面向航空控制软件的形式化建模技术报告

文档作者：冯劲草

学号：51151500089

目录

[0.引言 2](#_Toc493678672)

[1. 绪论 3](#_Toc493678673)

[1.1 嵌入式控制软件需求建模的特点与主要挑战 3](#_Toc493678674)

[1.2 航空发动机控制软件的特征分析 5](#_Toc493678675)

[1.3 传统建模方式的问题 7](#_Toc493678676)

[1.4 国内外研究现状 7](#_Toc493678677)

[1.5 建模方法概述 9](#_Toc493678678)

[2. AEDL建模语言 12](#_Toc493678679)

[2.1 语言概述 12](#_Toc493678680)

[2.2 AEDL语法 16](#_Toc493678681)

[2.2.1 AEDL模型 16](#_Toc493678682)

[2.2.2 模式图 16](#_Toc493678683)

[2.2.3 迁移卫式 18](#_Toc493678684)

[2.2.4 控制流 19](#_Toc493678685)

[2.2.5 模块 20](#_Toc493678686)

[2.2.6 模式 21](#_Toc493678687)

[2.2.7 数据字典 21](#_Toc493678688)

[2.2.8 单位 22](#_Toc493678689)

[2.2.9 其它 22](#_Toc493678690)

[2.3 操作语义 24](#_Toc493678691)

[2.3.1 模式层操作语义 24](#_Toc493678692)

[2.3.2 控制流层操作语义 29](#_Toc493678693)

[3. AEDL建模方案 31](#_Toc493678694)

[3.1 需求模板撰写规范 31](#_Toc493678695)

[3.1.1模式（Mode） 31](#_Toc493678696)

[3.1.2 流程（Procedure） 33](#_Toc493678697)

[3.1.3 任务（Task） 33](#_Toc493678698)

[3.1.3 数据字典（Data Dictionary） 34](#_Toc493678699)

[3.1.4 需求模板提供 35](#_Toc493678700)

[3.2 建模过程 39](#_Toc493678701)

[3.2.1 模型示例 40](#_Toc493678702)

[4. AEDL工具设计 41](#_Toc493678703)

[4.1 需求文档存储 41](#_Toc493678704)

[4.1.1 存储结构 41](#_Toc493678705)

[4.1.2 中间模型设计 43](#_Toc493678706)

[4.1.3文档抽取 44](#_Toc493678707)

[4.2 中间模型结构 45](#_Toc493678708)

[4.3 撰写可分析形式化需求的Antlr语法 45](#_Toc493678709)

[4.4 错误诊断 47](#_Toc493678710)

[4.4.1 变量检查 47](#_Toc493678711)

[4.4.2 类型检查 47](#_Toc493678712)

[4.5 静态分析 47](#_Toc493678713)

[4.5.1 Dataflow analysis思路 47](#_Toc493678714)

[4.5.2 Dataflow analysis 算法 48](#_Toc493678715)

[4.6 模型操作语义 53](#_Toc493678716)

[4.6.1 新定义函数 53](#_Toc493678717)

[4.6.2时间算子 54](#_Toc493678718)

[4.6.3模式操作语义 56](#_Toc493678719)

[4.6.4 控制流流的控制语义 59](#_Toc493678720)

[5. 结论 61](#_Toc493678721)

[硕士期间发表的学术论文、专著、重要科研成果 62](#_Toc493678722)

# 0.引言

在华东师范大学攻读专业硕士学位期间，实验室与中国航空商用发动机有限公司有一个科研合作项目。我有幸参与了此次科研项目，在这个过程中我收货颇丰。这篇技术报告作为此项工作的总结，同时也是我硕士研究生期间参与科研工作的一份专业技术报告。

嵌入式控制软件作为各种控制系统的核心，广泛分布于社会生产和生活中。各种航天军事装备例如火箭和无人机，工业生产中的核电控制系统，社会生活中的轨道交通控制系统和各种智能穿戴设备等，其系统的功能正确性、安全性和可靠性等均依赖于其内嵌的控制软件。由于大部分嵌入式控制软件直接操控安全攸关系统，例如核电站和飞行器的控制设备，一旦控制软件未能按照设计要求工作，则可能造成巨大的物质损失和人员伤亡。根据当前软件工程的理论和实践，人们认识到，从需求分析入手，提供有效的软件需求描述与确认方法，是从源头上确保嵌入式控制软件质量的方法。

而商用航空发动机控制软件是典型的安全攸关的嵌入式控制软件，其功能正确性、安全性和可靠性直接决定了飞行安全。针对航空发动机控制软件的质量保障，DO-333标准给出了建议，航空发动机控制软件应考虑采用软件的形式化方法并以此引导软件全生命周期的研发活动。

为此，我们提出了一种发动机控制软件需求建模的形式化工程方法，以软件形式化方法理论为基础，以系统化的工程方法引导工程人员构建高质量的软件需求模型，用以引导后续的代码编写和相关测试分析。

根据DO-333标准相关内容，结合航空发动机软件的特征，我们制定了航空发动机控制软件的形式化建模技术报告，用于规范发动机控制软件的形式化建模方法与流程，并给出建模方案。

# 1. 绪论

## 1.1 嵌入式控制软件需求建模的特点与主要挑战

在深入探讨嵌入式控制软件需求建模的具体方法之前，有必要对其特征进行总结，由此则可提出针对性的方法。综合来看，嵌入式控制软件有如下特征：

* 功能正确性要求高

嵌入式周期软件通常需要执行安全攸关的操作任务，例如列车在规定时间内完成制动行为等，软件是否正确实现了预期需求极其重要。

* 软硬件结合程度高

有别于其他类型的软件系统，嵌入式控制软件和所涉设备硬件以及外部环境的交互频繁。

* 可靠性要求高

与普通民用软件不同，终止软件、暂停服务、重启等故障恢复手段对于安全攸关的嵌入式控制软件往往不可接受。因此，嵌入式控制软件需要足够可靠。

由于嵌入式控制软件的上述特点，如何确保嵌入式控制软件质量极其重要。然而，嵌入式控制软件的研制又存在诸多难点：

* 成本约束严格

大量嵌入式控制软件涉及国防安全和经济建设的重要设施，其开发周期和成本受到严格地限制，一旦超期或超支，可能带来巨大损失。

* 需求内容复杂

嵌入式控制软件往往涉及复杂而精密的系统控制行为，例如列车运行的动力行为控制，因此需求文档往往庞大而包含复杂的功能和性能描述。

* 变更频繁

嵌入式控制软件是所涉控制系统总体方案的具体实现。在软件开发过程中，总体方案常会变化，用户需求可能随着软件设计的逐步推进而发生变更。特别是在大型控制系统开发后期，由于改动硬件的成本较高，用户往往选择对软件功能做出修改。

* 各个部门联系紧密

嵌入式控制软件的开发通常涉及不同行业与部门。软件工程师需要和行业专家有效沟通，正确理解需求，才能开发出符合需求的软件产品。例如，列车控制软件的需求描述往往需要车辆工程人员和软件设计人员共同完成。

由于嵌入式控制软件的特点和需求描述过程中存在的难点，随着嵌入式软件功能日趋复杂，各种软件缺陷和漏洞难以避免，这就是常说的“软件可信任”问题。

软件的“可信”是指软件系统的动态行为及其结果总是符合人们的预期。然而在现实中软件可信性总是不尽如人意。造成这样的现象其本质原因在于：软件需求作为软件开发的基础，其完备性（completeness）与正确性（correctness）难以得到保障，进而影响了软件质量。完备性指的是软件需求规约充分地将用户对软件的预期能力刻画出来；正确性指的是软件需求规约忠实准确地描述了用户对软件功能的期望。

形式化方法被认为是保证软件需求质量的重要手段，主要思想是建立形式化规约，用形式化规约语言精确地描述用户对软件的需求，通过对规约的逐步精化和验证得到可信的软件系统。形式方法包含两项重要技术：形式化规约（formal specification）与形式化验证（formal verification）。前者关心的是形式化建模，即关注如何用精确的、无二义性的数学语言来书写形式化规约用以描述软件需求。后者根据数学方法例如定理证明或模型检验（model checking）等手段，对已建立的形式化规约进行分析，确认其是否满足期望的性质，最大程度地发现需求模型中不一致和二义性等错误，从而保证软件开发能建立在一个正确的需求模型之上。

尽管形式化方法对保证软件可信性展示了令人鼓舞的前景，然而，在当前的嵌入式控制软件开发领域，形式化方法的实际应用仍面临许多现实困难，大量深层次的问题有待解决。从软件需求分析角度看，存在如下主要问题：

* 缺乏合适的形式化建模语言

以目经典形式化规约语言如Z与B等建立的软件需求模型采用基于一阶谓词逻辑和集合论的数学符号来描述软件需求。由于嵌入式控制软件日益复杂，软件与外部环境交互频繁，参与开发的人员背景多元，对于软件开发者来说，单纯以数学符号书写、以文本形式展现的形式化规约对于普通软件工程人员而言难度过大，特别是缺乏可读性，直接影响不同背景工程师直接沟通，导致需求模型往往缺失很多重要内容。而UML和AADL等图形化符号又因缺乏严格的形式化语义难以胜任精确化需求描述。

* 缺乏形式化规约构建的工程方法

经典形式化建模方法关注的是对抽象的形式化规约验证,从而获得满足用户期望性质的形式规约，而未提供相应的工程手段引导用户如何从原始的、不完备的、非形式化的需求描述出发，逐步建立形式化规约。同时，验证涉及的数学证明过程过于复杂，机器自动证明的方法和工具能力有限，特别是对于一个初始的含有较多错误的规约，直接验证它更困难。模型检验工具则因“状态爆炸”等现实问题而难以取得理想的效果。

