软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2023,34(7):3080−3098 [doi: 10.13328/j.cnki.jos.006861]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于 SMT 的区域控制器同步反应式模型的形式化验证*

李腾飞[1,2], 孙军峰[1], 吕新军[1], 陈 祥[1], 刘 静[2], 孙海英[2], 何积丰[2]

[1](卡斯柯信号有限公司, 上海 200072)

[2](华东师范大学 软件工程学院, 上海 200062)

通信作者: 孙军峰, E-mail: sunjunfeng@casco.com.cn; 刘静, E-mail: jliu@sei.ecnu.edu.cn

摘 要: 在安全关键系统的软件开发过程中, 形式化验证是一种经检验的提高软件质量的技术. 然而, 无论从理论上还是从应用角度来看, 软件的验证都必须是完整的, 数据流验证应该是对实现层软件模型进行验证的必要体现. 因此, 环境输入、泛型函数、高阶迭代运算和中间变量对于分析形式化验证的可用性至关重要. 为了验证同步反应式模型, 工程师很容易验证控制流模型(即安全状态机). 现有工作表明, 这类工作无法全面地验证安全关键系统的同步反应式模型, 尤其是数据流模型, 导致这些方法没有达到工业应用的要求, 这成为对工业安全软件进行形式化验证的一个挑战. 提出了一种自动化验证方法. 该方法可以实现对安全状态机和数据流模型的集成进行验证. 采用了一种基于程序综合的方法, 其中, SCADE 模型描述了功能需求、安全性质和环境输入, 可以通过对 Lustre 模型的程序综合, 采用基于 SMT 的模型检查器进行验证. 该技术将程序合成作为一种通用原理来提高形式化验证的完整性. 在轨道交通的工业级应用(近 200 万行 Lustre 代码)上评估了该方法. 实验结果表明, 该方法在大规模同步反应式模型长期存在的复杂验证问题上是有效的.

关键词: 形式化验证; 安全关键系统; 同步反应式模型; 高阶迭代; 程序转换

中图法分类号: TP311

中文引用格式: 李腾飞, 孙军峰, 吕新军, 陈祥, 刘静, 孙海英, 何积丰. 基于 SMT 的区域控制器同步反应式模型的形式化验证. 软件学报, 2023, 34(7): 3080–3098. http://www.jos.org.cn/1000-9825/6861.htm

英文引用格式: Li TF, Sun JF, Lü XJ, Chen X, Liu J, Sun HY, He JF. SMT-based Formal Verification of Synchronous Reactive Model for Zone Controller. Ruan Jian Xue Bao/Journal of Software, 2023, 34(7): 3080–3098 (in Chinese). http://www.jos.org.cn/1000-9825/6861.htm

SMT-based Formal Verification of Synchronous Reactive Model for Zone Controller

LI Teng-Fei[1,2], SUN Jun-Feng[1], LÜ Xin-Jun[1], CHEN Xiang[1], LIU Jing[2], SUN Hai-Ying[2], HE Ji-Feng[2]

[1](CASCO Signal Ltd., Shanghai 200072, China)

[2](Software Engineering Institute, East China Normal University, Shanghai 200062, China)

Abstract: Formal verification is a proven technique for improving product quality during software development of safety critical systems. However, the verification must be complete, both theoretically and in the interest of practicality. Data-flow verification is a pervading manifestation of verification of the software model in implementation level. Environmental input, generic function, high-order iterative operation, and intermediate variables are therefore crucial for analyzing usability of verification approaches. To verify a synchronous reactive model, engineers readily verify the control-flow model (i.e., safe state machine). Existing work shows that these approaches fall short of complete verification of synchronous reactive model of industrial software, which results in the loss of reaching the industrial requirements. It presents a significant pain point for adopting formal verification of industrial software. Thus, it is drawn on the insight that the synchronous reactive model of safety-critical systems should be verified completely, and the data-flow models should be considered. An approach is presented for automated, generic verification that tailor to verify the integration of safe state machines and

* 基金项目: 国家重点研发计划(2019YFA0706404); 国家自然科学基金(61972150); 上海市超级博士后基金(2021146)
本文由“形式化方法与应用”专题特约编辑董云卫教授、刘关俊教授、毛晓光教授推荐.
收稿时间: 2022-09-04; 修改时间: 2022-10-08; 采用时间: 2022-12-05; jos 在线出版时间: 2022-12-30

data-flow models. Furthermore, a synthesis-based approach is adopted where the SCADE models describe functional requirements, safety requirements and environmental inputs that can be verified for an SMT-based model checker through program synthesis to Lustre model. The proposed technique promotes program synthesis as a general primitive for improving the integrity of formal verification. The proposed approach is evaluated on an industrial application (nearly two million lines of Lustre code) in rail transit. It is show that the proposed approach is effective in sidestepping long-standing and complex verification issues in large scale synchronous reactive model.
Key words: formal verification; safety-critical system; synchronous reactive model; high-order iteration; program transformation

开发高质量、无错误的安全关键系统软件, 在很大程度上仍然是一个非全自动化且昂贵的过程. 形式化验证在模型驱动开发过程和在自动保证软件模型安全性上至关重要, 然而在应用形式化方法的实践中, 存在许多开放性挑战. 形式化建模和验证在理论上和实践上都必须考虑实际系统的复杂性, 开发人员对工具是否无缝集成到其现有工作流中很敏感. 至少, 形式化验证必须应用于验证具有状态机和数据流的安全关键模型对其安全性质的满足情况, 并提供消除同步反应式模型中的高阶迭代等机制. 问题是: 对验证过程的方法选择基本上是通用的, 输入假设和模型现实(即行业级应用)之间的差异可能会导致无效、无意义或错误的输出.

安全需求和路线地图的环境输入对于安全关键系统的形式化建模和验证过程至关重要. 常见方法包括对于状态机的验证(通常过于粗糙)[1], 这忽略了环境输入, 如在与每个时刻的路线地图相关联的安全需求. 将环境输入作为模型的一部分和对数据流模型的验证[2], 是对同步反应式模型进行形式化验证的一个挑战.

本文提出一种基于程序综合的方法, 通过制定 SCADE 语言[3]与 Lustre 语言[4]的语法转化规则, 转换了在语法上不匹配的部分, 并消除了冗余部分, 实现了 SCADE 安全模型到 Lustre 程序[5]的转化. 本文方法可被视为一个预处理步骤, 使用轻量级语法更改裁剪分析程序. 它在状态机和数据流模型的完全转换的基础上运行, 还可以在建模工程师开始对安全关键系统建模之前获取环境输入. 由于安全保证出现在整个软件生命周期中, 因此将本文方法的转换方法和工具作为通用方法, 在原有软件安全保证的基础上集成软件模型的形式化验证, 不仅可以减轻测试工程师的工作量, 也可以分摊质量安全工程师在系统安全分析的压力. 本文基于程序综合, 使得软件模型可以在语法层面实现所需的更改, 对转换的语义属性的研究强调了提高分析精度的潜力. 尽管数据流验证具有现实性和理论性, 但目前在实践中几乎没有证明其在真实工业控制领域中得到应用.

本文工作的一个关键目标是, 在实践中证明这个想法的可行性、适用性和有效性. 为此, 本文在 ZC 子系统中对大规模真实系统进行了评估. 一个重大挑战在于数据流模型, 尤其是模型中出现高阶迭代. 程序转换中的最新技术有助于应对这一挑战, 并为本文的方法的实施奠定了基础. 本文考虑: (a) 作为环境输入的路线地图; (b) 用于分析控制流源的数据流转换, 以解决同步反应式模型的验证问题. 本文开发转换程序以解决分析实现和推理中的缺陷. 本文的贡献如下:

- 本文将环境输入和安全性质作为建模安全关键系统的入口.
- 本文提出了同步数据流语言 SCADE 每个操作符的转化算法, 并表明本文的方法在 SCADE 模型的数据流模型转化, 特别是高阶迭代算子上是有效的.
- 本文证明了在基于 SMT 模型验证器对 SCADE 模型的形式化验证中是有效的, 并将验证方法与 SCADE 建模相结合.
- 本文在轨道交通 ZC 子系统反向跳跃的工业级模型(近 200 万行 Lustre 代码)上进行实验, 并分析了 4 种不同平台的验证能力.

本文第 1 节介绍 SCADE 模型转化的相关方法和研究现状. 第 2 节介绍本文所需的基础知识, 包括 ZC 子系统、SACDE 建模和 Jkind 验证器. 第 3 节介绍本文构建的安全模型. 第 4 节介绍本文提出的 SCADE 模型转化. 第 5 节进行实验并对结果进行分析. 最后总结全文.

1 相关工作

将本文的工作与 SCADE 模型形式化验证的类似想法进行比较. Ran 等人[6]提出了安全状态机到 SCADE 模型的转换, 并基于 Jkind 验证了转换后的 Lustre 模型. SCADE 模型中被忽视的高级操作符和 Sensor, 意味着

他们的工作只适用于验证安全关键系统部分模型的安全性. 另一个与本文类似的工作[7]也是通过 Jkind 来对 SCADE 模型进行验证, 然而该工作转换的 Lustre 模型规模并不大, 而且有些复杂的功能并没有实现转化, 导致其工具的可扩展性缺乏足够的说服力. Shi 等人[1]提出了状态机从 SCADE 到 NuSMV 的转换, 并通过模型检查器 NuSMV 验证了 SCADE 模型, 数据流转换的丢失限制了其在实际工业安全关键系统中的应用. 与之相似, Aniclulaesei 等人[8]借助 NuSMV 对 SCADE 模型进行形式化验证, 并且根据反例生成测试用例. 首先, 他们将形式化验证作为一个前提方法和辅助手段, 并且从转化后的 NuSMV 规模上看, 该系统规模被抽象在一个可验证的范围内, 这对方法的复用和对其他子系统进行验证的推广意义不够. 另一项与本文类似的工作将 SCADE 模型的子集转换为 LAMA 模型, 并通过求解器 Z3 验证转换后的 LAMA[9]. 他们的案例研究的大小, 即 SCADE 文件, 只有几千字节. 显然, 他们的工作并没有触及形式化验证能力的瓶颈. 虽然周佳铭[10]通过将 SCADE 模型转换为 Lustre 模型来进行验证, 但转换过程只涉及状态机和部分数据流元素, 忽略了复杂的 Sensor、数组、结构和高阶操作和 if block 等的转换, 导致该方法难以对一个实际工业系统的 SCADE 模型进行转化, 更加难以对验证能力上限进行评价. Bennour[11]对嵌入式系统的某些功能建模为带时间的同步数据流图(timed synchronous dataflow graphs, Timed SDFG), 将 Timed SDFG 模型转化为 Lustre 模型, 并基于 Kind2 对 Lustre 模型进行形式化验证. 尽管其验证器 Kind2 与本文采用的 Jkind 在原理和验证能力上是比较类似的, 其工作与本文有以下不同点: 第一, Bennour 没有对大规模工业控制系统进行建模, 而是对嵌入式系统具体的某个功能采用 SDFG 建模; 第二, Timed SDFG 是一种抽象模型, 其以 actor 作为图的节点, 屏蔽了每个 actor 内部的具体实现, 而 SACDE 通过数据流有反应式系统的细节; 第三, 由于缺少具体的变量, 其通过状态和迁移的个数来衡量系统规模也是不精确的.

