面。如图5.5所示，Scade2Nu 软件界面的中间是 SCADE 原模型和生成的 NuSMV 模型，最右侧的控制台显示了 NuSMV 的验证结果。由于 Scade2Nu 还整合了 NuSMV 的可视化工具 NuSeen，所以用户可以在中间下方的反例可视化窗口中看到 NuSMV 检查器生成的反例。界面左下方是用户为 SCADE 模型中的变量设置的边界值和初始值，以便优化目标模型中的状态空间。最初将 SCADE 模型转换到 NuSMV 模型时，我们使用它的默认范围，并没有为变量设置边界，于是当时 NuSMV 花了很多小时才能够得到验证结果。之后使用了变量边界优化策略后，NuSMV 的验证速度得到了非常大的提升，同时缩小了模型的状态空间，而两次的验证结果都是相同的。这也应证了之前提到的模型变量边界优化的有效性。由于 SCADE 模型中的变量往往反映的是环境或设备输出，所以边界一般缩小为它们的十分之一，因此并不会对模型的实际意义产生影响。不过目前 Scade2Nu 还不能自动设置变量边界，合适边界的设置需要由工程师指定，因此在这方面缺少一些自动化。

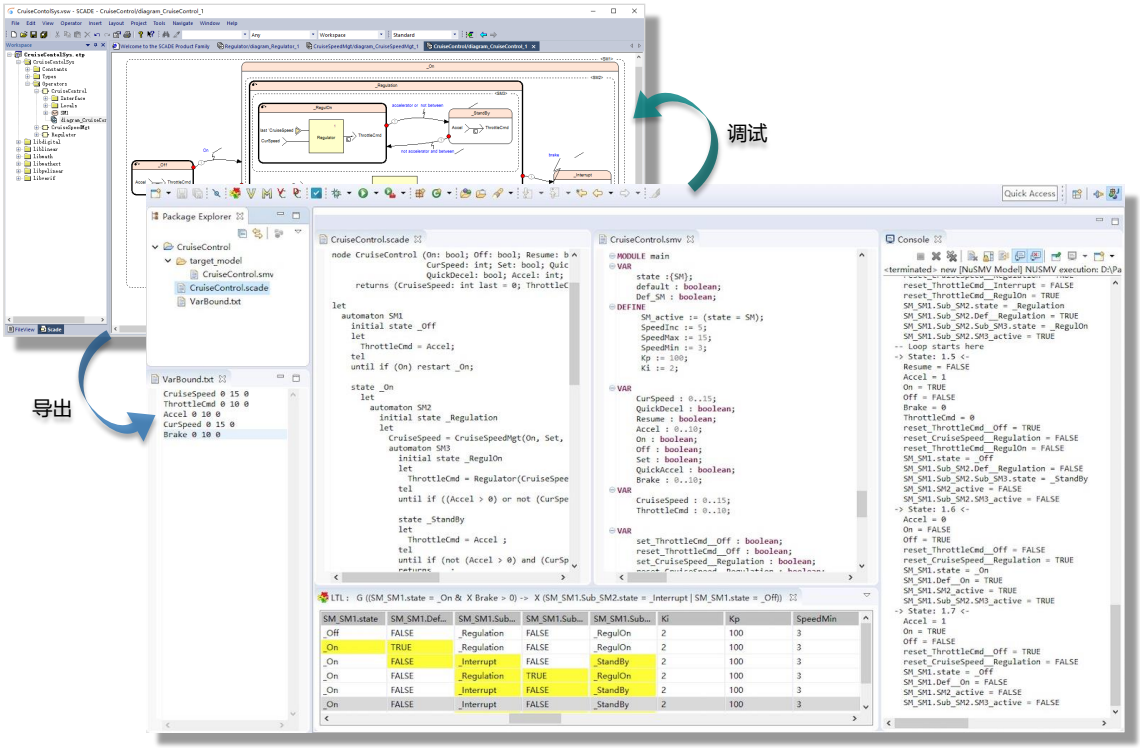


图 5.5: Scade2Nu 验证平台的交互界面

Scade2Nu 的优势在于它为 SCADE 模型引入了新的需求规范的表达方式，SCADE 用户不再局限于使用 Design Verifier 来对需求进行图形化建模，还可以使用 LTL 和 CTL 逻辑表达式来描述性质。由于 NuSMV 支持 LTL 与 CTL 的表达，所以这也拓展了 SCADE 对系统需求的描述能力。最显著的描述能力体现在时序相关的需求上，与 LTL 和 CTL 相比，SCADE 语言无法描述诸如总是（Globally）、最终（Finally）这种无边界的时态逻辑，并且 NuSMV 中提供的带时间边界的 LTL、实时 CTL 等也使 Scade2Nu 可以描述更复杂、更精确的时序性质。

5.3　本章小结

本章通过对 Scade2Nu 在多个实际领域中案例的应用说明了它的可行性和有效性。通过案例，本章完整地展示了用 Scade2Nu 对 SCADE 状态机模型进行需求性质验证的过程，同时也讲述了使用 Scade2Nu 过程中的经验和不足。

第六章　总结与展望

6.1　总结