* 形式化规约的需求确认方法尚有局限性

需求确认（Requirements Validation）通常侧重于检查需求描述是否充分且准确地反映了用户对软件功能的期望。当前在工业界常用的需求确认方法主要是需求文档的审查(review)和仿真(simulation)，学术界提出的其他方法包括规约的测试(testing)和动画(animation)等，但是这些方法缺乏严谨性和系统性，特别是和形式化理论缺乏有机融合，使得需求确认的效果往往不如人意。

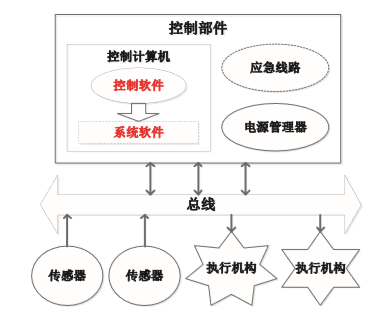
综上所述，当前嵌入式控制软件的需求建模面临的挑战可以表述为：缺少一种以形式化理论为科学依据的工程化方法，引导需求分析人员从原始需求出发逐步完成形式化规约的构建，且能有效地确认规约充分而准确地刻画了人们对软件功能的期望。

因此，我们需要一种以形式化方法为理论基础，面向工业界工程人员的系统化建模工程方法，引导工程人员从原始的以自然语言描述的需求出发，适应软件需求变更等常见情况，逐步构建精确描述系统功能的形式化需求规约，并提供相应技术实施需求确认以保证需求规约充分而准确地刻画了期望的功能。

## 1.2 航空发动机控制软件的特征分析

航空发动机控制软件是实时嵌入式软件，运行于电子控制器平台（EEC）中，实现发动机的运行控制，主要功能是按照飞机的指令实现发动机的启动、停车、推力控制、限制保护、作动部件控制、故障诊断及处理等。

图1.2.1展示了发动机控制系统的总体组成结构, 包括传感器, 执行机构, 总线, 控制部件等。传感器用于从环境测量必要的物理量, 例如, 燃油温度传感器, 排气温度传感器等；执行机构用于控制被控对象的状态, 例如,压气机，电磁阀等；总线用于连接控制部件，传感器和执行机构；控制部件用于根据传感器采集的物理量以及系统内部状态、执行控制算法和向执行机构下达控制指令；控制部件的核心是运行控制软件的控制计算机。本文所关注的重点是控制计算机中的控制软件，即航空发动机控制软件。



**图1.2.1 嵌入式周期控制系统总体结构**

航空发动机控制软件具有如下特征：

1. 控制软件输入为各传感器变量

2. 控制软件的输出为计算之后的信号量。

3. 控制软件需要周期性的进行工作，每个周期均需要通过控制规律进行相应的计算。

4. 控制软件的核心是控制规律，控制软件在不同的工作阶段有不同的控制规律。

5. 控制软件在符合一定条件的时候会发生控制规律的变化，在新的状态下会持续一段稳定的时间。

综上所述，本文将航空发动机控制软件视为一个以时间周期驱动的具有多个模式的控制系统。在每个模式内，根据设定的时间周期，完成模式内具体的运算任务和控制行为，并根据给定的条件，完成可能的模式切换或继续处于当前模式。

## 1.3 传统建模方式的问题

我们对航空发动机控制软件进行形式化建模。在正式开展建模活动之前，我们对已有的按照传统方式撰写的发动机控制软件需求文档进行了调研分析，总结了其中存在的缺陷：

1. 变量使用前未定义

原有的需求描述文档中，出现了大量未严格定义的变量。这些变量直接被应用于软件功能的描述，而这些变量在被使用前，工程人员未能准确地定义变量的类型或者初始值

1. 自然语言描述的二义性

原有的需求文档主要采用自然语言撰写。自然语言描述软件功能必然带来大量的模糊性与二义性，因此软件功能无法被精确地描述。

1. 整体控制逻辑不清晰

原有文档中，由于软件各具体功能条目未能精确描述，使得工程人员难以根据文档对系统的整体架构和控制逻辑有明晰的认识。由此，工程人员亦无法从需求中抽取出完整的控制流和数据流等逻辑特征进行深入分析，使得需求中存在的逻辑错误或功能缺陷无法及时被探测发现。

1. 需求不具有完备性

需求描述存在一定的不充分不完备现象，例如对功能输入范围的划分或变量取值空间的定义等。同时，通过与工程人员的交流沟通，我们发现仍有诸多需求和约束条件，未被需求文档撰写者在文档中刻画。

## 1.4 国内外研究现状

由于嵌入式控制软件通常部署于安全攸关系统或设备内，因此其功能正确性和性能可靠性及安全性长期以来受到了学术界和工业界的高度重视。传统的软件工程方法难以从深层次解决嵌入式控制软件当前所面临的挑战，而形式化方法因其完善的理论基础，被视为解决嵌入式控制软件质量问题最有前景的方法。

根据《NASA形式化方法指南》所总结的, “形式化方法(Formal Method)是一组基于数学模型和形式逻辑的技术与工具, 这些数学模型和形式逻辑通常用于确定和验证计算机系统和软件的需求, 设计与实现”。形式化建模主要通过以特定的数学语言对软件功能或行为进行抽象而精确地描述来实施，即对软件的需求进行建模。初始的模型可通过精化（refinement）转换为更为详细的模型，精化的过程通常通过运用定理证明等手段实现。而为了确保不同抽象层次模型的一致性，则可通过定理证明或模型检查等手段完成。理论上，用形式化方法开发软件，可在建立形式化需求模型后严格地通过精化和验证，得到最终软件代码。

经典的形式化语言和相应的建模方法有VDM，Z，Object-Z，B方法和Event-B等,均在软件工程领域确立了较大的影响力且有较成功工业界应用。巴黎城市地铁14号线全自动无人驾驶服务系统开发过程采用了B方法。精化过程完全使用形式化证明；证明过程中大量的错误被及时发现并纠正。

尽管上述形式化规约语言已通过各种应用展现了巨大潜力，然而在工业界的规模化应用仍面临较大困难。对于大型的嵌入式控制软件的形式化规约的构建，采用纯基于一阶谓词逻辑和集合论的形式化语言来描述软件需求对于软件开发者来说过于复杂，不利于开发者与用户交流。为了弥补这一缺陷，图形化建模方法也被应用于嵌入式控制软件的建模。上世纪90年代起，为了适应嵌入式控制系统等混合系统的建模，研究者提出采用混合自动机模型建模。混合自动机是状态机的模拟，主要从状态转换层面刻画软件行为，而对计算过程和软件体系结构的描述支持较少。类似地，状态图(State Chart)被广泛应用于软件的建模与分析。状态流(Stateflow)是商业软件Matlab/Simulink中提供的一种建模语言。状态流在状态图的基础上融合了基于控制流的计算过程, 可用于建模各种控制系统。SCADE工具是航空和轨道交通领域较受欢迎的形式化建模工具，其核心方法就是采用状态图描述需求，并通过形式化验证检查模型的一致性。但是SCADE本身并没有对建模的方法有明确的引导和约束。

SPARDL形式化建模语言对经典的状态图进行了拓展，提出了模式图（mode diagrams）用于描述嵌入式周期控制系统。模式图克服了状态图在时间周期方面描述能力的弱点，并且提供了模块化的描述结构用以支持控制流的建模。SOFL形式化建模语言提供了一种称为条件数据流图(Condition Data Flow Diagrams, CDFDs)的图形化符号，通过描述软件各模块间数据流的传递关系刻画系统行为和体系结构。然而CDFD缺少时间描述机制而无法对嵌入式控制软件进行形式化建模。

综上所述，我们不难发现，所谓“形式化建模方法”，在实际应用中，其实更多情况下是以建模语言的形式出现的。无论是经典的Z、VDM还是后续的Event-B或是针对特定领域的例如SPARDL语言，主要是在语言上和描述方式上存在较大差异，而背后的方法仍然是以定理证明为基础的精化过程。然而，一方面，定理证明为基础的精化在工业界应用的技术难度和成本直接制约其推广，当前的软件工具难以支持大规模工业界软件的精化，同时，工程人员也难以达到灵活运用精化方法所需的理论基础。另一方面，从软件工程方法层次看，这样的建模更接近于一种数学方法而非工程方法，即无法通过系统化的、与软件工程过程直接融合的流程，为工业界软件开发者提供清晰的建模过程引导。从工业界应用的需求看，当前的形式化方法缺少一种工程化的手段引导软件开发者从原始需求出发，通过一定的步骤构建形式化规约，是制约形式化建模在工业界广泛应用的重要因素。

Shaoying Liu教授提出了以SOFL（Structured Object-Oriented Formal Language）为形式化规约语言的形式化工程方法。SOFL方法提供了一种工程化的方法，即通过演化方式，引导开发者通过建立非形式化、半形式化和形式化规约完成需求模型与软件设计的精确描述。该方法已在传统软件和基于Web服务的软件建模过程中取得了良好的效果。然而，SOFL语言本身尚不支持嵌入式控制软件这类实时系统的建模，其建模过程也难以直接应用于嵌入式控制软件。

综上所述，对于嵌入式控制软件，无论是工业界还是学术界，形式化方法都尚未和软件工程过程实现有机融合，尚缺乏成熟的形式化规约构建的工程方法，将形式化规约的构建以系统化工程化的过程提供给工业界开发者。