基于全局异步和局部同步(GALS)的分层验证[12]是通过将契约网络转换为模型验证工具 SPIN 或 UPPAAL 的模型进行的. 同步组件在 SCADE 中描述, 并使用 SCADE 设计验证器进行验证. 然而, 对于区域控制器(zone controller, ZC)子系统, 它只涉及 SCADE 模型中的同步部分, 并且需要与具有接口的其他子系统进行交互, 而不是全局异步通信. 类似地, 另一个基于 GALS 的验证[2]提出了一个基于观察器的 C 代码验证器, 该验证器由 SCADE 模型生成. 对于复杂系统而言, 基于 AADL/AGREE 抽象架构模型的组合验证[13]似乎很有趣. 然而, 抽象的架构模型丢失了实现细节, 导致对工业系统完全模型的安全保证不够全面.

尽管 Ferrari 等人[14]对轨道信号系统设计领域的形式化方法工具进行了综述, 但他们的工作仅限于英文的文章, 而且倾向于研究型工具, 这就使得他们所综述的工具对于工业级系统进行建模和验证的适用性上会暴露出一定的问题. 因为工业问题一个最明显的特点是系统规模复杂、变量多, 对整个系统的完整性进行分析, 尤其是符合行业标准的实际系统的正确性进行形式化验证上, 需要在形式化理论、方法、工具、产业化适用、推广等各个方面都需要进行长期的投入. Liu 等人[15]对区域控制器子系统的验证聚焦于模型模块的分解上, 通过问题框架的方法, 将系统模型映射为多个可验证的子问题, 并基于映射后的子问题进行验证. 相同点都是在模型规模上进行取舍, 与本方法的方法的区别在于, 本方法在保证功能完整性基础上进行验证; 而 Liu 等人的工作聚焦于部分模型的验证, 其方法被限制于: 第一、子问题之间是等价的“与”“或”“非”操作; 第二、分解后的子问题必须是可验证的. 当子问题之间的特征不够明显, 或者子问题引起功能较复杂使模型规模大到无法验证时, 该方法的适用性将被受限.

SCADE 工具集成了 Prover 验证器[16]. 该验证器已成功应用于联锁子系统. ZC 子系统中的数据流和控制流比联锁子系统更复杂. 首先, 联锁子系统处理联锁设备中的布尔变量, 如道岔、计轴、轨道电路、信号机等. ZC 子系统操作复杂的数据结构, 如区段、列车包络和警冲区(fouling zone, FZ)的标识、长度和坐标等, 都是在许多约束条件下抽象出来的. 另一方面, Prover 仅验证本地联锁子系统, 因为它取决于某个具体的车站联锁信号布局图. 然而, 区域控制子系统控制一条跨越多个车站的较长轨道.

2 基础知识

下面对本文所涉及的 ZC 子系统、SACDE 建模和 Jkind 验证器相关知识进行介绍.

2.1 ZC子系统

ZC 子系统负责列车的安全保护[17], 它接受自动列车监视子系统(automatic train supervision, ATS)、联锁系统(interlocking, IC)和数据存储单元(data storing unit, DSU)的输入, 并计算列车的移动授权(mobility authorization, MA). 具体来说, 一旦收到列车的移动授权请求, ZC 将查询联锁设备的状态. 联锁系统收到 ZC 的查询请求后, 将道岔、区段、信号机等设备的状态发送给 ZC 子系统. ZC 收到联锁系统发送的联锁设备状态后为列车计算移动授权, 并将授权结果发送给列车. 当进入 ZC 边界时, ZC 子系统还需要向邻近 ZC 发送补充移动授权请求. ZC 在收到补充移动授权时, 将其与自身计算的移动授权进行拼接, 然后将拼接后的移动授权结果发送给列车.

为了列车的安全运行, ZC 必须具备处理所有场景的能力. 在计算移动授权状态的过程中, 不同的场景将激活相应的模块. ZC 子系统将计算设备的状态, 如屏蔽门(platform screen door, PSD)、警冲区(fouling zone, FZ)、车库门(depot gate, DG)等. 需要搜索每列列车和 MA 区域的列车包络以确定授权结果. 同时, 由于列车运行过程中硬件故障、通信失效等因素的影响, 要求 ZC 在核心功能的基础上进行功能的拓展, 以保证其能够应对这些突发情况下的安全运行. 在 ZC 子系统的 SCADE 模型中, 每个场景的设备或区域的个数、状态等信息都被存储为数组或结构体, 甚至是复合数据结构. 模型处理这些数据需要大量复杂的操作符, 这导致了大量的计算过程.

2.2 SCADE建模

同步数据流语言基于同步假设: 对于每个输入, 系统将在下一个输入之前完成计算并输出结果. 这一假设确保了任意两个反应之间不会有重叠.

同步语言是反应式系统的特征. 动作(输入-计算-输出)可以在瞬间执行.

SCADE 继承了同步语言 Lustre 和 Esterel[18]的特点, 对反应式系统进行编程. 具体来说, 每个时刻的数据流计算输入局部变量的结果, 并将输出传输到另一个节点或函数. 时间算子, 如前一个时刻 pre、初始时刻 init、延迟算子 fby 等, 在其上下文中记录变量的计算历史. 此外, SCADE 借用了函数式语言的一些特性, 如数组操作、高阶函数. 这些特性支持 SCADE 并行表达迭代或循环操作, 从而提高了对同步反应式系统建模的能力. 此外, SCADE 和面向对象编程语言一样, 引入了泛型函数以提高函数的可重用性.

2.3 Jkind模型验证器

Jkind[19]是一种基于 SMT 的模型检测器. 在默认情况下, BMC 引擎会执行".solver"标志指向 SMT 求解器的使用, 如 z3、yices2、mathsat、cvc5、SMTInterpol. 现在, Jkind 并行运行以下引擎, 试图证明 Lustre 模型中的属性.

- 有界模型检查(bounded model checking, BMC): 该引擎运行并试图找到反例. 其搜索总是从模型的初始状态开始, 因此随着 K 值的增大, 搜索空间的大小呈指数增长. 这也有助于确定基本情况在 k-归纳推理方法中是真的. 这个引擎可以用来证明和寻找反例.
- k-归纳(k-induction): 该引擎试图证明 k-归纳证明的推导步骤是正确的. 该算法从任意状态进行搜索, 并试图证明 k-归纳步骤是正确的. 这个引擎可以用来证明.
- 不变式生成(invariant generation): 该引擎帮助推导引擎查找有助于推导步骤覆盖的不变式. 这有点复杂, 但如果没有不变生成引擎, k-归纳法实际上无法解决困难的问题. 它还可以识别有助于 PDR 的不变式, 所以最好保持打开状态.
- 性质有向可达性(property directed reachability, PDR): 该求解器独立于其他 3 个引擎运行, 是自动证明性质的替代方法. 它与其他引擎共享学习到的信息, 并且这些工具确实工作得最好. 应至少有一个 PDR 引擎.

默认情况下, Jkind 使用 BMC 引擎、k-归纳引擎、不变式生成引擎和 PDR 引擎.

3 安全模型的构造

3.1 安全性质的分类

ZC 是 CBTC 系统的核心子系统. 它负责计算列车的移动授权(MA), 在 CBTC 系统中起着至关重要的作用. ZC 子系统的安全需求从 CBTC 系统需求中分解. CBTC 系统中的功能需求描述了系统可以做什么, 安全需求则表示系统应该满足哪些约束. 在上下文中, 功能需求是指系统或子系统需求, 安全需求又称为功能安全.

根据系统定义, 首先对系统风险进行分析和评估; 然后, 根据风险评估得到 CBTC 系统需求规范; 然后, 从需求规范中划分子系统, 并在子系统需求规范[20]中分配安全需求. 根据对实际功能的分析, 可提取出系统的安全性质.

根据需求中提取的安全性质, 本文将 ZC 子系统的安全性质分为 3 类, 如图 1 所示.

- 环境安全是线路地图的环境约束列表. 线路地图中的元素包括区段、道岔、列车包络、分支、屏蔽门、警冲区、紧急停车区(ESA)、车库门等. 这意味着线路地图中的元素必须满足不同场景下的特定约束. 例如, 当道岔的状态改变时, 向上或向下的区段索引必须满足其约束.
- 功能安全是指 ZC 子系统需要满足的要求. 例如, 如果可用反向跳跃区域、允许警冲区、允许 ESA 和允许 PSD 的交点有效, 则反向跳跃区域的跳跃授权可用.
- 授权安全分为移动授权状态下, 授权的条件必须满足的约束. 也可以表达为状态不变性, 是区域控制子系统在每个时刻必须满足的条件. 该安全约束是可以从其他模块计算的布尔表达式进行取与操作.