在安全攸关控制领域，基于模型的开发方式备受青睐，它高效的交互方式使工程师专注于系统功能的设计，无需考虑到代码实现，而这正是软件开发方式的理想方向。SCADE 作为该领域的一款流行产品，早在 2003 年就被 Esterel 科技公司收购，并一直助力安全控制系统的开发。形式化验证能够在系统设计阶段检验工程师开发模型的正确性，该验证技术使用严格的形式化语义来描述系统、模型以及需求性质，表达准确且无二义性，通过模型检查算法可以验证系统的正确性，从而保证软件的安全与可靠。Design Verifier（DV）就是 SCADE 的形式化验证组件，能够验证模型的安全性质，但随着现代安全控制软件应用场景的丰富，对软件需求的苛刻和多样，用户亟须更强需求表达能力的验证组件。而 Design Verifier 的描述能力是不够的，尤其在时序性质方面其表达能力不及时态逻辑规范。本文提出了验证 SCADE 状态机的新方案，并开发了验证工具 Scade2Nu，旨在拓展 SCADE 的需求性质表达能力和对它们的验证能力，在系统设计阶段进一步减少潜在的设计错误，从而使工程师能够开发更加安全可靠的控制系统和应用程序。本文的主要贡献如下：

1. 本文将线性时态逻辑（LTL）、计算树逻辑（CTL）引入到 SCADE 需求性质的描述规范中，提出了基于模型转换的一套 SCADE 模型时序性质验证方法。本文提出了 SCADE 状态机到 NuSMV 输入模型的转换规则，该转换规则基于 STP 方法，包括了 SCADE 状态机中初始状态、子状态机、状态转移的转换。同时本文提出了 SCADE 变量监控机制，该机制改进了 STP 方法中的监控方

式，同时也帮助转换后的 NuSMV 模型能够操作变量。因此，通过本文的验证方法我们便可将状态机主导的 SCADE 模型转换至 NuSMV 的输入模型上，待写好需求性质规范后便可对模型进行模型检查。

2. 本文形式化定义了 SCADE 层次化状态机结构，用于描述 SCADE 层次化状态机的特点。与此同时，本文基于该结构和转换规则，设计了从 SCADE 状态机模型到 NuSMV 输入模型的转换算法。转换算法将生成 NuSMV 目标模型中的四类代码块，分别是子状态机模块、变量监控模块、函数节点模块以及顶层状态机模块。

3. 本文进一步设计并实现了 SCADE 安全性质分析工具 Scade2Nu。本文基于以上研究，开发 SCADE 模型的时序性质验证工具 Scade2Nu，该工具拥有三个核心组件分别是文本模型解析器、符号表容器、以及模型转换器。我们开发了可视化桌面应用，提高工程师的交互体验，对于简单的安全需求性质，工程师可选择不用再在 DV 中进行图形化建模，而仅仅只需写下一个规范表达式；对于那些 DV 不能表达的时序性质，Scade2Nu 能够提供应有的表达能力。通过将 NuSMV 模型检查工具整合进 Scade2Nu，工程师便能进行完整的 SCADE 模型验证流程，通过分析验证结果来发现系统设计中发生的潜在错误，修正 SCADE 模型。