## 1.5 建模方法概述

形式化方法是建立在数学基础上的针对数字化系统进行规格说明撰写、软件开发、软件验证的技术。形式化方法的数学基础包括形式逻辑，离散数学和机器可识别语言。

形式化模型是一种无二义性的用数学语法和语义刻画的模型，是一种对软件诸多方面的抽象表达形式，用于后续的分析、仿真和代码生成（在有些条件下，直接从模型生成代码可能无法实现）。应用形式化方法的基础是对即将开发的软件建立一个形式化模型。

计算机系统和软件的需求和设计往往是用“通俗易懂”的语言撰写的，这些语言一般来说，是自然语言和一些约定俗成的符号，图标以及伪代码。由于采用自然语言，某个对撰写者来说，清晰无误的表述，全可能被其他人理解成其他意思。导致不同的开发者对相同的需求和设计产生模棱两可的理解，即所谓的二义性。

为了避免这种由于用语不规范导致的模棱两可带来的二义性，常要求以标准的形式来描述需求和设计。这种标准形式一般以数学语言为基础。这种标准化的描述规范就构成了形式化方法的基础。

形式化方法的目标不仅局限于准确地描述系统的需求和设计，还包括验证需求的一致性，设计是否满足需求等。验证是一个步进式的过程，在开发周期的各个阶段都要对该阶段的产品进行验证。检查该阶段的新产品是否与之前的产品保持一致性，是否仍然满足需求。例如，在概要设计已被确认遵循了各个方面的需求的情况下，详细设计阶段要验证其是否沿袭了概要设计。

本次形式化建模的工作流如图1.3.1所示



**图1.3.1 建模工作流**

针对航空发动机控制软件的特征，我们设计了建模语言AEDL（Aero-Engine Description Language），及其图形化表示模式图（ModeDiagram）。众所周知，自然语言的处理和理解是困难的，为了方便商发工作人员的使用，尽量不改变传统的需求描述方式，我们提供了规范化的需求模板，帮助将传统的自然语言撰写的需求文档转化为规范化的需求文档，然后设计工具从规范化的需求文档中抽取必要的信息，构建AEDL软件模型。

为了分析和验证AEDL软件模型，本文定义了AEDL的操作语义。基于语法和操作语义，可以设计AEDL的支撑工具，使之能够读取并解析AEDL所描述的物理模型，从而对其进行模型验证和性质验证。

# 2. AEDL建模语言

## 2.1 语言概述

为了给商用航空发动机控制软件提供一套清晰无二义性的建模方法，我们提出了形式化建模语言AEDL(Aero-Engine Description Language)，用于描述发动机控制系统的结构与功能模型；同时，也设计了其图形化表示模式图（ModeDiagram）。

该语言支持发动机控制系统的多模式，模式嵌套，模式迁移，周期驱动等要素。其主要特征是:

1。基于模式: AEDL的基本建模元素是“模式”(Mode)，每个模式表明系统处于特定的状态， 使用特定的控制算法。

2。 层次结构: AEDL的建模粒度分为三个层次，顶层展示了各个模式以及模式迁移关系，包含关系。中间层描述每个模式内部的计算流程。底层是具体的控制算法。层次结构既反应了发动机控制系统本身的特征，也有利于软件工程师理解开发需求，分解开发任务。

3。 面向计算:由于发动机控制系统需要处理许多数据以便确定当前发动机所提供的推力大小，并通过控制油门阀角度来进行调整，控制软件会涉及到许多复杂的计算。这些计算被封装为不同的控制算法，位于模型的第三个层次。该层次对控制算法的封装，既有利于开发任务的分解，也有利于软件产品的重用。

4。 周期驱动: 发动机控制系统是一种不会终止的反应式系统(reactive system)，这种系统按照给定的时间反复执行计算任务。AEDL模型中的模式阐明了控制软件的周期。并不要求各个模式的周期相同，因此，控制软件处于不同的模式时，可以有不同的周期。

AEDL描述的控制软件的模型是一个三元组

Model::= (Dictionary， ModeDiagram， Modules)

其中

• Dictionary: 数据字典，定义了系统中使用的全局变量。

• ModeDiagram: 模式图，阐明了组成该软件模型的各个模式，这些模式的流程，以及模式之间的迁移关系，组合关系。

• Modules: 模块， 定义了模式过程中会调用的各个控制模块。

数据字典描述了该控制系统中出现的变量，模式和模块中使用到的变量均定义在数据字典中。作为一个控制系统，变量是有一定的物理含义的。因此，数据字典的具体定义待以后详细给出。

ModelDiagram（模式图）拥有关系集合，迁移卫式以及模式。

其中：

•关系集合用于说明模式-子模式之间的关系。

•迁移卫式指的是模式之间切换需要满足的条件，在模式图中只有当系统符合一定的前置条件时才可能发生模式的切换。如表1所示，即为针对发动机故障诊断与处理的相关需求进行的迁移卫士举例。

•模式包含模式名称，系统周期，控制流。其中：

•系统周期指的是当系统处于该模式下时，系统的工作周期是多少。

•控制流用于描述模式内部的计算任务。模式的计算流程和封装了控制算法的模块(Module)的计算流程均由控制流描述，包括三种控制结构: 顺序组合，分支，循环，三种基本语句: 赋值，过程调用，空语句。其中：

•模块（Module）指的是完成某个特定计算功能的实体。其由名称，输入，输出，模式体构成。其中：

•输入为具体控制功能的输入

•输出为具体控制功能的输出

•模式体是一个控制流，利用输入输出进行特定的计算，一般而言，物理公式的计算应当放入该控制流中进行计算。

图2.1.1给出了使用AEDL模型描述的发动机控制系统的图形化表示（模式图）。该系统的第一层，模式迁移层，描述了7个模式，其中运行模式，故障模式和维护模式为顶层模式，启动模式和慢车及慢车以上模式是运行模式的子模式，慢车模式和慢车上模式是慢车及慢车以上模式的子模式。初始状态时，上电0.6s之后系统稳定则默认进入运行模式。各个模式之间通过满足相应的迁移卫式来进行模式转换。该系统的第二层描述了系统各个模式的计算流程，这些计算流程会调用具体的控制算法。第三层给出了第二个层次所调用的控制算法的细节。

**图2.1.1 模式图举例**

图2.1.1描述了模型的层次结构，较为详细的介绍和表达了其内部实现的细节。有时也会隐去具体的控制流图和控制算法，只用第一层的模式来抽象表达整个模型，如下图2.1.2所示。

**图2.1.2 模式迁移关系举例**

## 2.2 AEDL语法

### 2.2.1 AEDL模型

AEDL描述的嵌入式控制软件的模型是一个三元组，描述如下：



其中

- Dictionary: 数据字典,定义了系统中使用的全局变量。

- ModeDiagram: 模式图,阐明了组成该软件模型的各个模式,这些模式的流程,以及模式之间的迁移关系,组合关系。

- Modules: 模块, 定义了模式过程中会调用的各个控制模块。

通过各个模式和各个控制模块的控制流中出现的调用语句,可以获得模块之间的调用关系，对于模块，有,当且仅当m1的控制流中含有语句call m2。

SPARDL的模块之间也不得存在直接或间接的递归调用。设CallP是关系Call的闭包,我们要求：，其中：



### 2.2.2 模式图

模式图(ModeDiagram)是一个定义在给定的变量集合V (包含标量和矩阵两类变量),迁移卫式集合G(V)和控制流集合CFG(V)上的三元组,它的语法结构如下：

****

其中：

1. **：**包括子模式在内的所有模式构成的集合,其中的元素用字母m表示。一个模式m是一个四元组(name, inital, period, cfg)。

– : 模式名称。

–  : 说明模式m是不是初始模式。

– : 系统周期, 单位是毫秒.

– : 用于描述模式内部的计算任务。如果该模式含有子模式，则不含控制流，本文使用记号⊥作为占位符，表示该模式有子模式，无控制流。

1. : 该二元关系用于说明模式-子模式的关系。
2. : 描述了模式迁移关系。

一个合法的模式图应当满足一定的语法约束，需要满足的约束包括：

- Contains关系必须是无环的：



- 一个模式不能是多个模式的子模式：



- 一个复合模式的周期长度应当是它的子模式的周期长度的整数倍：



- 对一个复合模式来说,它的各个子模式中,有且只有一个初始模式：





- 在所有的顶级模式中, 有且只有一个初始模式：



- 如果一个模式没有子模式,那么它必须含有控制流：



给定一个模式图MD,我们定义下列函数:

- ，表示在模式图MD中获取从模式m到m′的迁移。其中，

- 返回一组模式,这些模式都在中，且它们的周期长度与给定的模式m和周期计数k一致。

- ，其中：



- ，表示获取给定模式的初始子模式,要求给定的模式不是叶子模式。其中。

- ，返回模式序列中的所有模式的迁移。

。它的输入是迁移集合T和迁移t,返回迁移集合T’, T’由T中所有优先级高于t的迁移组成。

- ，这是迁移t的迁移卫式与所有与它“同源”,且优先级比它更高的迁移的卫式取反的合取。这里的“同源”是指源模式相同,或t的源模式是该模式的子模式。其中：



### 2.2.3 迁移卫式

迁移卫式是用于模式迁移的条件。给定状态表达式集合SExpr, 迁移卫式由下列BNF定义：











其中：

1. 和是布尔常量。
2. 是作用在状态表达式上的谓词。
3. 是布尔表达式,是一个布尔项,或布尔项和布尔表达式的逻辑组合。
4. 是时间谓词。

给定一个迁移，我们给出下列操作：

，获取该迁移的源模式。

，获取该迁移的优先级。

，获取该迁移的条件卫式。

，获取该迁移的目标模式。

### 2.2.4 控制流

给定变量集合V,则控制流由下列BNF定义。这些控制流构成的集合记作CFG(V)。











1.  控制流分为两个部分, decalres:局部变量定义,stmts:语句。要求局部变量定义中的各项(declare)的名字(name)不能重复。设局部变量定义构成的变量集合是VL, 则语句中出现的变量在集合V ∪ VL中。
2. 语句分为两种，primitive statements：基本语句，compound statements：复合语句。

i：基本语句包括赋值语句模块调用语句, 空语句三种。

①赋值语句又可以细分为以下三类: 第一类左值是标量变量, 右值是状态表达式。第二类左值是矩阵变量或左值显式矩阵, 右值是矩阵表达式。第三类左值是矩阵元素, 右值是状态表达式。

②复合语句分为两种, 分别是顺序语句, 分支语句。

1. 基本语句,分支语句的标签，为正整数，同一控制流内不重复。
2. ,分别表示标量变量和矩阵变量。

### 2.2.5 模块

模块是定义在给定的变量集合V(包含标量和矩阵两类变量)和控制流集合CFG(V)上的模块module是一个四元组, 语法结构如下:



其中：

- name: 模块的名称, 是模块的唯一标识。

- :分别表示该模块的输入变量集合与输出变量集合。

- :表示模式体,是一个控制流。

### 2.2.6 模式

我们将一个没有子模式的模式称为基本模式(PrimitiveModes),将有子模式的 模式称为复合模式(CompoundModes),将不是任何模式的子模式的模式称为顶级模式(TopModes)，它们的语法如下：







给定一个模式,我们定义下列操作：

，获取该模式的周期长度。

，检查该模式是不是初始模式。

，获取该模式的控制流,如果该模式不是叶子模式,则返回占位符⊥。

### 2.2.7 数据字典



1. name 表示变量名, 是变量的唯一标识。
2. type 代表变量类型, 包括布尔型(Bool), 整型(Int)和浮点型(Float)。
3. unit 表示变量的单位, 采用标准公制单位, 由一个五元组表示（语义1）。
4. usage 包括只读(R), 只写(W), 可读可写(RW)三种取值. 只读变量用于表示来自传感器的输入, 只写变量用于表示针对执行机构的输出, 可读可写的变量用于维护系统自身的状态。
5. dimension 表示变量的维度. 标量变量的取值是0, 矩阵变量的维度是一个二元组(m, n), 表示这个变量是一个m×n的矩阵. 特别地, 1×n表示长度为n的行向量, n×1表示长度为n的列向量。

defaultVal 表示变量的默认值。

### 2.2.8 单位



1），表示单位的系数, 用于处理公里, 英里等非基本量纲。

2）表示该单位在5个基本量纲上的取值。

–  长度单位, 米(L)

–  质量单位, 千克(M)

–  时间单位, 秒(T)

–  电流单位, 安培(A)

–  温度单位, 开尔文(K)

### 2.2.9 其它

我们以产生式的形式给出AEDL模型的语法定义,这些产生式符合Antlr语法分析的格式要求。以下给出部分产生式示例：

**模式迁移：**

mode\_transition : TRANSITIONS^LEFTBRACE transition\_element \*

RIGHTBRACE ;

transition\_element : INT^ transition\_body ;

transition\_body :LEFTBRACE transtion\_priority transtion\_cond

transtion\_action? transtion\_target transtion\_

modeChangeAction? RIGHTBRACE;

transtion\_priority : PRIORITY^ LEFTBRACE INT RIGHTBRACE ;

transtion\_cond : COND^ LEFTBRACE expr\_bool RIGHTBRACE ;

transtion\_action : ACTION^ comp\_stmt ;

transtion\_target : TARGET^ comp\_stmt ;

transtion\_modeChangeAction : MODECHANGEACTION^ comp\_stmt;

**模块:**

module\_decls : MODULE^ LEFTBRACE module\_element\* RIGHTBRACE ;

module\_element : LEFTBRACE^ module\_oldName module\_name

module\_input module\_output module\_

locvar? module\_formula RIGHTBRACE ;

module\_oldName : MOLDNAME^ LEFTBRACE STRING RIGHTBRACE ;

module\_name : MName^ LEFTBRACE STRING RIGHTBRACE ;

module\_input : INPUT^ LEFTBRACE ID(COMMA ID)\* RIGHTBRACE ;

module\_output : OUTPUT^ LEFTBRACE ID(COMMA ID)\* RIGHTBRACE ;

module\_locvar : LOCALVARIABLE^ LEFTBRACE ID(COMMA ID)

\* RIGHTBRACE ;

module\_formula : FORMULA^ comp\_stmt ;

**模式图:**

compilationUnit : data\_decls module\_decls mode\_decls;

mode\_decls : MODE^ LEFTBRACE mode\_element+ RIGHTBRACE ;

mode\_element : ELEMENT^ LEFTBRACE mode\_name mode\_guard ((mode\_init

mode\_procedure) | mode\_element+) mode\_transition

RIGHTBRACE ;

mode\_name : NAME^ LEFTBRACE STRING RIGHTBRACE ;

mode\_guard : GUARD^ LEFTBRACE expr\_bool RIGHTBRACE ;

mode\_init : INIT^ comp\_stmt;

mode\_procedure : PROCEDURE^ LEFTBRACE procedure\_element\*

RIGHTBRACE ;

procedure\_element : MODULE^ procedure\_body ;

procedure\_body : procedure\_period comp\_stmt ;

procedure\_period : PERIOD^ LEFTBRACE (FLOAT|INT) TU\_SEC RIGHTBRACE ;

## 2.3 操作语义

### 2.3.1 模式层操作语义

#### 2.3.1.1 AEDL语义空间

在模式层,我们只关心由数据字典定义的全局变量,故而State的变量集合是 这些全局变量以及一个特殊的变量timestamp构成的集合。timestamp代表时间戳。AEDL的语义空间定义如下：

****

1. MD表示整个模式图, 表示系统当前所处的模式。
2. 用于表示系统所处的阶段,Begin表示周期初始, Execute表示准备执行周期任务, End表示周期末尾。
3. , 其中是一个程序计数器,用于标识模式体的控制流的执行情况. N是控制流内部的位置标记. Start和Exit分别表示控制流的开始位置和结束位置. 如果当前的模式没有控制流, 那么使用符号⊥作为占位符。
4. 第四元k是周期计数器. 每个周期加1. 如果系统发生模式切换, 那么k将被置为1。
5. Σ是变量状态构成的序列, 记录了历史状态. 序列中的每个元素Σ都是一个变量名到变量取值的映射Σ。

#### 2.3.1.2 状态表达式在语义空间上的解释

给定标量变量(Scalar Variables)矩阵变量(Matrix Variables)构成的集合V,则状态表达式由下列BNF定义：







该表达式在语义空间上的解释如表2.1所示。

|  |  |
| --- | --- |
| （常量） | 其中c是常量 |
| （变量-1） | 其中x是变量 |
| （变量-2） | 其中m是矩阵变量 |
| （变量-3） | 其中m是行向量变量 |
| （变量-4） | 其中m是列向量变量 |
| （矩阵） | 其中  , |
| (函数） | 其中f是函数 |

**表2.3.1 状态表达式在语义空间上的解释**

#### 2.3.1.3 卫式在语义空间上的解释

给状态表达式集合SExpr,则用于模式迁移的条件,迁移卫式,由下列BNF定义：











该卫式在语义空间上的解释如下表2.3.1所示。

**表2.3.1 卫式在语义空间上的解释**

|  |
| --- |
|  |
|  |
| 或 |
| 或 |
|  |
| 其中是长度为n的状态序列， |

#### 2.3.1.4 子模式

子模式描述了在周期开始时, 如果当前最内层的模式含有子模式, 那么SPARDL模型立即进入子模式中的初始模式。如果该初始子模式有控制流, 那么 将程序计数器pc置为Start,表示在本周期将从头开始执行该子模式的控制流。如果该初始子模式也含有子模式,则将将程序计数器pc置为⊥。在这种情况下, 将再次应用该规则,进入子模式的初始子模式。语义表述如下：

****

其中，****，****

#### 2.3.1.5 采样

如果系统处于周期初始,最内层的模式有控制流,那么系统通过传感器进行采样,用传感器获取的值更新系统中的只读变量。然后开始执行模式的控制流。语义表述如下：

****

其中，**。**是数据字典中定义的只读变量。****表示从传感器获取的值。

#### 2.3.1.6 执行

描述了以程序计数器pc为程序起始位置,以当前模式的周期长度为执行时间,执行当前模式的控制流。函数execute获取控制流的执行结果,程序计数器从位置pc变为,新的程序状态是。语义表述如下：

****

其中**，**

#### 2.3.1.7 继续

语义表述如下：

****

#### 2.3.1.8 重复

描述了如果所有模式迁移的条件都不满足, 那么系统将在下个周期继续处于当前模式。语义表述如下：

****

#### 2.3.1.9 切换

描述了如果某个模式图迁移的条件满足,且它的优先级高于其他任何条件的迁移, 那么该迁移将会发生。SPARDL模型将在下个周期转入目标模式。语义表述如下：

****

****

****

其中，****，****

### 2.3.2 控制流层操作语义

#### 2.3.2.1 控制流的语义空间

控制流描述了模式和模块的计算过程。控制流分为两个部分,局部变量定义(declares)和语句(stmts)。前者构成了控制流的局部状态,后者说明了控制流的计算过程。控制流的语义涉及局部变量的语义，SPARDL模型分配给该控制流的执行时间和待执行的语句，因此控制流的语义空间定义如下:



其中和s都是变量赋值,分别代表全局变量状态和控制流的局部变量状态。 和s共同组成控制流的变量状态,因此定义这两个变量状态的和操作如下:



#### 2.3.2.2 赋值

赋值语句分为两类,一是对变量的赋值(规则(赋值-1)),二是对(矩阵)变量的成员的赋值(规则(2-赋值-2))。前者直接修改变量的状态, 后者要求左值的下标没有超过矩阵变量的维度。两种赋值语义表示如下：

赋值-1：，其中

赋值-2：，其中

#### 2.3.2.3 调用

规则(调用)描述了模块调用语句的语义。被调用的模块应当满足原子性。即如果无法在给定的时间t之内执行完成,则不能执行该模块的调用,该模块对全局变量状态的改变不起作用。语义表述如下：

****

#### 2.3.2.4 分支语句

分支语句分为正确分支和错误分支，其语义描述分别如下：

True:****

False:****

#### 2.3.2.5 顺序语句

顺序语句的描述如下：

顺序-：

顺序：

# 3. AEDL建模方案

## 3.1 需求模板撰写规范

撰写形式化需求应正确使用建模语言提供的各种关键字字段，并为每个字段撰写相应的内容。本形式化建模语言主要通过模式和其内部结构表示软件的功能。

### 3.1.1模式（Mode）

模式是本建模语言第一个功能描述层次。一个模式由以下几个部分组成：名称（Name），进入条件（Guard），初始化（Initial），流程（Procedure）以及转换条件（Transition）。每个模式要严格按照以下格式书写：

**模式名称：** 字符串

**进入条件**： 表达式

**初始化**： 语句块

**调用流程**：

**周期任务**：时间

语句块

….

**周期任务**：时间

语句块

**模式转换**：

1 **优先级**: 整数

**满足**：表达式

**动作**：语句块(可选)

**目标**：语句块

2

…….

图3.1.1 模式结构

上图加粗体的都表示是关健字。其中，字符串是一系列字母的组合。进入条件表示当系统当前状态满足这个条件时，则系统进入该模块，允许其执行模块内任务。时间是整数或浮点数和单位的组合，单位可以是秒（s），毫秒(ms)。语句块可以是一条语句或是多条语句的组合。

每条语句前可加上标签（自定），每条语句后以分号(;)结束。语句有赋值，选择，调用三大类。

选择语句的格式是：if (expr) then 语句块 else。

语句块调用语句的格式是：call 语句块。

对于一般的赋值语句，变量为一维的，格式为Var = expr

对于特殊的赋值语句，我们有如下约定：

(1)对于向量的赋值语句，形如 ,直接忽略转置，将值读出分别赋值即可。

(2)对于矩阵的赋值，采用如下形式：X(j,i)=0, j in (1,…,8), I in (1,2,3)

表达式的书写规范遵循C语言的表达式规范。表达式可以是数据字典中出现的变量符号，也可以是常数，以及他们的组合。根据组合的算子我们可以把表达式分为关系表达式，算术表达式，逻辑表达式，位运算表达式，以及我们引进的C语言中没有的时间表达式。它们各自的算子在表3.1.1中给出：

**表3.1.1 表达式书写规范**

|  |  |
| --- | --- |
| 表达式类型 | 表达式的组成算子 |
| 关系表达式 | >,< ,==,>=,<=,!= |
| 逻辑表达式 | && , || , ~ |
| 算术表达式 | + , - ,\* , / , % |
| 位运算表达式 | >> ,<< ,>= , <= , & ,| ,^ |
| 时间表达式 | Duration(Const,Unit,expr), after(Const, Unit, expr) |

对于最后一个时间表达式，我们引进了duration和after两个算子，分别对应需求中可能出现的关于时间的两种情况。比如，需求文档中会出现这样一种表述：“连续100个周期满足a>b”，这类表述我们在我们的规范中就用duration来表述，这句话在应用我们的规范之后应该写为“duration(100,周期,a>b)”，这个算子的三个参数分别表达持续的时间，时间单位以及满足的条件。而对于需求文档中的另外一种可能的关于时间的表述：“满足Gmode=2之后240s”，这种表述可以用我们定义的after算子来描述：“”after(240,s,Gmode=2)”.在这两个算子中，单位可以是秒(s),毫秒(ms), 周期等一系列的时间单位。

调用过程中，周期任务可以多次出现；同样的模式转换中也是。

### 3.1.2 流程（Procedure）

流程是每个模式内承载具体计算功能的描述部分，描述了系统中的各个模式的计算过程，这些计算流程会调用具体的控制算法。流程是建模语言的第二层次功能描述，其本质等价于流程图。

### 3.1.3 任务（Task）

任务是建模语言的第三层次功能描述，刻画了每个流程（procedure）中单个计算任务是如何完成的。其基本结构是一组输入和输出变量之间的关系，本质上等价于数据流图中的一个活动（process）。

每个任务均按以下格式书写，参见图3.1.2：

**任务名称**：字符串

**编号**：字符串

**功能**：字符串

**输入**：字符+

**输出**：字符+

**公式**：语句块

**图3.1.2 任务结构**

1) 字符串是字母的组合。但是注意编号的字符串必须是惟一的，即不同的功能模块它的编号是不一样的。

2) 输入和输出是一系列数据字典中数据符号的集合，不同符号之间统一用逗号隔开。

3) 为了与前面的规范保持一致，我们把公式也看成是一个语句块，当然这边的语句块里只有赋值语句这一类。

### 3.1.3 数据字典（Data Dictionary）

数据字典用于描述每个模式用到的各变量的定义。在需求文档中，数据字典是一个或多个表格，每个表中的项有以下几个：符号、物理意义、位数、数值表示（类型）、量纲、维数、初值和说明。

具体如下表3.1.1所示：

**表3.1.1 数据字典规范**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 符号 | 物理意义 | 位数 | 数值表示 | 量纲 | 维数 | 初值 | 说明 |
| …… | ……. | …….. | ……… | …….. | …….. | …….. | …….. |

1) 符号表示该数据量的符号表示，是一个word公式编辑的符号。

2) 物理意义可以有两种书写形式。一种是一个二元组序列，用如下形式书写：(int , stirng)…其中，int表示变量的取值，string是对这个取值的解释，是一个字符串。另外物理意义也可以只是一个字符串，可以为空。

3) 位数是指该变量在计算机中的存储位数，是一个整数

4) 数值表示是指该变量的数据类型，可以是整数，浮点

5) 量纲是指它的物理量纲，一个字符串，可以为空

6) 维数是一个二元组(a,b）,当a和b都为0时标识该变量是一般变量，当a=1,b>1时表示它是长度为b的行向量，当a>1,b=1时表示它是长度为a的列向量。当a>1,b>1时表示它是a\*b的矩阵。

7) 初值给出了该变量的初始值，可以是整数，浮点数，以及向量形式或是矩阵形式

8) 说明是对该变量的其他解释，是一个字符串，可以为空

发动机控制软件中当前使用简化版的数据字典。

具体如下表3.1.2所示：

**表3.1.2 简化版数据字典规范**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 变量名 | 变量符号 | 类型 | 类型解释 | 变量注释 |
| TBV4L回路故障 |  | Bool | 1：故障，0：无故障 |  |
|  |  |  |  |  |

### 3.1.4 需求模板提供

#### 3.1.4.1 需求模板语法

**模式名称：**字符串

**进入条件**：表达式

**初始化**： 语句块

**调用流程**：

**周期任务**： Int (s)

语句块

….

**周期任务**：时间

                            语句块

**模式转换**：

          Int **优先级**： Int

**功能：**字符串

**满足**：表达式

**动作**：语句块(可选)

       Int **优先级**： Int

**功能：**字符串

**满足**：表达式

**动作**：语句块(可选)

**任务名称**：字符串

**编号**：字符串

**功能**：字符串

**前置条件：**语句块

**输入**：字符

**输出**：字符

**公式**：语句块

#### 3.1.4.2 需求模板语法解释

通用部分：

加粗字体为关键字，书写文档时关键字不可更改。

字符串指普通字符串，直接输入文本即可。

表达式指判断条件，如 a==b这种，返回值应为一个bool值，结尾不加分号。

语句块指的是复合语句，复合语句应当加{}。即{a=2;b=3; }这种形式。

Int指的是在书写需求文档时，该位置需要一个int类型的变量填充。

语句块部分如果为空，需使用分号代表为空。即 {;}

编号命名不可重复。

每条语句前可加上标签（自定），每条语句后以分号(;)结束。语句有赋值，选择，调用三大类。选择语句的格式是：if (expr) 语句块 else。

语句块调用语句的格式是：call 语句块。

对于一般的赋值语句，变量为一维的，格式为Var = expr

对于特殊的赋值语句，我们有如下约定：

异常语句：throw +字符串。

模式部分：

调用流程部分可包含多个周期任务。在每条语句中，周期任务为关键字，后面跟int变量+”(s)”。即周期任务：2(s){Call (\_4.2.4.5.1);Call (\_4.2.4.6.1);}这种形式。