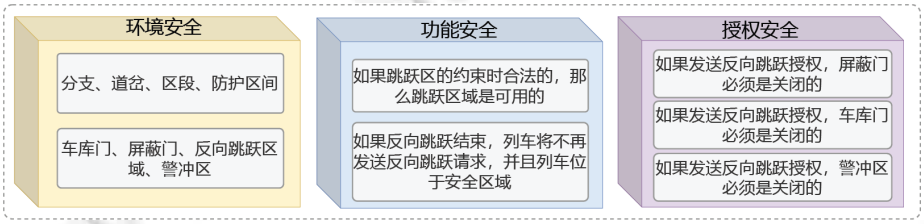


图 1 安全性质分类

每个安全需求在 ZC 子系统需求规范中进行了标记, 可以通过 SCADE 工具建模为安全模型.

3.2 安全性质的规约

依据 ZC 子系统的安全分配, 从实际功能中提取出的安全性质, 可以在 SCADE 中建模. 根据第 3.1 节的安全性质分类, 每一种类都包含较多的安全性质. 表 1 对每一类具有代表性的部分安全性质进行了形式化规约. 可通过读取线路地图中的值来描述线路地图的安全性质. 它可以分为物理结构约束和抽象数据结构约束. 物理结构的约束由需求 R_1 描述. 如果一个道岔的状态是定位 *P_N*, 在区段 *B_b* 的下行方向, 下一个区段是 *B_a*. 而如果这个道岔的状态是反位 *P_R*, 区段 *B_b* 的下行方向的下一个区段是 *B_c*. 对于线路地图的所有区段, 本文可以使用高阶运算实现道岔和区段之间的所有约束. 像列车包络这样的抽象数据结构包含列车的长度加上延伸长度, 该安全需求如表 1 的 R_2 所示.

环境的安全性质需要通过一个复杂的过程来计算, 因为环境的数据结构通常是数组和结构的嵌套. R_3–R_6 中描述了此类安全性质. 例如, 移动授权状态需要一些复杂计算的合成过程. 授权安全性质描述了授权状态的约束, 对于授权状态可达性描述为: 从初始状态 *JZ_Init* 出发的一条执行路径, 可以到达授权状态. 如安全性质 R_7 所示, 只要可以计算出 *JZ* 状态, 就可以通过 *k*-induction 引擎导出状态不变式. 迁移的约束不仅需要移动授权的状态, 还需要计算两个状态之间迁移的触发条件, 如安全性质 R_8 所示, 从初始状态 *JZ_Init* 出发的一条路径, 如果前一状态是等待授权状态 *JZ_WAIT_AUTHO*, 当前状态是授权状态 *JZ_AUTHO*, 那么授权条件 *bCondition* 是成立的.

表 1 安全性质的规约

分组	安全性质	形式化规约
R_1	如果道岔状态为定位, 区段 *B_b* 的下行方向的下一个区段是 *B_a*, 如果道岔状态为反位, 区段 *B_b* 的下行方向的下一个区段是 *B_c*,	(*P_N*⇒*B_b.D*$_n$=*B_a*) && (*P_R*⇒*B_b.D*$_n$=*B_c*)
R_2	给出列车包络所在分支上的一个自然序, *AP_a* 增 方向的下一个 *AP* 是 *AP_b*, 减方向的 *AP* 是 *AP_c*	(*AP_a.Inc*=*AP_b*) && (*AP_a.Dec*=*AP_c*)
R_3	授权状态时, 跳跃请求是否合法	(*State*=*JZ_AUTHO*)⇒*bRJZ_IsValidRevJogReq*
R_4	授权状态时, 车库门是否关闭	(*State*=*JZ_AUTHO*)⇒*bJZ_IsAvailable_DepotGate*
R_5	授权状态时, 退行区没有其他车	(*State*=*JZ_AUTHO*)⇒*bRJZ_IsNoOtherTrain*
R_6	授权状态时, *MAOVChain* 是否被占用	(*State*=*JZ_AUTHO*)⇒ *bRJZ_ActMAOVChain_IsNotOcc*
R_7	授权状态可达性	(*JZ_Init*→*pre State*)=*JZ_AUTHO*
R_8	授权状态迁移	(*JZ_Init*→*pre State*)=*JZ_WAIT_AUTHO* and *State*=*JZ_AUTHO*)⇒*bCondition*

4 程序转化

通过从 SCADE 模型到 Lustre 模型的模型转换, 合成 Lustre 程序. 该过程需要处理类型、常数、Sensor、变量、运算符(即算术、逻辑、比较、结构、数组、时间、高阶)和组成(节点、函数、包、if block、状态)的转换. 此外, 在一些特定形式中, SCADE 引入了泛型函数以提高函数的可重用性, 而 Lustre 忽略了性能. 表 2 给出了 SCADE 模型到 Lustre 模型在语言表达能力上的映射规则.

表 2 SACDE 模型转化 Lustre 模型的映射规则

语言	SCADE	Lustre
语法规则	常量	常量
	Sensor	
	类型	类型
	数学操作	数学操作
	逻辑操作	逻辑操作
	时态操作	时态操作
	复杂数组、结构体操作	简单数组、结构体操作
	泛型函数	一般函数
	高阶迭代	顺序执行函数
	if block	if-then-else 等式
	状态机	
	node	node
	function	

最基本的常量和类型在两个语言之间是等价的, 但是由于 Lustre 的定位是作为底层语言, 允许上层建模语言的转化, 不直接与环境进行交互, 因此需要将 Sensor 进行转化为常量来处理(见第 4.1 节). 数学操作和逻辑操作在两个语言中是同等表达的, 不需要多余的处理即可转化. 对于数组操作, 由于实际的应用中需要大量的数组来存储数据, 因此, SCADE 语言构建了许多数组算子来对这些数组进行处理; 而 Lustre 并没有这些处理, 因此需要将这些复杂的数组操作转化为 Lustre 中简单的数组操作. 结构体的操作与数组操作类似(见第 4.2 节). SCADE 可以采用 if block 及其嵌套来表达一个代码块的执行, 但 Lustre 不具备这一特色, 因此需要转化为 Lustre 可识别的 if-then-else 等式. SCADE 借鉴面向对象语言, 引入了泛型函数, 增加了函数的可复用性, 这一特色可被转化为 Lustre 的一般函数, 但是需要根据调用该泛型函数的地方传入具体的类型和数组长度. 由于早期的同步语言是从同步电路中引申而来, 缺少循环的表达, SCADE 为了增加语言的表达能力, 引入了高阶迭代函数来表达函数的重复调用, 这一特色需要转化为 Lustre 的一般顺序执行函数. 状态机可以更加清晰地表达系统的控制流, SCADE 引入了安全状态机, 从早期专注于声明式语言转变为同时具备声明式语言和命令式语言的特征. 这一特征可以转化为 Lustre 的 if-then-else 等式. 对于同步语言的一个 node 或 function, 用于组合一个数据流的模型结构块是组件模型分层组合嵌套的最小单位, SCADE 中的 node 会记录内存, 而 function 不需要. Lustre 将涉及等式运算的组合单位都称为 node, 其使用 function 是为了抽象. 因此, 需要将

SCADE 中的 node 和 function 全部转化为 Lustre 的 node.

此外, SCADE 引入了局部变量作为常量、输入变量、输出变量、算子和函数之间的传值接口, 这些中间变量方便了 SCADE 对数据流的表达. 在程序综合过程中, 这些中间变量又增加了 Lustre 模型的局部变量和方程的个数. 消除这些中间变量有助于大量减少方程的个数, 进而大大减少模型验证过程中的计算量.

4.1 数组操作和结构体操作的转化

SCADE 语言定义了多个数组和结构体操作, 这些数组操作集成了复杂的数据结构和处理数组的基本算法. 这些操作集成了一些高级算法, 作为对数组或结构的原始操作. 数组操作包括数据结构、数据数组、标量到矢量、切片、串联、反转、转置、投影、动态投影. 结构体操作包含结构体元素的赋值、生成、展平. 数组和结构体算子的转换将合成操作展开为单个操作. 本文提供了数组切片操作的算法 1. 读者可以仔细推导其他运算符的变换.

算法 1. 切分数组操作的转化.

Input: *SliceOp Slice*, *Variables v*.

Output: A transformed equation *eq*.

```
1:  Initialized: eq←"v=["
2:  fromIndexNum, toIndexNum, array←Slice
3:  for i>fromIndexNum and i≤toIndexNum do
4:      eq.append("array[i] ")
5:      if i≠toIndexNum then
6:          eq.append(",")
7:      end if
8:  end for
9:  eq.append("];")
10: return eq
```

数组切片操作的转换将切片操作和变量作为输入, 并返回转换后的方程 *eq*. 该方程将字符串初始化为 "*v*=[". 从运算符切片操作中, 获得数组、起始索引和结束索引. 在从头到尾的循环过程中, 转换后的方程附加数组的每个元素, 后跟逗号. 最后, 该方程附加了一个右方括号. 数组和结构运算的转换是消除 if block 和高阶运算的基础.

SCADE 中有很多复合的数组操作和结构体操作, 如标量转矢量操作、数组转置、数组映射、结构体展开等等, 这些操作方便了建模时对系统的描述. 本节给出数组切片操作的转化算法, 由于数组和结构体的转化都是直接转化为简单的数组和结构体操作, 因此读者借鉴切片操作的转化算法, 能够很容易地将 SCADE 语言中这些复合的数组操作和结构体操作转化为 Lustre 中对应的简单操作.

4.2 Sensor转化

SCADE 中的 Sensor 描述了列车运行时的不同场景. 这些 Sensor 可以通过线路地图实现. 由于相应的 Lustre 语言不支持外部变量, 因此有必要预先存储 Sensor 的值. 首先, 在分析 SCADE 模型时, Sensor 应存储在哈希图中, 其节点名或函数名为密钥; 其次, 叶子节点(其子节点没有 Sensor)应标记为叶子节点; 第三, 将为 Sensor 分配一定的值, 以便无论 Sensor 出现在哪个节点, 都可以使用 Sensor 进行计算; 第四, Sensor 需要作为节点之间的接口进行传输. 如果本文将调用节点的关系视为节点树, 那么显然 Sensor 树是节点树的子树. 根节点中有所有 Sensor. 随着 Sensor 的使用, 作为接口向下传输的 Sensor 越来越少. 最后, 当到达叶子节点时, Sensor 将不再转移到其节点的子节点.