6.2 展望

尽管 Scade2Nu 能够对 SCADE 状态机模型进行验证，但仍有许多可做的工作和不足之处：

1. 本文提出的验证方法主要适用于以状态机建模方式为主导的 SCADE 模型，但由于 SCADE 是一个成熟的商业工具，并且它的工业用途涵盖面很广，因此现实中无法避免遇到以数据流为主导的 SCADE 模型，为了能够支持更多复杂应用场景下的 SCADE 模型，未来 Scade2Nu 还需支持处理 SCADE 语义中

复杂的时态操作符以及高阶操作符。

2. 本文假设了 SCADE 状态机里的转移条件都是相同优先级的，并且状态机中的初始状态优先级大于迁移条件。SCADE 的状态转移中还分弱转移、强转移同步转移，对于混合优先级的状态转移的情况，本文的方法还有一些局限性，因此未来工作中需要对这一方面进行拓展和补充。

3. 最后，由于 Scade2Nu 是一个可视化桌面软件，如何优化它的界面、使用方式来提高用户体验也是一个值得考虑的问题。

附录 A　模型示例

A.1　NuSMV 目标模型

```
MODULE SM1(TrainInZone, LightColour, GatePosition,
set_LightColour_Green, reset_LightColour_Green,
..., -- 继续声明与SM1有关的其他控参数
default, active)
DEFINE
  OpenPosition := 10;
  ClosedPosition := 0;
  ArmIncr := 1;
VAR
  state : {RedWithTrain, Green, Orange, RedWithoutTrain};
ASSIGN
  init(state) := RedWithoutTrain;
  next(state) :=
    case next(active) & next(default) : RedWithoutTrain;
      active & state = RedWithoutTrain & !TrainInZone : Green;
      ...; -- 状态转移关系
      active & state = Green & TrainInZone : Orange;
      TRUE : state;
    esac;

  next(set_LightColour_Green) := active & next(active)
      & case
        state = Green & !(TrainInZone) : TRUE;
        TRUE : FALSE;
      esac;
  next(reset_LightColour_Green) := active & next(active)
      & case
        state = RedWithoutTrain & !TrainInZone : TRUE;
        state = RedWithTrain & !TrainInZone : TRUE;
        TRUE : FALSE;
      esac;
  ... -- 继续赋值该状态机对应的监控参数
```

Listing A.1: NuSMV 目标模型示例-子状态机模块

```
MODULE Set_LightColour(TrainInZone, LightColour, GatePosition,
    set_LightColour_Green, reset_LightColour_Green, ...,
    set_LightColour_RedWithTrain, reset_LightColour_RedWithTrain)
DEFINE
  OpenPosition := 10;
  ClosedPosition := 0;
  ArmIncr := 1;
INIT
  LightColour = REDNoTrain;
ASSIGN
  next(LightColour) :=
    case
      next(set_LightColour_Green) : GREEN ;
      next(reset_LightColour_Green) : GREEN ;
      ...; -- 继续根据监控参数得到输出变量LightColour的赋值语句
      TRUE : LightColour;
    esac;

MODULE Set_GatePosition(TrainInZone, LightColour, GatePosition,
    set_GatePosition_Orange, reset_GatePosition_Orange, ...,
    set_GatePosition_RedWithTrain,
    reset_GatePosition_RedWithTrain, func1, func2)
DEFINE
    OpenPosition := 10;
    ClosedPosition := 0;
    ArmIncr := 1;
INIT
  GatePosition = OpenPosition;
ASSIGN
  next(GatePosition) :=
    case
      next(set_GatePosition_Orange) : GatePosition ;
      next(reset_GatePosition_Orange) : GatePosition ;
      ...; -- 继续根据监控参数得到输出变量GatePosition的赋值语句
      TRUE : GatePosition;
    esac;
```