模式转换中，优先级前方仍有一个Int变量，不可省略。

Call语句为特定的调用指定模块部分。其格式为 Call关键字+(模块名称),即 Call (4.2.4.6.1)这种形式。

模块部分：

输入输出中的各个变量之间使用逗号分隔，即 a,b,c 这种形式。

数据字典部分：

Type类型包含bool,int,float,char,string,enum类型。

需要有初值的变量应在数据字典中给出。

#### 3.1.4.3 需求模板

**数据字典：**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 变量名 | 变量符号 | 类型 | 类型解释 | 初始值 | 变量注释 |
| 字符串 | ID标识符 | Type | 字符串 |  | 字符串 |
|  |  |  |  |  |  |

模式名称:模式1

进入条件:条件1

初始化:{;}

调用过程:

周期任务: 1 (s){

Call();

}

周期任务: 2 (s){

Call();

}

模式转换:

1优先级: 100

功能：功能目标1

满足:条件11

动作：{;}

2优先级: 10

功能：功能目标2

满足:条件12

动作：{;}

模式名称:模式2

进入条件:条件2

初始化: {;}

调用过程:

周期任务:Period 1 (s){

Call();

}

周期任务: Period 2 (s){

Call();

}

模式转换:

1优先级: 100

功能：功能目标21

满足:条件21

动作：{;}

2优先级: 10

功能：功能目标22

满足:条件22

动作：{;}

模块名称: 模块1

编号: 模块编号1

功能: 功能描述1

前置条件：{;}

输入: 变量1,变量2,变量3

输出: 变量4,变量5,变量6

公式: {;}

模块名称: 模块2

编号: 模块编号2

功能: 功能描述2

前置条件：{;}

输入: 变量21,变量22,变量23

输出: 变量24,变量25,变量26

公式: {;}

数据字典:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 变量名 | 变量符号 | 类型 | 类型解释 | 初始值 | 变量注释 |
| TBV4L回路故障 |  | Bool | 1：故障，0：无故障 | 0 |  |
|  |  |  |  |  |  |

## 3.2 建模过程

1) 识别出软件可能处于的模式集合，确定每个模块名称。在形式化需求文档中，初步定义模式，撰写各个模式名称。

2) 建立系统的数据字典，将可能用到的物理变量的名称、含义、量纲等基本信息予以记录，数据字典格式见4.2章节相关内容。

3) 针对每个模式，逐步建立其内部结构，即完成初始化、转换条件等内部模块定义，以刻画每个模块对应的软件行为。模块的结构定义见4.2章节相关内容。

4) 软件工具将识别工程师撰写的需求文档，将其处理后，转换为最终的形式化模型。

### 3.2.1 模型示例

由于文档涉密，在例子中将变量名替换。以下为半形式化需求文档中所用到的模式的例子。

**模式名称：**模式1

**进入条件：**ES==ES\_MODE1

**初始化：**;

**调用流程：**

**周期任务**：5(ms)

{

Call(\_4.2.2.6.1.1);

}

**模式转换：**

**1优先级**: 1

**功能**：进入模式2

**满足**：A.SignalFlt==0 &&WheelLoadSignal==1

**动作**：{ ES= ES\_MODE2;}

# 4. AEDL工具设计

## 4.1 需求文档存储

### 4.1.1 存储结构

满足上述需求的文档，可以被转换成一个中间模型，其整体数据结构定义如下：

#### 4.1.1.1 高层模型

Model = {

**Mode**{ModeEle}

**Module**{ModuleEle}

**Data**{DataEle}

}

其中，Mode部分即对应上述需求文档中模式部分，Module对应上述需求文档中模块部分，Data部分对应上述文档中数据字典。

#### 4.1.1.2 模式

模式各个数据项中的元素所包含的内容如下：

ModeEle = element{

**Name**{}

**Guard**{}

**Init**{}

**Procedure**{}

**Transitions**{}

}

其中，Name为模块名称，Guard为进入条件，Init为初始化内容，Procedure为进入该模式之后所要执行的过程，Transiton为模式转换条件。

#### 4.1.1.3 模块

ModuleEle = element {

**MID**{}

**MName**{}

**Description**{}

**PreCondition**{}

**Input**{}

**Output**{}

**Formula**{}

}

其中，MID为模块ID，MName为模块名称，Description为模块描述，PreCondition为前置条件，Input为输入，Output为输出（用列表存储），Formula为公式部分。

#### 4.1.1.4 数据字典

DataEle = element {

**DName**{}

**NName**{}\*

**Type**{}\*

**Note**{}

**InitVal{}\***

}

其中，DName为汉语名字，NName为英语名字，Type为变量类型，Note为变量注释，InitVal是初始化值。其中NName、Type、InitVal是必填项。

### 4.1.2 中间模型设计

Mode的Datatable的详细设计如下：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 名称 | 进入条件 | 初始化 | 周期 | 迁移条件 | 其他 |
|  |  |  |  |  |  |

等价于以下数据结构：

Mode {

**Name** { String }

**Guard** { expr }

**Init** { Statements }

**Procedure** { (Period { Const unit} {Statements})+}

**Transitions** {

( **Priority** {int}

**Cond** { expr }

( **Action** { Statements}) (可选)

**Target** {Statements}

)

}

}

Module的Datatable的详细设计如下：

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 任务名称 | 编号 | 功能 | 前置条件 | 输入 | 输出 | 公式 | 其他 |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

等价于以下数据结构：

**Module** {

(**FName** { String }

**Input** { Var (, Var)\*}

**Output** { Var (, Var)\*}

**Formula** { Statements} )+

}

数据字典的Datatable的详细设计如下：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 名称 | 别名 | 物理意义 | 类型 | 维度 | 初始值 | 其他 |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

等价于以下数据结构：

**Data** {

(

**Name** { String }

**NName** { String }

**Meaning** {( (int , String) )\* }

**Type** { int |float |Vector | Matrix}

**Unit** { String}

**Dimension** { (int ,int)}

**InitValue** {int |float | Vector |Matrix}

**Note** {String }

)

}

### 4.1.3文档抽取

抽取需求文档的word文档之后将文档内容一式两份分别存储于内存中的datatable和StringBuilder中。其中Datatable供debug阶段使用，主要是为了方便开发人员查看。StringBuilder依据数据结构生成，作为antlr的input生成AST语法树。

利用openXmlSdkV2.5来实现Word文档的生成，读取，修改操作。需要根据相对应的模板进行操作或者将word文档整个读入内存中再进行处理。依据正则表达式读取正确按照关键字序列描述的mode，module及数据字典。

通过antlr的vistor对生成的ast树进行访问，按照定义的中间模型依次将需求文档的各个部分放置到中间模型中。

## 4.2 中间模型结构

根据生成的树结构建立能够存储树结构的中间模型。接下来，依据自顶向下的方法描述整个中间模型结构：

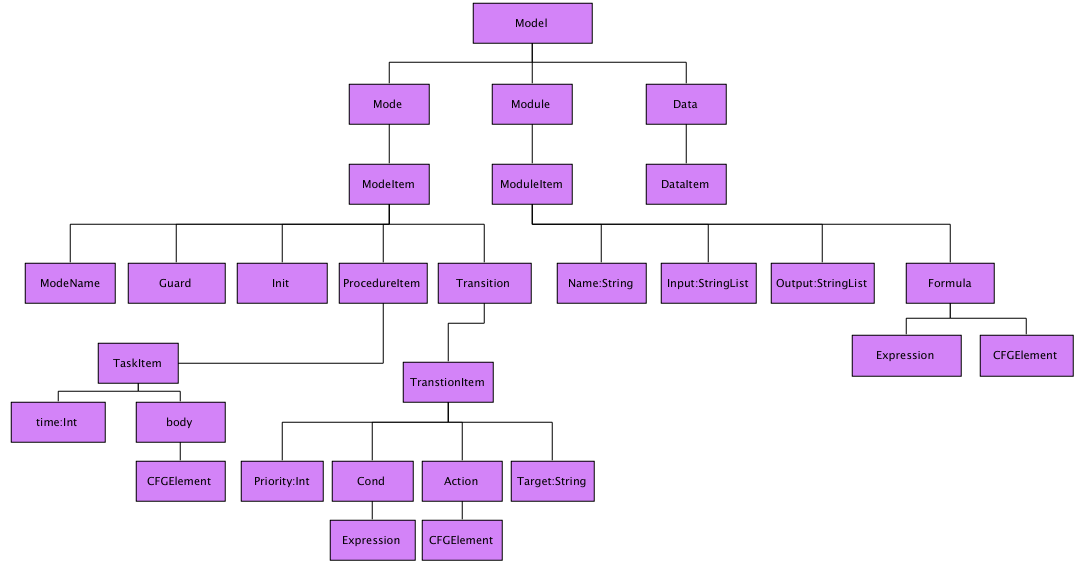


图4.2.1 中间模型结构图

## 4.3 撰写可分析形式化需求的Antlr语法

根据设计的形式化语言撰写antlr可识别的grammar，解析生成相对应的AST。核心语法如下所示：

compilationUnit

: data\_decls module\_decls mode\_decls

;

//Modes

mode\_decls

: MODE LEFTBRACE mode\_element+ RIGHTBRACE

;

mode\_element

: ELEMENT LEFTBRACE mode\_name mode\_guard locvars? mode\_init? ( mode\_procedure | (mode\_element+) ) mode\_transition RIGHTBRACE

;

: COND LEFTBRACE expr\_bool RIGHTBRACE

;

//Module

module\_decls

: MODULE LEFTBRACE module\_element\* RIGHTBRACE

;

module\_element

: ELEMENT LEFTBRACE module\_name module\_oldName modeule\_description ;