算法 2 给出了 Sensor 的接口传输. 节点中, Sensor 的传输可以通过向其父节点添加 Sensor 来实现. 在分析每个节点或函数时, Sensor 被添加到其哈希表 M_{Sensor} 中. 开始时, 应标记叶子节点以表示其子节点没有 Sensor.

算法 2. Sensor 转化.

Input: Parent node N_P, Main node N_M, Sensor node *map* M_{Sensor}, Sub node *map* M_{Sub}, Sub node list L_{Sub}, Sensors of sub node list L_{SubS}, Sensors of parent node list L_{PrtS}, Sensors of new parent node list L_{NPrtS}.

1: *Initialized*: L_{Sub}:=*NULL*, L_{PrtS}:=*NULL*, L_{NPrtS}:=*NULL*
2: **if** ¬*isEmpty*(N_P) **then**
3: L_{Sub}←M_{Sub}.*get*(N_P)
4: **end if**
5: **for** *subN*∈L_{Sub} **do**
6: L_{SubS}←M_{Sensor}.*get*(*subN*)
7: **if** *isEmpty*(L_{SubS})∨0∉L_{SubS} **then**
8: *addSensor*(*subN*,N_M)
9: **end if**
10: L_{PrtS}←M_{Sensor}.*get*(N_P)
11: **if** ¬*isEmpty*(L_{PrtS}) **then**
12: **for** *s*∈L_{SubS} **do**
13: **if** *s*∉L_{PrtS}∧¬*isInteger*(*s*) **then**
14: L_{NPrtS}.*add*(*s*)
15: **end if**
16: **end for**
17: **if** ¬*isEqual*(*subN*,N_{Sub}) **then**
18: M_{Sensor}.*put*(N_P,L_{NPrtS})
19: **end if**
20: L_{NPrtS}:=*NULL*
21: **end if**
22: **end for**

该算法将父节点 N_P、主节点 N_M、Sensor 和节点的映射 M_{Sensor}、子节点的映射 M_{Sub}、子节点 L_{Sub} 列表、子节点的 Sensor 列表 L_{Sub}、父节点的 Sensor 列表 L_{PrtS} 和新父节点的 Sensor 列表 L_{NPrtS} 作为输入. 最初, 子节点 L_{Sub} 的列表、子节点 L_{Sub} 的 Sensor 列表和新父节点 L_{NPrtS} 的 Sensor 列表设置为空. 如果父节点不为空, 则将从映射 M_{Sub} 获得其子节点, 并将其分配到子节点列表 L_{Sub}.

在 SCADE 模型中, 节点或函数之间的调用关系可被视为一棵树. 因此, 本文将树从主节点向下遍历到叶子节点. 在找到叶子节点后, 本文向上将 Sensor 添加到其父节点. 在循环过程中, 本文将遍历子节点列表并实现子节点子网. 首先, 将子节点子网的 Sensor 实现为子节点列表 L_{Sub}. 如果 L_{SubS} 不为空或其元素不包含 0, 则将迭代调用该算法; 否则, 将从数据结构 M_{Sensor} 调用父节点 L_{PrtS} 中的 Sensor 列表. 如果 Sensor 列表 L_{PrtS} 不为空, 则算法将进入循环, 并实现列表 L_{Sub} 中的 Sensor.

如果 Sensor *s* 在父节点中不存在, 并且不是整数, 则将 *s* 添加到新父节点的 Sensor 列表 L_{NPrtS} 中. 循环后, 如果子节点 *subN* 不等于 N_{Sub}, 则列表 L_{NPrtS} 将被放入父节点 *NP* 的 Sensor 映射. 在外循环的最后, 新父节点的 Sensor 列表设置为空. Lustre 不定义 Sensor 或外部变量, 因此在合成 Lustre 程序时, Sensor 将作为全局常数输出. 算法 2 已将 Sensor 插入每个节点, 本文需要将 Sensor 插入其输入变量. 生成节点时, 将输出变量和输入变量列表中的 Sensor.

4.3 If Block的消去

SCADE 中的 if block 不仅可以支持模块开发, 而且可以提高完整性. 然而, Lustre 在其语法中没有引入 if block. 为了验证基于 Lustre 模型程序合成的 SCADE 模型, 必须将 if block 展开到每个方程. 同时, 还有许多

无用的方程. if block 的消除可分为解析过程、消除过程、存储过程和写入过程.

在解析过程中, 分别解析每个块. 特别是, if block 的嵌套需要分解. 解析 if block 的困难似乎是局部变量和方程. 每个块都有自己的局部变量. 通常, 一个块中的局部变量与另一个块中的局部变量有相同的名称, 因此, 局部变量必须更改其名称以相互区别. 理论上, 模 if block 可以无限嵌套. 然而, 如果系统模型设计有过多的 if block 模块, 则必须对其进行调整以提高可读性. 在不失一般性的情况下, 本文假设 if block 的嵌套应为 4 类: 第 1 类是 if block, 其结构为 if.then{…}else{…}; 第 2 类是 if block, 其结构为 if.then{…}else if.then{…}else{…}; 第 3 类可以是具有 if.then{if…then{…}else{·}}else{…}; 最后一类可以是具有 if.then{…}else{if…then{…}else{…}}. 如果 if block 的结构超过上述类别, 则解析过程类似.

在消除过程中, 由于局部变量的名称已扩展为增量名称, 因此可以在 if block 式算法先消除局部变量. 例如, then 模块中的局部变量 $\_L_5$ 被命名为 $\_L_5\_then$. 在 if block 中展开方程后, 本文已经获得了每个方程中的元素. 具体来说, 消除过程将从每个模块中解析的方程列表中添加方程. 但是, 如果一个等式只有赋值变量和调用参数, 则不会添加该等式. 此外, 如果在消除过程中未添加方程, 则将其存储在存储过程中. 未添加的等式意味着在传递中间值时将其消除.

在存储过程中, 为了区分 if block 的差异, 需要创建一个新的数据结构. 在表 3 中, 本文给出了在 if block 中存储每个方程的数据结构. 当本文设计具有 if block 的系统模式时, if block 伴随着一个索引. 索引用于识别等式出现的 if block 的顺序. 嵌套关系被命名为增量标识, 如 elseblockthen 和 thenthen. 第 1 列用于标记类别, 最后一列中的标签用于标记高阶算子是否出现在等式中. 这些局部变量和方程应逐个写, 而不是 if block.

表 3 存储 if block 的方程

类别	需要存储的元素										
	index	variable	index	element	index	element	index	element	index	element	label
1	1	V	1	then	1	else	–	–	–	–	–
2	1	V	1	then	1	else	1	elseblockthen	1	elseblockelse	–
3	1	V	1	then	1	else	2	thenthen	2	thenelse	–
4	1	V	1	then	1	else	2	elsethen	2	elseelse	–

改写过程包括两部分: 改写局部变量和改写方程. 在包含 if block 的节点中, 局部变量由节点中的局部变量、每个模块中的局部变量(如 then 模块、elseblockthen 模块等)组成. 后者的名称已修改, 以避免消除过程中出现重复别名. 在写过程中, 将根据表 3 中存储的 if block 方程合成 Lustre 程序的方程. 与局部变量类似, 要写的方程包括节点中的方程和每个模块中的方程. 每个方程存储在一个数组中. 在写公式时, 数组中的元素将被解析并以 Lustre 语法的格式写入.

最后, 写方程阶段将上述更新后的方程数据结构按照 Lustre 语言的格式进行输出, 这个阶段输出的方程经过了改写阶段的处理, 大大减少了方程个数. 例如, 一个简单的 if block 如图 2 所示.

```
1  # a, b, c are variables with int type
2  if cond1 then {
3      a=1;
4      b=5;
5  } else {
6      a=2;
7      c=7;
8      if cond2 then {
9          a=3;
10         c=8;
11     } else {
12         a=4;
13         b=6;
14     }
15 }
```

```
1 # a, b, c are variables with int type
a=if cond1 then 1
   else if (!cond1 and cond2) then 3
   else if (!cond1 and ! cond2) then 4
   else 2;
b=if (cond1) then 5
   else if (!cond1 and ! cond2)
   else default;
c=if (!cond1) then
      if (cond2) then 8
      else if (!cond2) then default
      else 7
   else default;
```

图 2 if block 的例子

在左侧展开 SCADE 中的 if block 后, 它存储在 8 个等式中. 它们是 *a_then*=1, *a_else*=2, *a_elsethen*=3,

a_elseelse=4, *b_then*=5, *b_elseelse*=6, *c_else*=7 和 *c_elsethen*=8. 最后, 这些方程存储为 Lustre 中的 3 个方程, 如图 2 右侧所示. 在 else 模块中修改了相同的返回值, 因此应消除 then 模块中的 *a*=2.

在 if block 中, 部分中间变量作为方程之间的传值, 这些不必要的中间变量也需要消除. 最关键的操作在于搜索过程, 对于一个方程 eq_1 的中间变量, 当搜索到方程 eq_2 中传值给该中间变量的常量或简单表达式时, 这个常量或简单表达式将会直接替换等式 eq_1 中的中间变量, 在更新方程的数据结构时, eq_2 将会被删除. 所以, 在写方程阶段将不会再输出方程 eq_2, 消除中间变量达到了消除多余方程的目的.