Listing A.2: NuSMV 目标模型示例-监控变量模块

```
MODULE CloseArmGate(CurrentPos, Incr)
DEFINE
  OpenPosition := 10;
  ClosedPosition := 0;
  ArmIncr := 1;
VAR
  GatePosition : ClosedPosition..OpenPosition;
DEFINE
  L1 := L4 >= L5;
  L2 := L5 - L6;
```

```
  L3 := L1 ? ( L5 ) : ( L2 );
  L4 := ClosedPosition;
  L5 := CurrentPos;
  L6 := Incr;
ASSIGN
  GatePosition :=
    case
       ( L3 >= ClosedPosition &  L3 <= OpenPosition) :  L3;
       ( L3 < ClosedPosition) : ClosedPosition;
       ( L3 > OpenPosition) : OpenPosition;
    esac;
```

Listing A.3: NuSMV 目标模型示例-自定义函数节点模块

```
MODULE main
VAR
  state :{SM};
  default : boolean;
  Def_SM : boolean;
DEFINE
   SM_active := (state = SM);
   OpenPosition := 10;
   ClosedPosition := 0;
   ArmIncr := 1;
VAR
  TrainInZone : boolean;
VAR
  LightColour : {GREEN, ORANGE, REDTrain, REDNoTrain};
  GatePosition : ClosedPosition..OpenPosition;

VAR
  set_LightColour_Green : boolean;
  reset_LightColour_Green : boolean;
  set_GatePosition_Orange : boolean;
  reset_GatePosition_Orange : boolean;
  set_GatePosition_RedWithoutTrain : boolean;
  reset_GatePosition_RedWithoutTrain : boolean;
  set_LightColour_Orange : boolean;
  reset_LightColour_Orange : boolean;
  set_GatePosition_Green : boolean;
  reset_GatePosition_Green : boolean;
  set_LightColour_RedWithoutTrain : boolean;
  reset_LightColour_RedWithoutTrain : boolean;
  set_GatePosition_RedWithTrain : boolean;
  reset_GatePosition_RedWithTrain : boolean;
  set_LightColour_RedWithTrain : boolean;
  reset_LightColour_RedWithTrain : boolean;
VAR
   func1 : OpenArmGate(GatePosition, ArmIncr);
```

```
36    func2 : CloseArmGate(GatePosition, ArmIncr);
37 VAR
38   LightColour_set : Set_LightColour(TrainInZone, LightColour,
      GatePosition, set_LightColour_Green, reset_LightColour_Green,
       set_LightColour_Orange, reset_LightColour_Orange,
      set_LightColour_RedWithoutTrain,
      reset_LightColour_RedWithoutTrain,
      set_LightColour_RedWithTrain, reset_LightColour_RedWithTrain)
      ;
39
40   GatePosition_set : Set_GatePosition(TrainInZone, LightColour,
      GatePosition, set_GatePosition_Orange,
      reset_GatePosition_Orange, set_GatePosition_RedWithoutTrain,
      reset_GatePosition_RedWithoutTrain, set_GatePosition_Green,
      reset_GatePosition_Green, set_GatePosition_RedWithTrain,
      reset_GatePosition_RedWithTrain, func1, func2);
41 VAR
42    SM_SM1 : SM1(TrainInZone, LightColour, GatePosition,
      set_LightColour_Green, reset_LightColour_Green,
      set_GatePosition_Orange, reset_GatePosition_Orange,
      set_GatePosition_RedWithoutTrain,
      reset_GatePosition_RedWithoutTrain, set_LightColour_Orange,
      reset_LightColour_Orange, set_GatePosition_Green,
      reset_GatePosition_Green, set_LightColour_RedWithoutTrain,
      reset_LightColour_RedWithoutTrain,
      set_GatePosition_RedWithTrain,
      reset_GatePosition_RedWithTrain, set_LightColour_RedWithTrain
      , reset_LightColour_RedWithTrain, Def_SM, SM_active);
43 INIT
44    default = TRUE;
45 ASSIGN
46    init(Def_SM) := TRUE;
47    next(default) := FALSE;
48    next(Def_SM) := FALSE;
49 INIT
50    state = SM;
51 TRANS
52    state = SM & next(state) = state;
53 ASSIGN
54    init(set_LightColour_Green):= FALSE;
55    ... -- 继续其他监控参数的初始化步骤;
56    init(reset_LightColour_RedWithTrain):= FALSE;
```