//Data Dictionary

data\_decls

: DATA LEFTBRACE data\_element\* RIGHTBRACE

;

data\_element

: ELEMENT LEFTBRACE data\_oldName data\_newName data\_meaning data\_bit data\_type

## 4.4 错误诊断

### 4.4.1 变量检查

变量检查分为两种类型的检查。分别是静态检查和动态检查，其区别在于：

静态检查在生成中间模型的过程中，首先会根据代码去数据字典中查找所涉及变量，如果变量存在，则记录其名称，类型，初值等需要的参数。如果变量不存在，则提示出错，并编译不通过。

动态检查则直接报错，此时停止运行程序或者将错误写入日志文件并继续运行程序。

变量检查的步骤为：

1. 定义数据字典存储结构
2. 先遍历自定义结构中的数据字典子树，将其数据填入dataset结构中。
3. 接着遍历自定义结构中模式子树，如果结点属性为变量，则在dataset中查找其是否存在。如果不存在，提示出错，如果存在，则将该变量放在一个临时的存储模式中用到的变量list中。

### 4.4.2 类型检查

在我们的自定义结构(2.4.2)中，需要遍历其mode(模式)部分，如果检测到运算符(即Expression)，则通过递归实现检查其子节点类型是否匹配。共有三种类型：整型int， 浮点型float，字符串类型string。

类型检查的步骤为：

1. 遍历自定义中间结构的模式子树。
2. 若结点是运算符（+,-,\*,/,%,>,<,==,>=,<=,!=,=,&&,||都是二元运算符），则递归计算其左右子树类型，再检测两个子树类型是否匹配。
3. 计算表达式类型的算法

## 4.5 静态分析

### 4.5.1 Dataflow analysis思路

通过对中间模型的遍历，寻找到赋值语句中的左值与右值，判定右值对左值有直接影响关系，当赋值语句出现在条件语句中时，判定赋值语句的右值对左值的直接影响需要一定的条件。

对于一个语句结点中的变量x，如果它出现在该结点的定义变量集合中，那么判定它被这个结点中使用变量集合中的每一个变量影响。在判定间接影响关系的时候，如果在结点1中，x的影响集合为{y,z}，在下一个结点2中，y的影响集合为{w},那么x最终的影响集合为{z,w},其中z对x有直接影响关系，w对x有间接影响关系。在寻找变量的间接影响关系时，首先根据中间模型构建数据流图，记录每一个语句中变量的定义和使用关系，然后对相邻的结点进行变量的间接影响关系分析。如图4.1.1所示为间接影响关系分析示例。

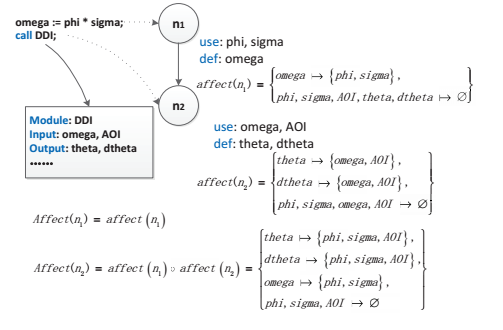


图4.1.1 间接影响关系分析示例

### 4.5.2 Dataflow analysis 算法

#### 4.5.2.1 直接影响关系分析

|  |
| --- |
| **Procedure**:DirectDataflow |
| **Input**: model:中间模型 |
| **Output:**dataflowPair:变量直接影响列表 |
| **Begin：** |
| **s**tmt :**=** model.getstmt |
| **switch** stmt **do** |
| **case** CompoundStmt |
| **foreach** element **in** stmt |
| DirectDataflow(stmt) |
| **case** AssignStmt |
| variable := stmt.leftvalue |
| assign := DirectDataflow(stmt.rightvalue) |
| **if** variable **in** dataflowPair |
| Index := getVariableIndex() |
| **foreach** element **in** assign |
| dataflowPair[index].add(element) |
| **else** |
| dataflowPair.add(assign) |
| **case** ChioceStmt |
| condition := stmt.condition |
| **c**onditionList.add(condition) |
| **foreach** element **in** stmt |
| DirectDataflow(element) |
| **foreach** element **in** stmt.TrueBlock |
| tmpassign := DirectDataflow(element) |
| **if** conditionList.count > 0 |
| **Find element variable index in dataflowPair** |
| **foreach** element1 **between** DirectDataflow[index].count - 1 **and** DirectDataflow[index] - counttmpassign.count |
| **foreach element in conditionList** |
| DirectDataflow[index].addcondition(element) |
| **case** CallStmt |
| **foreach** element **in** stmt |
| tmpassign := DirectDataflow(element) |
| **case** BracketExpr **or** Expr **or** PolymericExpr |
| **foreach** element **in** stmt |
| tmpassign := DirectDataflow(element) |
| **case** ExprID |
| tmpassign.clear() |
| basevar =convert.tostring(expr) |
| **t**mplist = new List<string>() |
| tmpassign.add(basevar) |
| tmpassign.add(tmplist) |
| **return** tmpassign |

由列表直接影响列表生成影响关系图

|  |
| --- |
| Procedure：ProcessDataflow |
| Input: dataflowList: 变量影响关系列表 |
| Output: restore\_graph: 变量影响关系图 |
| begin |
| **foreach** element **in** dataflowList |
| restore\_graph.addVertex(element.variable) |
| **foreach** element1 **in** element.definepair |
| **if** element1[0] **not in** restore\_graph |
| restore\_graph.add(element1[0]) |
| restore\_graph.addEdges(element1[0], variable, element1[1]) |
| **return** restore\_graph |

#### 4.5.2.2 间接影响关系分析

##### 4.5.2.2.1 基础函数

|  |
| --- |
| a) connect(g1,g2) |
| Vars = Vars1 ∪ Vars2 |
| Nodes = Nodes1 ∪ Nodes2 - {n1e, n2s} |
| Edges = {(n1,n2) |  (n1,n2) ∈Edges1 ∧ n2 ≠ ne1 ∨  (n1,n2) ∈Edge2 ∧ n1 ≠ ns2 ∨  (n1,ne1) ∈Edges1 ∧ (n2s,n2) ∈Edge2} |
| ns = ns1 |
| ne = ne2 |
| use = use1 ∪ use2 |
| def = def1 ∪ def2 |

|  |
| --- |
| b) merge(g1,g2) |
| Vars = Vars1 ∪ Vars2 |
| Nodes = Nodes1 ∪ Nodes2 - {n2s, n2e} |
| Edges = {(n1,n2) |  (n1,n2) ∈Edges1 ∨  (n1,n2) ∈Edge2 ∧ n1 ≠ ns2 ∧ n2 ≠ ne2 ∨  (ns2,n2) ∈Edges2 ∧ n1 = ns1∨  (n1,ne2) ∈Edges2 ∧ n2 = ne1 |
| ns = ns1 |
| ne = ne2 |
| use = use1 ∪ use2 |
| def = def1 ∪ def2 |

|  |
| --- |
| c) AddNewNode(g,nnew) |
| Nodes = g.Nodes ∪ {nnew} |
| Edges = g.Edges ∪ {nnew, g.ns} |
| ns = nnew |
| ne = g.ne |
| use = g.use |
| def = g.def |

|  |
| --- |
| d) SimpleDFG(s1, s2) |
| Vars = s1 ∪ s2 |
| Nodes = {ns, n1, ne} |
| Edges = {(ns, n1), (n1, ne)} |
| use = {ns → ∅, n1→ s1, ne → ∅} |
| def = {ns → ∅, n1→ s2, ne → ∅} |

connect函数用于顺序连接，merge函数用于分支情况的连接

##### 4.5.2.2.2 数据流图构建

|  |
| --- |
| **Procedure**: ConsturctDFG |
| **Input**: stmt: a statement |
| **Output**: dfg: constructed dataflow graph |
| **begin** |
| **switch** stmt **do** |
| **case** stmts1; stmts2 |
| g1 := ConstructDFG(stmts1) |
| g2 := ConstructDFG(stmts2) |
| **return** connect(g1, g2) |
| **case if** BExpr **then** stmt1 **else** stmt2 |
| g1 := ConstructDFG(stmts1) |
| g2 := ConstructDFG(stmts2) |
| g := merge(g1, g2) |
| g.Edges := g.Edges ∪ (g.ns, g.ne) |
| g.use(g.ns) = FV(BExpr) |
| **return** AddNewStart(g) |
| **case** x := e |
| **return** simpleDFG({x}, FV(e)) |

## 4.6 模型操作语义

### 4.6.1 新定义函数

在Expression部分，我们新增了多个函数，如下表所示:

|  |  |
| --- | --- |
| 函数体 | 解释 |
| Int Find(X1,X2,X3) | 第一个参数为表名，第二个参数为需要传入的参数1，第三个参数为需要传入的参数2，返回值为查表得到的结果。 |
| Int get(ID\_expr) | 第一个参数为端口，根据端口名称查找到初始值，并返回。 |
| abs(expr\_arithmetic) | 将expression取绝对值 |
| min(ID (,ID)+ ) | 取最小值 |
| max(ID (,ID)+ ) | 取最大值 |
| sqrt(expr\_arithmetic) | 取平方根 |
| power(number,power) | 所得值为number^power |
| sum(start,end, expr\_arithmetic) | 从start到end对expr\_arithmetic进行累加 |
| Output | 输出值到指定端口 |
| Sat | 将小于0和大于1的值就近映射到0或1，大于0小于1的值保持不变 |