4.4 时态算子转化

SCADE 拓展了时态算子, 以表达某个变量或者某个操作需要随着时刻的改变进行改变. 在 SCADE 模型的设计过程中, 具有时态算子操作的赋值等式必须放在 node 里. SCADE 中含有初始时刻算子 Init、采样算子 When、前一时刻算子 Previous、延迟算子 Followed by、合并算子 Merge. 与其他算子表示计算某一时刻的数据流不同, 时态算子表达了每一时刻变量值的改变, 记录了模型状态的改变. 其中, Lustre 语言中定义了两个基本算子: 初始时刻算子、前一时刻算子. 对于采样算子 When, 可以通过 if-then-else 等式来表达, 即条件成立时触发事件, 不成立时赋默认值. 算法 3 给出 SCADE 中复杂的延迟算子的转化过程, 合并算子的转化可以根据这一算法进行推导得出.

算法 3. 延迟算子转化 *elimFollowedBy* (*fby*,V_A,Arr_E,L_E).

Input: Followed by operator *fby*, Assignment variables V_A, Equation array Arr_E, the list of equations L_E.

Output: Equation *eq*.

1: ***Initialized***: V_A:=*NULL*, Arr_E:=*NULL*, L_E:=*NULL*
2: *flow*←*parseFlow*(*fby*)
3: *delay*←*parseDelay*(*fby*)
4: *values*←*parseValues*(*fby*)
5: Arr_E←[*fby*,V_A,*flow*,*delay*,*values*]
6: L_E.*add*(Arr_E)
7: L_E←*elimin*(L_E)
8: An interation string *iterationStr*=*flow*
9: **for** (*i*∈ *delay*) **do**
10: A temporary string *tempStr*=*iterationEqStr*
11: *iterationEqStr*=*NULL*
12: *iterationEqStr*.*append*(*values*).*append*("→*pre*(").*append*(*tempEqStr*).*append*(")")
13: **end for**
14: *eq*.*append*(V_A).*append*("=").*append*(*iterationEqStr*).*append*(;)

算法 3 给出了延迟算子的转化方法, 包含解析阶段、消去阶段和输出阶段. 解析阶段根据算子名称读取赋值变量 V_A、输入流 *flow*、延迟时刻 *delay*、延迟期间 V_A 的取值 *values*. 解析后拼装成数组 Arr_E, 并被添加到方程列表 L_E 中. 消去阶段是将中间变量和多余的方程消去, 得到精简后的方程. 输出阶段是将消去阶段得到的方程列表按照 Lustre 的语法进行输出. 这里比较重要的是一个迭代字符串 *iterationStr*, 用来更新 *pre* 算子处理的变量.

图 3 给出了延迟算子转化过程的实例. 左边是解析后的 SCADE 代码, 右边是输出为 Lustre 代码的格式. 在 SCADE 建模时, 延迟时刻 *delay* 必须是一个整数, 这里假设为 3. *value* 是 *OutFlow* 跟随 *InFlow* 之前的取值. 更具体地, 假如 *InFlow* 在第 0–5 时刻内的取值为 i_0, i_1, i_2, i_3, i_4, i_5, 那么 *OutFlow* 在相应时刻的取值为 *value*, *value*, *value*, i_0, i_1, i_2. 对于合并算子 *Merge* 的等式 *OutFlow*=*merge*(*c*,$InFlow_1$,$InFlow_2$), 转化后的 Lustre 代码为一个 if-then-else 语句: *OutFlow*=if *c* then $InFlow_1$ else $InFlow_2$. 可以根据延迟算子的转化进行实现.

```
1  _L1=InFlow
2  _L2=value                #假设延迟时刻delay为3
3  OutFlow=_L3              OutFlow=value→pre(value→pre(value→pre(InFlow)))
4  _L3=fby(_L1,delay,_L2)
```

图 3 延迟算子的转化

4.5 状态机的转化

在 SCADE 中, 状态机由状态、转换和分层状态机组成. 每个状态对应一个瞬间. 状态由一系列方程组成, 这些方程可以分配给状态变量、信号变量和输出变量. 两个状态之间的每个转换都包含保护和操作. 保护可以是信号的触发器、时钟的到达或布尔表达式的满足. 该操作通常使用关键字 emit 更新信号. 转换与时钟相关. 也就是说, 跃迁分为强跃迁和弱跃迁, 这意味着处于目标状态的物体分别在下一个周期和当前周期重置.

算法 4 将状态机和状态到信号的映射作为输入, 并返回状态机中的方程列表. 该算法分为全局状态、全局信号和输出变量的过程. 每个子过程返回一个方程列表, 这些方程被添加到算法的方程总和中. 全局状态的转换遍历状态列表. 在实现每个状态的转换后, 它进入转换循环. 在每个转换中, 将分析其条件和效果. 创建一个状态和条件数组, 并将其添加到列表 L_{stSig} 中, 该列表放入状态映射中, 以发送信号 M_{stSig}. 在全局信号子过程中, 状态和信号列表 L_{stSig} 将从映射 M_{stSig} 中获得. 如果列表 L_{stSig} 不是空的, 则将根据 Lustre 的语法遍历并输出它. 对于处于每个状态的方程, 本文忽略了方程解析的呈现, 并立即提供解析后的方程列表作为结果. 从状态列表和状态到方程的映射中写入方程后, 添加并返回所有方程的列表.

算法 4. 状态机的转化 *transfSM*(*stateMachine*,M_{stSig}).

Input: State machine *stateMachine*, the *map* of states to signal M_{stSig}.

Output: A list of state machine equations L_{smeq}.

1: ***Initialized***: L_{smeq}:=*NULL*, M_{stSig}:=*NULL*
2: A list of states and signals L_{st}, L_{sig}←*stateMachine*
3: The equation of states eq_{st}:=*NULL*
4: **for** $s\in L_{st}$ **do**
5: **if** *isInitial*(*s*) **then**
6: eq_{st}.*append*(*s*).*append*(“→”)
7: **end if**
8: The list of transitions L_{tr}←*s.transition*
9: **for** $tr\in L_{tr}$ **do**
10: The condition *cond*←*tr.condition*
11: The list of state signals: L_{stSig}:=*NULL*
12: The condition of state c_{st}←⟨*s*,*cond*⟩
13: L_{stSig}.*add*(c_{st})
14: The effect *eff*←*tr.effect*
15: M_{stSig}.*put*(*eff*,L_{stSig})
16: Target state s_{tar}←*tr.targetState*
17: eq_{st}.*append*(“if(”+*cond*+“and (*pre s*=”+*s*+“)) then”+s_{tar}.
18: **if** not finished **then**
19: eq_{st}.*append*(“else”)
20: **end if**
21: **end for**
22: **end for**
23: L_{smeq}.*add*(eq_{st})

24: The equation of signals eq_{sig}:=*NULL*
25: **for** $sig \in L_{sig}$ **do**
26: $L_{stSig}=M_{stSig}.get(sig)$
27: **if** $\neg isEmpty(L_{stSig})$ **then**
28: $eq_{sig}.append(sig).append(\text{“=if (”})$
29: **for** $c_{st} \in L_{stSig}$ **do**
30: $eq_{sig}.append(\text{“}State\text{=”}+c_{st}[0]+\text{“and”})$
31: $eq_{sig}.append(c_{st}[1]+\text{“) then true”})$
32: $default=signal.getDefaultValue(signal.type)$
33: $eq_{sig}.append(\text{“else”}+default)$
34: **end for**
35: **end if**
36: **end for**
37: $L_{smeq}.add(eq_{sig})$
38: A *map* of state equations M_{steq}:=*NULL*
39: A list of equation L_{eq}:=*NULL*
40: **for** $s \in L_{st}$ **do**
41: Parsed equation list $eqL \leftarrow s.equation$
42: **for** $eq \in eqL$ **do**
43: $L_{eq}.add(parseStateEq(eq,s))$
44: **end for**
45: $M_{steq}.put(s,L_{eq})$
46: **end for**
47: $L_{smeq}.add(writeStateMachineEq(L_{st},M_{steq}))$
48: **return** L_{smeq}

4.6 高阶算子的消去

SCADE 引入了高阶算子来表示循环和迭代运算, 但 Lustre 在其方程中缺少这些运算. 因此, 有必要将 SCADE 模型中的复杂高阶运算展开为单个方程的多次执行. 算法 5 介绍了高阶算子的消除, 它分为 4 个过程: 解析过程、修改过程、存储过程和写过程.

算法 5. 高阶算子消去 $elimHO(V_A,V_P,L_{LV},L_E,L_D,itrV,ind,C,iter)$.

Input: Assignment variables V_A, Call parameters V_P, index of if block *ind*, the list of local variables L_{LV}, operator *op*, iterator *iter*, if condition *C*, the list of equations L_E, the list of default value L_D, iteration value *itrV*.

1: ***Initialized***: the operator with parameters *opp*:=*NULL*, the array of equations Arr_E:=*NULL*
2: **if** $isEqual(iter,IteratorOp)$ **then**
3: $opp=parseIter(V_A,V_P,ind,L_{LV},op)$
4: **else if** $isEqual(iter,PartialIteratorOp)$ **then**
5: $opp=parsePIter(V_A,V_P,ind,C,L_{LV},op)$
6: **end if**
7: **for** $i \in itrV$ **do**
8: **if** $isEqual(iter,IteratorOp)$ **then**
9: $modifyIter(V_A,V_P,ind,L_E,i,L_{LV},opp)$
10: $Arr_E=storeIter(V_A,V_P,L_E,Arr_E,L_{LV})$

```
11:       writeIter(Arr_E,ind)
12:    else if isEqual(iter,PartialIteratorOp) then
13:       modifyPIter(V_A,V_P,ind,L_E,L_D,C,i,L_LV,opp)
14:       Arr_E=storePIter(V_A,V_P,L_E,Arr_E,L_LV)
15:       writePIter(Arr_E,ind)
16:    end if
17: end for
```

在解析过程中, 本文将首先获取高阶算子的名称和实例参数, 然后获取其调用参数和赋值变量. 本文将分析不同的运算符. 对于迭代器 *op* 算子, 即 *map* 和 *mapfold* 算子, 解析迭代器算子将赋值变量 V_A、调用参数 V_P、if block 索引、局部变量 L_{LV} 列表和算子 *op* 作为输入; 对于具有参数的迭代器 opp 算子, 将实现 if 条件及其迭代索引.