Listing A.4: NuSMV 目标模型示例-顶层主函数模块

参考文献

[1] 杜泽民, 陈宜成. 基于模型驱动的嵌入式软件需求验证研究 [J]. 电子世界, 2018(8) : 208 – 208.

[2] 王庆胜, 朱罕. 模型驱动开发在安全苛求系统中的应用 [J]. 城市轨道交通研究, 2016, 19(z2).

[3] ANSYS. ANSYS SCADE Suite: Model-Based Development[C/OL] // . . https://www.ansys.com/products/embedded-software/ansys-scade-suite.

[4] 胡春凤. 基于 SCADE 的 ATP 建模与验证 [D]. [S.l.] : [s.n.], 2018.

[5] ABDULLA P A, DENEUX J, STÅLMARCK G, et al. Designing safe, reliable systems using scade[C] // International Symposium On Leveraging Applications of Formal Methods, Verification and Validation. 2004 : 115 – 129.

[6] CLARKE E, BIERE A, RAIMI R, et al. Bounded model checking using satisfiability solving[J]. Formal methods in system design, 2001, 19(1) : 7 – 34.

[7] DASKAYA I, HUHN M, MILIUS S. Formal safety analysis in industrial practice[C] // International Workshop on Formal Methods for Industrial Critical Systems. 2011 : 68 – 84.

[8] BENGTSSON J, LARSEN K, LARSSON F, et al. UPPAAL—a tool suite for automatic verification of real-time systems[C] // International Hybrid Systems Workshop. 1995 : 232 – 243.

[9] SCHNOEBELEN P. The Complexity of Temporal Logic Model Checking.[J]. Advances in modal logic, 2002, 4(393-436) : 35.

[10] LETTNIN D, NALLA P K, RUF J, et al. Verification of temporal properties in automotive embedded software[C] // Proceedings of the conference on Design, automation and test in Europe. 2008 : 164 – 169.

[11] WIELS V. Formal Methods in Aerospace: Constraints, Assets and Challenges[C] // . .

[12] HALBWACHS N, CASPI P, RAYMOND P, et al. The synchronous data flow programming language LUSTRE[J]. Proceedings of the IEEE, 1991, 79(9) : 1305 – 1320.

[13] HAGEN G E. VERIFYING SAFETY PROPERTIES OF LUSTRE PROGRAMS: AN[D]. [S.l.] : The University of Iowa, 2008.

[14] HAGEN G, TINELLI C. Scaling up the formal verification of Lustre programs with SMT-based techniques[C] // Formal Methods in Computer-Aided Design, 2008. FMCAD'08. 2008 : 1 – 9.

[15] BASOLD H, GÜNTHER H, HUHN M, et al. An open alternative for SMT-based verification of SCADE models[C] // International Workshop on Formal Methods for Industrial Critical Systems. 2014 : 124 – 139.

[16] COLAÇO J-L, PAGANO B, POUZET M. SCADE 6: A formal language for embedded critical software development[C] // 2017 International Symposium on Theoretical Aspects of Software Engineering (TASE). 2017 : 1 – 11.

[17] BAIER C, KATOEN J-P. Principles of model checking[M]. [S.l.] : MIT press, 2008.

[18] DEVROEY X, CORDY M, SCHOBBENS P-Y, et al. State machine flattening, a map-

ping study and tools assessment[C] // Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on. 2015 : 1 – 8.

[19] HAREL D. Statecharts: A visual formalism for complex systems[J]. Science of computer programming, 1987, 8(3) : 231 – 274.