在实现的过程中，我们为find，get，output，sum和sat分别设置对应的expr类型，abs，min，max，sqrt，power作为MathExpr类型进行存储。在生成模型时把对应的函数存成对应的模型，模型主要存储函数的参数。在模拟执行时，如果递归到相应的类型，那么针对当前类型模拟函数的语义。比如读到max函数，在模拟执行时，遍历MathExpr类型中的参数，取出其中最大值返回，就模拟了max函数的语义。

### 4.6.2时间算子

#### 4.6.2.1 duration和after

时间谓词duration(time,时间单位,条件)如果成立，当且仅当条件在距离当前状态时间间隔为time的状态内都成立，即可判断该布尔谓词为true，否则为false。

时间谓词after(time,时间单位，条件)如果成立，当且仅当其中的条件成立time之后，需要执行其他的CFG流程。

比如，需求文档中会出现这样一种表述：“连续100秒满足a>b”，这类表述我们在我们的规范中就用duration来表述，这句话在应用我们的规范之后应该写为“duration(100,s,a>b)”，这个算子的三个参数分别表达持续的时间，时间单位以及满足的条件。而对于需求文档中的另外一种可能的关于时间的表述：“满足Gmode=2之后240s”，这种表述可以用我们定义的after算子来描述：“after(240,s,Gmode=2)”。在这两个算子中，单位可以是秒(s),毫秒(ms), 周期等一系列的时间单位。

#### 4.6.2.2 实现

duration谓词和after谓词的第二个参数都为单位参数，一般是s和ms，在处理时都将其转换成ms，所以定义两个函数duration和after只需要两个参数，一个是guard，表示时间谓词中的布尔表达式b在当前状态的真假。第二个参数使bound，对应时间谓词中的时间长度。两个函数实现如下：

|  |
| --- |
| float clock\_a = 0.0f;  int after (int b,float bound)  {  if (clock\_a >0.0f)  {  clock\_a += 1;  if (clock\_a >= bound)  {  clock\_a = 0.0f;  return 1;  }  }  else  {  if (b != 0)  clock\_a += 1;  }  return 0;  } |

|  |
| --- |
| float clock\_b = 0.0f  int duration (int b,float bound)  {  if (b != 0)  {  clock\_b += 1;  if (clock\_b >= bound)  {  clock\_b = 0.0f;  return 1;  }  }  else  {  clock\_b = 0.0f;  }  return 0;  } |

### 4.6.3模式操作语义

在模式层，我们关心数据字典定义的全局变量，故而State的变量集合是这些全局变量以及一个特殊的timestamp变量构成的集合，它的语义空间定义如下：

* MD表示整个模式图，表示系统所处的状态。
* 用于表示系统所处的阶段，Begin为周期初始，Execute表示准备执行周期任务，End表示周期末尾。
* ,其中是一个程序计数器，用于识别模式体的控制流执行情况，为控制流的内部为指标记，Start和Exit分别表示控制流的开始和结束为止。如果当前模式没有控制流，则使用符号作为占位符。
* k为周期计数器，每个周期加1。如果系统发生模式切换，那么k会被置1.
* E为变量状态构成的序列，记录了历史状态。序列中每个元素都是一个变量名到变量取值的映射。

首先给出逻辑表达式的解释：

|  |  |
| --- | --- |
| 1-常量 |  |
| 2-常量 |  |
| 3-谓词 |  |
| 4-否定 |  |
| 5-合取 |  |

Expression在语义空间的语义解释如下表：

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  | 其中E是长度为n的状态序列， |

如果该Expression为布尔项(b),则改Expression在状态序列上成立，当且仅当b在的最后一个状态上的求值结果为true。

确定了语义空间以及状态表达式和卫式的语义后，本文给出模式级别的操作语义规则。推理规则的形式如下：

其中，g是应用推理规则的条件，用于描述模型在语义空间的状态变迁。模式级别的语义主要关心模式切换以及按周期执行模式的控制流等行为，忽略流程图的具体执行过程，而将其概括为语义空间中的一次状态变迁。为此，定义函数execute描述流程图的执行效果：

输入参数的第一元是待执行的流程图，第二元是一个程序计数器，表示执行的起始位置，第三元是一个程序状态，第四元是一个非负实数，表示执行时间，一般而言，是指含有该流程图的模式的周期长度。返回值的第一元是一个程序计数器，表示执行结束时的流程图位置，第二元表示结束时的程序状态，第三元为非负实数，表示执行该CFG耗费的时间。

下表为操作语义：

|  |  |
| --- | --- |
| （采样） | 其中，,    表示从传感器获取的值 |
| (执行) | 其中， |
| (重复) |  |
| (切换) | 其中， |

规则（采样）表示如果系统初始周其初始，那么系统通过传感器采样，用传感器获取的值更新系统中的变量（一般预置在数据字典中），然后开始执行模式的控制流。

规则（执行）描述以程序计数器pc为程序起始位置，以当前模式的周期长度为执行时间，执行当前模式的控制流。函数execute获取控制流的执行结果，程序计数器从位置pc变为pc’，新的程序状态为。

如果控制流正常执行结束（pc=Exit），则检查模式迁移的条件。规则（重复）描述了如果所有模式迁移的条件都不满足，那么系统将在下个周期继续处于当前模式，规则（切换）描述了如果某个模式图迁移的条件满足，且其优先级该与其他任何条件也满足的迁移，那么该迁移将会发生。该模型将会在下个周期转入目标模式。

### 4.6.4 控制流流的控制语义

控制流由语句构成，即statements，除了语句，还有分配给该控制流的执行时间和待执行的语句，所以他的语义空间定义如下：

其中是变量赋值，代表全局变量。其中，t是一个非负实数，表示分配给该控制流的执行时间，stmts为待执行的语句，语义空间上的推理规则如下:

公式中，g表示该推理规则运用的条件，代表控制流在语义空间中的状态。 代表发生这个迁移耗费的时间。根据被执行的语句可以推算出耗费的时间，函数的第一个参数表示执行之前的语句，第二个参数表示执行之后的语句，返回值表示执行语句耗费的时间。

下表为控制流的语义推导规则：

|  |  |
| --- | --- |
| 赋值-1 | 其中 |
| 选择-T |  |
| 选择-F |  |
| 顺序- |  |
| 顺序 |  |

上表中 ，表示空语句，且对任意语句s，都有 ，对于任意都有，箭头表示被执行语句stmts变迁为stmts’所耗费的时间为x。

赋值语句中，右值赋给左值，此时，全局变量的状态会根据update函数更改，它定义如下：

由于被调用的模块应当满足原子性，即如果无法在给定的t时间内执行完成，则不能执行该模块的调用。

根据给定执行语句的时间消耗函数

可以定义控制流的execute函数如下：

首先根据输入参数构造控制流语义空间的初始状态,其中stmt是cfg中的语句，其初始位置标签为l，根据上表进行推理时，予以状态，则返回值的取值如下：

# 5. 结论

在这份报告中，我们提出了一种面向航空领域的控制软件需求建模形式化工程方法，设计与实现了针对商用航空发动机需求文档形式化模型语言。

在设计语言的过程之中，为了方便需求撰写人员将自然语言需求文档转化为半形式化语言需求文档，我们还提供了需求模板和需求撰写规范帮助进一步规范需求文档。在规范化的过程中，需求分析人员可利用我们提出的建模语言需求文档描述的质量，减少潜在的错误与缺陷。

我们开发了AEDL工具用于支撑本次建模工作的读取与分析工作。AEDL主要支持检查半形式化需求文档的语法错误以及类型错误。在撰写本报告的时间结点内，我们正在进行AEDL的进一步开发，实现模型的分析与验证工作，包括静态分析，动态分析，模型性质检查，测试用例自动生成等工作。

目前该建模语言已经设计完毕，并且已经完成了某型号需求文档的半形式化转化工作。解决了如下问题：

1).规范原有需求文档，使其条目化。

2).整理需求文档逻辑性，使其更清晰。

3).生成中间模型，为下一步模型检查奠定基础。

同时，也发现了如下问题：

a).输入未完全列举。

b).自然语言描述需求的二义性错误。

由于文档涉密所以在这里只能简述b错误：工程师在转化为半形式化需求文档的过程中无法准确理解“定时任务”这个术语具体指的是什么，关于如何理解不同的工程师又产生了分歧。此为典型的需求文档描述错误，即需求撰写人员和需求开发人员对于同一个词理解是不尽相同的。而在转为为我们设计的半形式化模型之后，工程师对于转化后的模型理解均一致，无二义性产生。

上面这个案例体现了形式化建模语优于工业界传统手工审查方法的缺陷发现能力，也证明了我们工作是卓有成效的。

在本次项目合作中，因为原有需求文档有很长的篇幅，我们双方在转化为半形式化文档的阶段花费了大量的时间。如果自然语言处理可以代为解决这一步，那么我们的工作效率会得到很大的提升。因此，未来我们将进一步探索自然语言处理技术与需求建模活动相结合的方法，尽可能将需求撰写中的人为因素和主观因素降低以减少需求文档中潜在的缺陷和错误，并提高建模效率。

# 硕士期间发表的学术论文、专著、重要科研成果

**学术论文：**

1. Jincao Feng, Jiayi Shen, Yang Liu, Yihao Huang, Xiaoxuan Chen, Weikai Miao and Geguang Pu, A Formal Engineering Method for the Requirements Modeling of Railway Embedded Control Software, The 2nd National Conference on Formal Methods and Applications , 2017.(第一作者)