在修改过程中, 语言 Lustre 不支持迭代执行, 因此每次修改高阶算子都需要迭代索引来展开高阶运算的执行. 对于高阶运算, 除了每次迭代的迭代器、索引和条件外, 其输入和输出是相同的维度. 它将赋值变量 L_V、调用参数 V_P、if block 的索引、方程列表 L_E、每次迭代 i 的索引、局部变量列表 L_{LV} 和运算符 *op* 作为输入, 使用 *IteratorOp* 运算符修改方程. 此外, 修改 *PartialIteratorOp* 需要条件 C 和默认值 L_D 列表作为每次迭代的输入. 该过程修改 SCADE 语法, 并根据其输入合成 Lustre 模型或程序.

存储过程从赋值变量 V_A、调用参数 V_P、方程列表 L_E、方程的空数组和局部变量列表 L_{LV} 中对方程数组 Arr_E 进行更新. 首先, 该过程将判断 if block 中局部变量所出现的结构模块. 在这里, 模块意味着局部变量列表可以出现在 if block 的位置, 如 then 模块、elseblockthen 模块等. 表 3 对出现的模块进行了明确描述. 简洁的存储方法是 if block 中高阶运算的特定结构的结果. 更重要的是, 当 if block 的嵌套比表 3 中的定义更复杂时, 可以扩展用于存储方程元素的数据结构, 但是出现更复杂的 if block 嵌套时, 往往需要检查功能模型的设计问题.

写入过程将赋值变量 V_A 和方程 Arr_E 的数组作为输入, 并按照 Lustre 语法写入方程. 具体来说, 写入过程从数组的元素中获取值. 在存储过程之后, 数组包括 if 条件、默认值和每个模块中的元素. 如果不同模块中的方程具有相同的赋值变量, 则它们将合并为一个方程, 并作为 if-then-else 语句的嵌套写入. 此外, 赋值变量成对出现. 与清单 1 中的代码类似, 变量 b=3 的赋值只出现在 then 模块的等式中, 因此它必须由 else 模块中的默认值组成. 值 3 可以替换为高阶表达式.

4.7 转化方法的评价

值得注意的是, 本文采用 Jkind 模型验证器[19]中的 Lustre 版本, 与 Verimag 实验室[21]的版本略有不同. 例如, 数组迭代器和时态操作(如 when、fby 和 merge)没有在 Jkind 版本的 Lustre 中引入, 而在 Verimag 版本中进行了定义. SCADE 从早期的 Lustre 版本进行了演化, 虽然没有沿着 Verimag 实验室 Lustre 的方向进行发展, 但是也借鉴了目前流行语言的特色. 比如, 从函数式编程语言引入的高阶迭代使得 SCADE 可以表达循环, 从面向对象语言引入的泛型使得采用 SCADE 构建的底层函数可以得到复用. 引入这些特色虽然增加了 SCADE 语言的表达能力, 但是需要增加对这些操作的转换. 此外, 为了呈现复杂操作和分层结构的转换, 本文忽略了对类型、常量、数学操作、比较操作、逻辑操作、位操作这些简单转换过程的介绍, 这部分操作在 Lustre 和 SCADE 的语言规范中的定义是相同的, 转化过程不需要做改变. 对于状态机的消除, 本方法的转化方法忽略了层次结构的表示(可以按照消除 if block 的方式进行)和强弱迁移的区分. 本文致力于解决对工业级大规模复杂系统进行形式化验证中遇到的挑战.

基于上述转化算法, 尽管合成的 Lustre 模型与 SCADE 模型在语法上有差别, 但是在语义上是一致的. 本文忽略一致性的证明, 从 3 个角度来说明语义的一致性. 迭代体中有一类是部分迭代体算子 *PartialIterator*, 当条件不满足时, 这类迭代体的执行将会结束, 输出变量被赋默认值. 转换算法的实现展开这一迭代体为顺序执行的方式, 在 Lustre 模型中, 每次进入下一次迭代执行时都会进行条件的判断, 只要条件不满足, 就会赋

默认值, 并且进入下次迭代体判断条件被置为 false, 这就使得后续的所有输出变量被置为默认值. 进而与 SCADE 模型的执行顺序、次数和结果保持一致. 另外一个角度是根据索引读数组元素的操作, 在 SCADE 中, 一个 *projection* 算子即可完成, 但是在 Lustre 中需要写成 if-then-else 结构的性质, 每个 if 判断索引值是否等于当前的值, 一直判断到等于数组长度的次数. 任何一次匹配都会执行结束, 这与 SCADE 的 *projection* 操作是一致的. 此外, 由于 SCADE 引入了 if block, 即根据条件判断某一模型块的执行, 不满足条件的模型块不会执行. 转化后的 Lustre 代码会根据每个模型的进入条件, 将 SCADE 的 if block 进行展开为多个 if-then-else 等式. 在 Lustre 代码执行的过程中, 需要对每一条 if-then-else 等式进行判断, 进而根据条件满足情况返回相应的结果. 这使得 Lustre 模型中的 if-then-else 等式在执行次数和执行结果上与展开前的 SCADE 模型 if block 都是一致的. 其他转化算法的等价性可以通过类似的分析得到. 此外, 需要注意的是, 中间变量消去算法的影响, 中间变量消去减少了数据流的长度, 使输入变量更早地被使用, 更快地得到输出变量, 这只是缩短了执行路径, 对于执行结果没有改变.

另外, 以形式化方法解决工业系统实际问题为目标, 本文忽略了对算法复杂度的分析, 因为运行转化过程消耗的时间相对于形式化验证过程可忽略不计. 以本文区域控制器子系统的反向跳跃功能为例, 数秒内即可完成从 SCADE 模型到 Lustre 模型的转化过程, 而对合成后的 Lustre 模型进行验证的时间却往往要长很多.

5 实验分析

为了验证安全关键系统的模型, 本文首先给出了 CBTC 区域控制器子系统软件模型的验证方法和工具原型; 然后, 对本文提出的方法进行了实验; 最后, 对本文方法的有效性进行了评估.

5.1 方法和工具实现

本文在图 4 中提出了验证 CBTC[17]中区域控制器子系统安全性的方法. 该方法将安全需求、功能需求规范和线路地图作为输入.

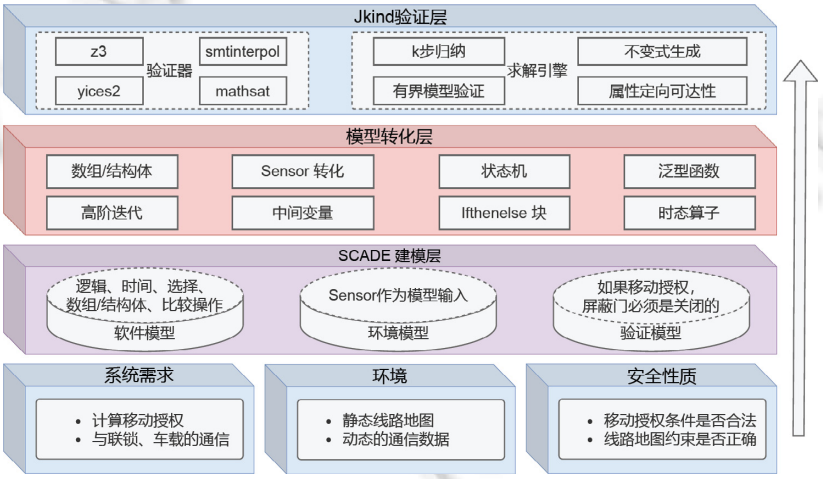


图 4 SCADE 软件模型的形式化验证框架

在建模阶段, 本文将安全需求分为 3 类: 环境、功能和不变式. 环境安全意味着线路地图必须满足模型的约束: 一方面, 线路地图必须满足特定约束; 另一方面, ZC 子系统从线路地图中读取 Sensor, 以计算移动授权的条件. 此外, ZC 子系统定义了列车包络和缓冲区 BZ, 以提高运行列车的安全性. 功能需求规范产生了 ZC 子系统的 SCADE 模型. 不变式意味着移动授权在每个时刻的状态必须满足相应的约束.

SCADE 模型的输入来自 Sensor, 其中包括来自线路地图的数据以及与其他子系统的通信数据. 一旦建立 SCADE 模型, 则 Lustre 模型将通过特定算法合成, 如接口传输、数组或结构转换、高阶算子消除、泛型函数

消除、状态机转换等. 根据上述合成的 Lustre 模型, 验证步骤在 Jkind 上执行. 基于线路地图中的 Sensor, 如果系统 Lustre 模型的输入和输出满足安全 Lustre 模型, 则检查器返回 Valid[22,23]; 否则, Jkind 将提示 Invalid, 并返回反例列表.

给定 ZC 子系统的 SCADE 模型 M 及其执行环境 E, 安全性φ的形式化验证问题可以定义为$(E,M)\models\varphi$. 本文使用观察器 Observer 对安全性质进行建模. 观察器是一种特殊的 SCADE 模型, 合成到 Lustre 模型后, 将其命名为 Lustre 安全模型 $L(\varphi)$. 而系统 Lustre 模型可以表示为 $L(M)$. 环境输入, 作为每周期模型 $L(M)$执行的输入, 由 3 个部分组成: 来自线路地图的静态数据; 其他子系统发送过来的通信数据; 上周期执行的结果, 被转换为 Lustre 模型 $L(E)$. 因此, SCADE 模型的形式化验证问题可以重写为$(L(E),L(M))\models L(\varphi)$, 即系统模型 $L(M)$在给定环境输入 $L(E)$下满足安全性质 $L(\varphi)$.

基于上述转换算法, 本文实现了一个名为 StoL 的合成器工具原型, 用于从 SCADE 模型合成 Lustre 模型. 该实现是用 JDK 1.8 版本的 Java 编程语言开发的. StoL 将 SCADE 文件的扩展名 xscade 作为输入, 并生成 lus 扩展名的 Lustre 模型. 开始时, StoL 定义根节点, 以指示 StoL 应在何处解析 SCADE 模型. 在合成的开始, StoL 将解析 SCADE 模型的类型和常数, 以便在解析节点或函数时可以匹配变量的类型.