[20] HAREL D, PNUELI A. On the development of reactive systems[G] // Logics and models of concurrent systems. [S.l.] : Springer, 1985 : 477 – 498.

[21] LATELLA D, MAJZIK I, MASSINK M. Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker[J]. Formal aspects of computing, 1999, 11(6) : 637 – 664.

[22] LILIUS J, PALTOR I P. Formalising UML state machines for model checking[C] // International Conference on the Unified Modeling Language. 1999 : 430 – 444.

[23] HOLZMANN G J. The model checker SPIN[J]. IEEE Transactions on software engineering, 1997, 23(5) : 279 – 295.

[24] MCMILLAN K L. Symbolic model checking[G] // Symbolic Model Checking. [S.l.] : Springer, 1993 : 25 – 60.

[25] CHAN W, ANDERSON R J, BEAME P, et al. Model checking large software specifications[J]. IEEE Transactions on software Engineering, 1998, 24(7) : 498 – 520.

[26] HAREL D, LACHOVER H, NAAMAD A, et al. Statemate: A working environment for the development of complex reactive systems[J]. IEEE Transactions on software engineering, 1990, 16(4) : 403 – 414.

[27] CLARKE E M, HEINLE W. Modular translation of Statecharts to SMV[R]. [S.l.] : Citeseer, 2000.

[28] CIMATTI A, CLARKE E, GIUNCHIGLIA E, et al. Nusmv 2: An opensource tool for symbolic model checking[C] // International Conference on Computer Aided Verification. 2002 : 359 – 364.

[29] MENDLER M. Cycle-based Programming of Distributed Systems: The Synchrony Hypothesis[C/OL] // . 2009.
http://www.ibr.cs.tu-bs.de/cm/events/tubs.CITY-symposium-2009/slides/Mendler_Michael.pdf.

[30] CASPI P, CURIC A, MAIGNAN A, et al. From Simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications[C] // ACM Sigplan Notices : Vol 38. 2003 : 153 – 162.

[31] ANDRÉ C. Semantics of SSM (safe state machine)[J]. I3S Laboratory–UMR, 2003, 6070.

[32] FORNARI X. Understanding how SCADE suite KCG generates safe C code[J]. White paper, Esterel Technologies, 2010.

[33] 陈淑珍, 陈荣武, 李耀. 基于 SCADE 的安全软件开发方法研究 [J]. 铁路计算机应用, 2015(3) : 14 – 18.

[34] BRYANT R E. Graph-based algorithms for boolean function manipulation[R]. [S.l.] : CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE, 2001.

[35] BURCH J R, CLARKE E M, MCMILLAN K L, et al. Symbolic model checking: 1020 states and beyond[J]. Information and computation, 1992, 98(2) : 142 – 170.

[36] CIMATTI A, CLARKE E, GIUNCHIGLIA F, et al. NuSMV: a new symbolic model checker[J]. International Journal on Software Tools for Technology Transfer, 2000, 2(4) : 410 – 425.

[37] CAVADA R, CIMATTI A, JOCHIM C A, et al. Nusmv 2.4 user manual[J]. CMU and ITC-irst, 2005.

[38] ARCAINI P, GARGANTINI A, VAVASSORI P, et al. NuSeen: an eclipse-based environment for the NuSMV model checker[C] // Eclipse-IT 2013: Proceedings of VIII Workshop of the Italian Eclipse Community : Vol 2464. 2013.

[39] DORMOY F-X. Scade 6: a model based solution for safety critical software development[C] // Proceedings of the 4th European Congress on Embedded Real Time Software (ERTS'08). 2008 : 1 – 9.

[40] COLAÇO J-L, PAGANO B, POUZET M. A conservative extension of synchronous data-flow with state machines[C] // Proceedings of the 5th ACM international conference on Embedded software. 2005 : 173 – 182.

[41] COLAÇO J-L, HAMON G, POUZET M. Mixing signals and modes in synchronous data-flow systems[C] // Proceedings of the 6th ACM & IEEE International conference on Embedded software. 2006 : 73 – 82.