5.2 实验实施

上述方法是基于卡斯柯信号有限公司的区域控制子系统(ZC 子系统)进行描述, 本文的实验也在该 ZC 子系统上实施. 为了评估区域控制子系统软件 SCADE 模型的复杂性, 本文采用反向跳跃功能对 SCADE 模型规模进行量化, 这是区域控制器子系统中相对复杂的功能. 一旦列车驶出反向跳跃区, 列车需要运行一定距离才能停车. 因此, 对于可以授权列车点动的反向跳跃区域, 如果列车需要反向跳跃并应向 ZC 发送反向跳跃请求, 则 ZC 应在发车授权前检查从该区域起点沿反向跳跃方向是否有有效的退行区域: 如果是, 则联锁子系统将锁定该区域, 并阻止后方列车进入反向跳跃区域的授权请求.

可以为相同的需求构建各种模型. 对于反向跳跃, SCADE 模型中包含许多通用函数库来操作数组和结构体类型的基本数据. 同时, 为了计算具有嵌套数组和结构体的复杂数据的 Sensor 或变量, 本文采用高阶迭代来进行功能的建模, 其中, 迭代参数可以是区段的最大值、移动授权区的最大值、道岔的最大值、警冲区的最大值、屏蔽门的最大值、移动授权覆盖区的最大数量等. 例如, 创建名为 *g_arrsTrain_LocMaxBrchPoss* 的 Sensor 是为了将一个 ZC 区域内所有列车的位置信息存储起来. 它由每个列车的最大头和最大尾坐标所在的分支索引和分支坐标偏移量构成.

当 SCADE 模型中的一个方程通过调用另一个节点或函数来计算其结果时, 节点或函数的嵌套形成一棵调用树. 当方程是高阶算子时, 它将迭代执行节点或函数. 此外, 如果具有高阶算子的方程调用的节点或函数调用其他函数, 则使用迭代器执行高阶算子将以指数形式生成变量和方程. 在 Jkind 执行过程中, 这些变量和方程将被翻译为局部变量或方程, 以便 SMT 求解器计算.

反向跳跃的 SCADE 模型中有 341 个节点和函数, 在这些节点和函数之间建立了 46 个高阶迭代体函数. 从根节点到叶子函数的最深层级为 24. 为了进行实验, 本文在 Jkind v4.5.1 中部署了求解器, 分别是 z3 v4.8.13、mathsat v5.6.6、yices v2.6.4. 另外, Jkind 自身集成了 smtinterpol 求解器. 在 Jkind 配置文件中, 本文为不同的机器设置了尽可能大的最大堆栈空间. 同时, 本文没有致力于在大内存(如成千上百 GB)机器上的验证, 而是在正常可接触到的内存上进行实验.

Jkind 运行两种方式的执行方式: 命令行执行和源码执行. 命令行执行可以指定求解器(如 z3、smtinterpol、yices2、mathsat 等)、求解策略(有界模型检查、不变成生成、*k*-induction、pdr、切片等)和输出方式等. 源码执行允许通过断点对模型逐步执行的状态进行分析, 尤其是其翻译阶段, 可以根据 Lustre 模型翻译的局部变量和方程个数判断模型的规模.

5.3 实验结果

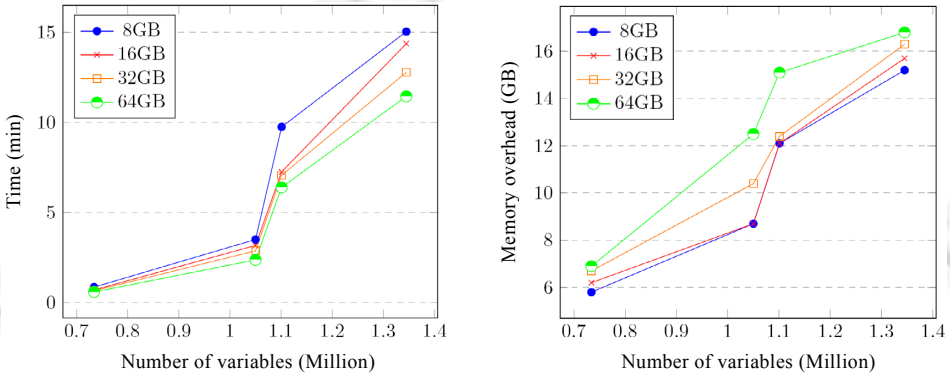
在模型检查的执行过程中, 高阶算子的迭代参数将影响存储在 Jkind 翻译阶段的变量. 然后, 这会影响求

解阶段中状态空间的大小. 反向跳跃的这些参数对形式化验证的内存开销和执行时间有很大影响. 表 4 显示了 4 台不同内存的 Windows 操作系统机器的最大参数结果. 第 1 行显示 4 台机器的不同内存. 第 1 列介绍了区域控制子系统中反向跳跃的迭代体参数. 其中, *MAX_BLOCK*、*MAX_POINT*、*MAX_DEPOT_GATE* 和 *MAX_TRAIN* 是具体的物理设备, 其他参数是为了保证列车安全所抽象出的轨道逻辑区域.

表 4 不同内存机器对应的迭代体参数值

	8 GB	16 GB	32 GB	64 GB
MAX_BLOCK	20	30	30	40
MAX_MA_ZONE	10	20	20	30
MAX_POINT	5	20	15	20
MAX_TRAIN	10	10	10	15
MAX_DEPOT_GATE	10	10	10	12
MAX_MAOV_ZONE	8	9	10	12
MAX_FOULZONE	8	9	10	12
MAX_ESB	8	9	10	12
MAX_PSD_ZONE	8	9	10	12

图 5 给出了在不同内容机器上执行形式化验证过程时, 不同的变量个数对验证时间和内存开销的影响. 图 5(a)表示当变量个数较少时, 在各个机器上的验证时间影响不大; 当变量个数达到 10 万级时, 验证时间对变量个数的影响较大, 尤其是相对小内存的机器更为敏感. 图 5(b)表示变量个数对内存的影响. 大内存的机器可以提供更多的堆栈空间用于存储求解过程的变量, 尤其是循环程序中的变量, 这也进一步加快了验证过程的求解.

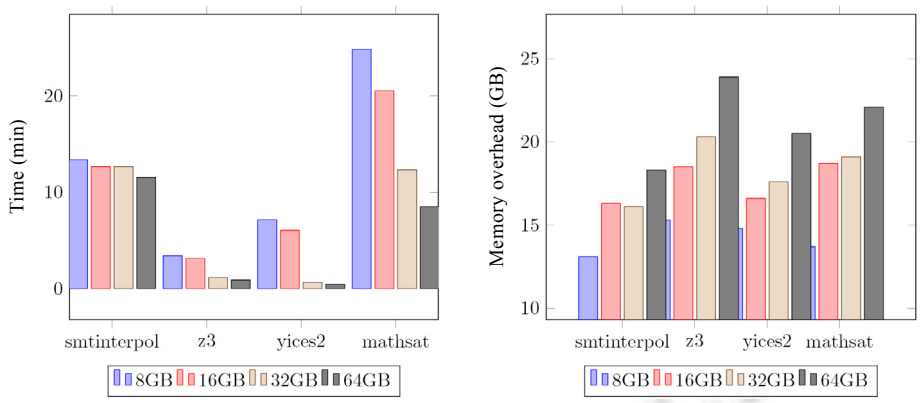


(a) 随着变量个数改变的时间消耗 (b) 随着变量个数改变的内存开销

图 5 随着不同机器上变量个数的变化, 消耗的时间和内存开销的变化情况

值得注意的是, 本文对系统规模的评价是借助于 Jkind 工具的翻译器进行的. 在 Jkind 的内部, Lustre 代码经过解析之后会首先翻译为 L 语言, 然后再通过并行求解器进行求解. 翻译为 L 语言的变量个数可以更加客观地衡量 ZC 子系统系统的规模, 也能客观地评价本方法的优势所在, 更重要的是, 可以客观地评价目前形式化验证在工业控制领域的应用上限. 图 5 中的变量个数是以 Jkind 翻译器翻译之后的变量个数.

图 6 给出了不同求解器在不同内存的机器上执行反向跳跃功能模型的验证过程中的时间消耗和内存开销. 图 6(a)表示 smtinterpol 求解器在不同的机器上消耗时间差别不大; mathsat 求解器的验证时间对机器内存特别敏感; z3 和 yices2 是相对高效的求解器, 尤其是 Z3 在不同内存的机器上的表现都比较稳定. 图 6(b)表示机器内存是影响验证过程内存开销的主要因素, 这解释了较大内存的机器可以提供更多的堆栈空间. 而且相对而言, Z3 比其他求解器在不同机器上消耗的内存都多, 对于验证极限以内的模型, 可以更快地获得验证结果. 而 smtinterpol 求解较慢, 但是其消耗的内存较小, 当接近于验证极限时, 它可以求解更大规模的模型.



(a) 不同求解器的时间消耗　　(b) 不同求解器的内存开销

图 6　不同求解器的时间消耗和内存开销

5.4 实验分析与评价

本文的验证过程是以 ZC 子系统的 SCADE 模型、线路地图和安全性质作为输入, 通过大量的算法实现过程, 将其转化为 Lustre 模型, 并对 Lustre 模型进行形式化验证. 对 Lustre 模型的规模量化可以分为以下情况进行分析.

- 第一, 因为 Jkind 对 Lustre 语法有要求, 但是对格式要求不严格, 而且 Lustre 模型的规模(即代码量)还取决于输出格式. 由于 Lustre 不支持变量作为索引, 因此数组的运算需要将数组的索引根据数组长度进行展开. 比如, Lustre 不支持数组索引为变量, 为了读一个数组, 不得不写成: “if (*i*=0) then *a*[0] else if (*i*=1) then *a*[1] else if (*i*=2) then *a*[2] else ...”, 为了可读性更强, 本文在转化过程中输出格式为:
  if (*i*=0) **then** *a*[0] **else**
  if (*i*=1) **then** *a*[1] **else**
  if (*i*=2) **then** *a*[2] **else**
  ...
  这一小小的不同, 对代码行数的影响是非常大的, 因为 ZC 子系统中的数组往往都是以百和千为单位.
- 第二, 在反向跳跃中, 本文提取线路地图的数据. 其中, 许多数据是通过数组和结构类型的嵌套构建的. 为了处理这些数据, 必须使用一些高阶迭代器(如 *map*、*mapfold*、*mapwi*、*foldwi* 等)构建了反向跳跃的 SCADE 模型. 高阶迭代的每次执行都会出现丰富的变量和方程. 此外, SCADE 模型中节点或函数的数量与模型规模不匹配. 也就是说, 一旦通过迭代创建节点, 模型比例将随着参数大小与变量和方程数量的乘积而增加. 节点或函数中出现的迭代器越多, 模型比例指数增长越快.
- 第三, 随着产品的迭代, ZC 子系统的功能在不断增加, 这导致了 ZC 子系统的不同版本. ZC 需求变更的过程也引起了 SCADE 模型的变化, 进而引起 Lustre 模型的变化.
- 第四, 由于轨道交通遵循“安全-故障”原则, 必须考虑物理环境可能的故障. 以列控系统为例, 必须要考虑通信故障、设备故障带来的系统失效问题, 这也使得系统设计时的复杂程度会比较高. 在 SCADE 模型中, 需要增加较多故障或功能失效场景的处理, 这部分故障和失效处理在 SCADE 模型中占据了一定比例.
- 第五, 考虑时间操作的存在, 形式化验证过程中的变量个数为某一时刻的变量数乘以执行时刻数.

基于以上的分析, 本文在某版本 ZC 子系统建立的 SCADE 模型所转化的 Lustre 模型的代码行数约为 192 万行. 对于具体 ZC 子系统某个功能的 SCADE 模型, 迭代器出现的次数、参数值的大小和内存大小是评估 ZC 子系统的模型规模和复杂性的 3 个因素. 而且, SCADE 对功能需求的实现太难更改, 这导致几乎不可能修改迭代器的出现时间. 参数和内存成为判断 ZC 子系统复杂性的基本因素. 此外,参数的大小与系统的性能成比

例. 从 ZC 子系统规模量化的角度, 对反向跳跃功能的形式化验证可分为功能的验证和性能的验证, 为了评估 ZC 子系统的模型规模, 本文保留了完整功能的验证, 通过减小迭代体参数弱化了对性能的验证. 具体体现是: 迭代体造成了变量个数的急剧增加, 超出了验证器的上限.

6 总 结

本文引入了一种新方法, 通过程序转换影响形式化验证中的改变, 从模型、验证器、机器内存等角度探索验证能力的上限. 本文的方法在SCADE模型中融合了环境和安全需求. 该转换用于转换安全状态机和数据流模型, 特别是高阶迭代和泛型函数. 本文观察到, 验证工程师在转化和验证过程中几乎没有决定权, 但程序转换为利用对验证过程的影响提供了一种基本方法. 为此, 本文证明了验证实际系统需要借助于环境输入和数据流转换, 而这弥补了状态机模型转换的不足. 本文提出了形式化验证的行业级研究, 在效率较高的验证器和大规模实际工业系统上进行了评估. 事实上, 由于存在众所周知的状态空间爆炸问题, 当前参数不足以满足 ZC 软件交付的性能. 长期工作将从模型分解与组合、模型抽象、验证算法改进和机器内存提升的角度来减少状态空间.

References:

[1] Shi J, Shi JQ, Huang YH, *et al*. Scade2Nu: A tool for verifying safety requirements of SCADE models with temporal specifications. In: Joint Proc. of the 25th Int'l Conf. on Requirements Engineering: Foundation for Software Quality (REFSQ 2019). 2019. 1−6.

[2] Heim S, Xavier D, Bonnafous E, *et al*. Model checking of SCADE designed systems. In: Proc. of the 8th European Congress on Embedded Real Time Software and Systems (ERTS 2016). 2016. 1−8.

[3] Yuan ZH, Chen XH, Liu J, *et al*. Simplifying the formal verification of safety requirements in zone controllers through problem frames and constraint-based projection. IEEE Trans. on Intelligent Transportation Systems, 2018, 19(11): 3517−3528.

[4] Caspi P, Pilaud D, Halbwachs N, *et al*. LUSTRE: A declarative language for real-time programming. In: Proc. of the ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages. ACM, 1987. 178−188.

[5] Nicolas H, Paul C, Pascal R, *et al*. The synchronous data flow programming language LUSTRE. Proc. of the IEEE, 1991, 70(9): 1305−1320.

[6] Ran D, Chen Z, Sun Y, *et al*. SCADE model checking based on program transformation. Computer Science, 2021, 48(12): 125−130.

[7] Vassil T, Safouan T, Frederic B, *et al*. Improved invariant generation for industrial software model checking of time properties. In: Proc. of the IEEE 19th Int'l Conf. on Software Quality, Reliability and Security (QRS). IEEE, 2019. 334−341.

[8] Aniculaesei A, Vorwald A, Rausch A. Using the SCADE toolchain to generate requirements-based test cases for an adaptive cruise control system. In: Proc. of the 22nd ACM/IEEE Int'l Conf. on Model Driven Engineering Languages and Systems Companion (MODELS-C). 2019. 503−513. [doi: 10.1109/MODELS-C.2019.00079]

[9] Henning B, Henning G, Michaela H, *et al*. An open alternative for SMT-based verification of SCADE models. In: Proc. of the 19th Int'l Conf. on Formal Methods for Industrial Critical Systems (FMICS). Florence: Springer, 2014. 124−139.

[10] Zhou JM. Formal modeling and verification of SCADE model based on PVS for rail transit control system [MS. Thesis]. Shanghai: East China Normal University, 2011 (in Chinese with English abstract).

[11] Bennour IE. Formal verification of timed synchronous dataflow graphs using Lustre. Journal of Logical and Algebraic Methods in Programming, 2021, 121(100678): 1−24.

[12] Henning G, Stefan M, Oliver M. On the formal verification of systems of synchronous software components. In: Proc. of the 31st Int'l Conf. on Computer Safety, Reliability, and Security (SAFECOMP 2012). Magdeburg: Springer, 2012. 291−304.

[13] Milton S, Siddhartha B, Matthew AC, *et al*. Formal compositional reasoning of autonomous aerial systems with complex algorithms. In: Proc. of the IEEE Int'l Systems Conf. (SysCon). IEEE, 2020. 1−7.

[14] Ferrari A, Mazzanti F, Basile D, *et al*. Systematic evaluation and usability analysis of formal methods tools for railway signaling system design. IEEE Trans. on Software Engineering, 2022, 48(11): 4675−4691.

[15] Liu XS, Yuan ZH, Chen XH, *et al*. Safety requirements modeling and automatic verification for zone controllers. Ruan Jian Xue Bao/Journal of Software, 2020, 31(5): 1374−1391 (in Chinese with English abstract). http://www.jos.org. cn/1000-9825/5952.htm [doi: 10.13328/j.cnki.jos.005952]

[16] Blanc B, Bandmann O, Fredholm D. 2018. https://www.prover.com/https://www.prover.com/webinar/building-a-safety-case-for-rail-control-software-with-formal- verification/

[17] Li TF, Chen XH, Sun HY, *et al*. Modeling and verification of spatiotemporal intelligent transportation systems. In: Proc. of the 19th IEEE Int'l Conf. on Trust, Security and Privacy in Computing and Communications (TrustCom). IEEE, 2020. 568−575.

[18] Dumitru PB, Stephen AE, Gerard B. Compiling Esterel. Springer, 2007.

[19] Andrew G, John B, Mike W, *et al*. The JKind model checker. In: Proc. of the 30th Int'l Conf. on Computer Aided Verification (CAV). Springer, 2018. 20−27.

[20] EN 50126 Railway Applications. The specification and demonstration of reliability, availability, maintainability and safety (RAMS). In: Proc. of the Systems Approach to Safety. 2017. 11−25.

[21] Erwan J, Pascal R, Nicolas H. The Lustre V6 Reference Manual. Verimag Lab, 2017. https://www-verimag.imag.fr/Lustre-V6.html

[22] Amar B, Bernard D. Formal verification for model-based development. Journal of Passenger Cars: Electronic and Electrical Systems, 2005, 114: 171−181.

[23] Martin L, Schatalov M, Hagner M, *et al*. A methodology for model-based development and automated verification of software for aerospace systems. In: Proc. of the IEEE Aerospace Conf. IEEE, 2013. 1−19.

附中文参考文献:

[10] 周佳铭. 基于 PVS 对 SCADE 开发轨交控制系统的形式化建模与验证[硕士学位论文]. 上海: 华东师范大学, 2011.

[15] 刘筱珊, 袁正恒, 陈小红, 等. 区域控制器的安全需求建模与自动验证. 软件学报, 2020, 31(5): 1374−1391. http://www.jos.org. cn/1000-9825/5952.htm [doi: 10.13328/j.cnki.jos.005952]

李腾飞(1989－), 男, 博士, 工程师, CCF 专业会员, 主要研究领域为形式化方法, 工业控制系统安全.

孙军峰(1971－), 男, 正高级工程师,主要研究领域为安全软件构造, 列控系统安全保证方法.

吕新军(1978－), 男, 正高级工程师,主要研究领域为安全软件, 安全系统.

陈祥(1981－), 男, 高级工程师, 主要研究领域为列车控制系统, 模型驱动开发.

刘静(1964－), 女, 博士, 教授, 博士生导师, CCF 专业会员, 主要研究领域为形式化方法, 高可信理论和方法.

孙海英(1976－), 女, 博士, 讲师, CCF 专业会员, 主要研究领域为形式化建模与验证, 基于形式化的测试.

何积丰(1943－), 男, 博士, 教授, 博士生导师, CCF 会士,主要研究领域为高可信计算, 信息物理融合系统, 可信人工智能.