[42] ESTEREL TECHNOLOGIES I. SCADE Language - Reference Manual 2.1.[J], 2015.

[43] PARR T J, QUONG R W. ANTLR: A predicated-LL (k) parser generator[J]. Software: Practice and Experience, 1995, 25(7) : 789 – 810.

[44] PARR T. The definitive ANTLR 4 reference[M]. [S.l.] : Pragmatic Bookshelf, 2013.

[45] ALUR R, YANNAKAKIS M. Model checking of hierarchical state machines[C] // ACM SIGSOFT Software Engineering Notes : Vol 23. 1998 : 175 – 188.

[46] ALUR R, YANNAKAKIS M. Model checking of hierarchical state machines[J]. ACM Transactions on Programming Languages and Systems (TOPLAS), 2001, 23(3) : 273 – 303.

[47] CLARKE E M, KLIEBER W, NOVÁČEK M, et al. Model checking and the state explosion problem[C] // LASER Summer School on Software Engineering. 2011 : 1 – 30.

[48] MEENAKSHI B, BHATNAGAR A, ROY S. Tool for translating simulink models into input language of a model checker[C] // International Conference on Formal Engineering Methods. 2006 : 606 – 620.

[49] HEIM S, DUMAS X, BONNAFOUS E, et al. Model Checking of SCADE Designed Systems[C] // 8th European Congress on Embedded Real Time Software and Systems (ERTS 2016). 2016.

[50] TEAM P. Cruise Control Software Requirements[C] // . 2014.

致　　谢

我还能非常清晰地回忆起当年入学时的场景，自己对研究生生活心怀憧憬，而如今却已步入校园生活的尾声，除了感慨日月如梭外，更觉得自己在这儿收获了人生中的一段非常宝贵的经历。这几年的美好时光离不开老师和同学，现趁毕业论文即将完成之际，向陪伴我近三年成长的老师、同窗们表示最衷心的感谢。

首先我要感谢我的导师黄滟鸿老师以及史建琦老师。感谢史老师帮助我选择课题，为我的学术和科研工作指明了方向。记得每次与史老师讨论问题时都能够从中获得新的想法和更成熟的思路，让我在学习实践的道路上少走了许多歪路；我还要感谢导师黄老师，当我临近论文投稿，黄老师熬夜帮我细心地修改论文，给我提供了非常宝贵的修改建议，没有这些我也很难有现在的成果。老师们严谨治学、敢于拼搏、不畏艰险的精神给了我很大的启迪，让我终身受益。

其次我要感谢学院的老师们，你们精心准备的课程给我打下了理论基础，你们的谆谆教导也拓宽了我思维的边界。同时我还要特别感谢我的研究生辅导员刘国艳老师，她总是非常善解人意，在我遇到困难时能够帮助我、关心我、鼓励我，让我感受到了关怀和温暖。

我也要感谢我的舍友以及实验室的伙伴们，感谢你们在我困难时候给予我的帮助和关心，我会倍感珍惜我们之间的友谊。

我还要感谢我的家人、我的未婚妻，你们给了我坚强的后盾和坚持的勇气。

而我最要感谢的是我的母亲，她一直是我动力最主要的来源。

最后，再一次衷心祝愿大家前程似锦、幸福安康！

施 健

二零一九年三月

攻读硕士学位期间发表论文、参与科研和获得荣誉情况

■ 已完成学术论文

[1] Scade2Nu : A Tool for Verifying Safety Requirements of SCADE Models with Temporal Specifications. 已被 25th International Conference on Requirements Engineering: Foundation for Software Quality 2019 接受.（CCF C 类）

■ 已申请的发明专利

[1] 一种软件体系结构建模和仿真系统 [P]，华东师范大学，201611137739.2，2016-12-12

■ 参与的科研课题

[1] 上海市科学技术委员会重大项目：可信工业控制器基础软件与逻辑组态开发环境研制，项目编号：16DZ1100603

[2] 国家自然科学基金青年项目：安全工控程序分解与转换的形式化理论研究，项目编号：61602178