软件学报 ISSN 1000-9825, CODEN RUXUEW
2026,37(9):3521−3556 [doi: 10.13328/j.cnki.jos.007605] [CSTR: 32375.14.jos.007605]

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

从设计到安全分析: 异构模型转换与交叉验证*

吴梦丹[1], 杨顺昆[1], 侯展意[1], 佘志坤[2], 曾福萍[1], 冀振燕[3]

[1](北京航空航天大学 可靠性与系统工程学院, 北京 100191)

[2](北京航空航天大学 数学科学学院, 北京 100191)

[3](北京交通大学 网络空间安全学院, 北京 100044)

通信作者: 杨顺昆, E-mail: ysk@buaa.edu.cn

摘　要: 在复杂软件系统开发过程中, 设计阶段与验证阶段的有效衔接是确保系统可靠性和功能正确性的关键. 然而, 设计工具与验证工具在建模语言、语义和数据结构上的异构性, 易导致模型转换语义不一致、工具链互操作性不足以及验证覆盖不充分. 为解决上述挑战, 提出一种基于统一中间表示的分层转换与多重交叉验证机制. 该机制以设计模型为起点, 结合系统理论过程分析 (systems-theoretic process analysis, STPA) 开展危险分析与不安全控制行为识别, 提炼安全约束, 并与既有的功能、时间和安全属性进行互补校验. 随后, 构建设计模型到统一中间表示 (unified intermediate representation, UIR) 的映射, 在统一语法与语义域中形式化描述结构、行为、时序与安全约束, 并据此给出覆盖上述要素的分层转换规则及其追溯性元数据. 基于 UIR, 派生时序模型、逻辑模型与概率模型等互补的验证模型, 并将 STPA 导出的安全约束系统化映射为可验证属性, 开展源-中间-目标模型的一致性检查、时序与逻辑性验证、概率性分析以及安全约束可满足性验证等多视角交叉验证. 以自动驾驶汽车控制系统以及多域协同无人机控制系统为例的实际案例结果表明, 所提方法提高了验证的覆盖度与准确性, 增强了转换与验证过程的可追溯性, 为复杂系统的安全驱动开发提供了一条一致且可证的技术路径.

关键词: 安全风险分析; 统一中间表示; 模型转换; 多重交叉验证

中图法分类号: TP311

中文引用格式: 吴梦丹, 杨顺昆, 侯展意, 佘志坤, 曾福萍, 冀振燕. 从设计到安全分析: 异构模型转换与交叉验证. 软件学报, 2026, 37(9): 3521–3556. http://www.jos.org.cn/1000-9825/7605.htm

英文引用格式: Wu MD, Yang SK, Hou ZY, She ZK, Zeng FP, Ji ZY. From Design to Safety Analysis: Heterogeneous Model Transformation and Cross-validation. Ruan Jian Xue Bao/Journal of Software, 2026, 37(9): 3521–3556 (in Chinese). http://www.jos.org.cn/1000-9825/7605.htm

From Design to Safety Analysis: Heterogeneous Model Transformation and Cross-validation

WU Meng-Dan[1], YANG Shun-Kun[1], HOU Zhan-Yi[1], SHE Zhi-Kun[2], ZENG Fu-Ping[1], JI Zhen-Yan[3]

[1](School of Reliability and Systems Engineering, Beihang University, Beijing 100191, China)

[2](School of Mathematical Sciences, Beihang University, Beijing 100191, China)

[3](School of Cyberspace Science and Technology, Beijing Jiaotong University, Beijing 100044, China)

Abstract: In the development of complex software systems, effective integration between the design phase and the verification phase is crucial for ensuring system reliability and functional correctness. However, heterogeneity in modeling languages, semantics, and data structures across design and verification tools often leads to semantic inconsistencies during model transformation, insufficient toolchain interoperability, and inadequate verification coverage. To address these challenges, this study proposes a layered transformation and multi-perspective cross-verification mechanism based on a unified intermediate representation (UIR). Starting from the design model, systems-

* 基金项目: 国家重点研发计划 (2022YFA1005102)
本文由“形式化方法与应用”专题特约编辑安杰副研究员、顾斌研究员、李建文教授推荐.
收稿时间: 2025-09-08; 修改时间: 2025-10-28; 采用时间: 2025-12-08; jos 在线出版时间: 2025-12-24
CNKI 网络首发时间: 2026-06-10

theoretic process analysis (STPA) is applied to conduct hazard analysis and identify unsafe control actions, from which safety constraints are extracted and cross-checked against existing functional, timing, and safety properties. Subsequently, a mapping from the design model to the UIR is constructed, and the structure, behavior, timing, and safety constraints are formally specified within a unified syntactic and semantic domain. On this basis, layered transformation rules, together with traceability metadata, are defined to cover the above elements. Based on the UIR, complementary verification models, such as temporal, logical, and probabilistic models, are derived, and the safety constraints produced by STPA are systematically translated into verifiable properties. Multi-perspective cross-verification is then conducted, including consistency checking among source, intermediate, and target models, temporal and logical property verification, probabilistic analysis, and satisfiability checking of safety constraints. Case studies on an autonomous vehicle control system and a multi-domain collaborative unmanned aerial vehicle (UAV) control system demonstrate that the proposed method improves verification coverage and accuracy, enhances traceability throughout transformation and verification, and provides a consistent and provable technical pathway for safety-driven development of complex systems.

Key words: safety risk analysis; unified intermediate representation (UIR); model transformation; multi-perspective cross-verification

随着现代工业和信息技术的快速发展, 复杂软件系统在航空航天[1]、自动驾驶[2]、工业控制[3]、医疗设备[4]等安全攸关领域[5]的应用日益广泛. 这些系统通常具有高度的异构性和复杂性, 其开发过程需要设计阶段与验证阶段的高效协同, 以保障系统的功能正确性与可靠性. 实现设计模型与验证模型的正确转换与语义一致性保证, 已成为确保安全攸关系统质量的关键技术挑战.

针对设计模型与验证模型互操作问题, 现有研究主要围绕安全风险分析、模型转换与集成验证这 3 个核心技术领域展开探索. 安全风险分析方面包含基于模型检查的形式化方法[6,7]、系统理论/MBSE 集成方法[8,9]以及概率化与数据驱动扩展的方法[10]. 其中基于模型检查的形式化方法, 通过对并发与时序行为的近似穷尽探索, 配合反例回注与 MBSE 贯通, 形成早期缺陷发现与工程回注的闭环; 系统理论/MBSE 集成方法将 STPA (systems-theoretic process analysis) 工件属性化并在架构层施加系统级安全约束, 实现危险到设计约束的前置化映射; 概率化与数据驱动扩展方法引入时序逻辑、数据驱动可达性与贝叶斯网络等, 面向不确定性开展定量化评估. 但跨方法/工具的语义割裂, 难以在统一语义域内对齐结构、行为、时序、概率与故障等多维语义; 风险分析与形式化验证之间缺乏可组合的互证机制与端到端证据链, 导致安全约束难以以可验证的形式贯穿设计与验证.

模型转换研究方面主要从理论、工具以及工程层面发展. 其中理论上刻画转换空间、方向性 (单/双向、增量) 与执行机制 (如 QVT), 以提供可执行的语义基础[11]; 工具生态上发展 QVT/EMF、性能优化引擎与可视化语言以提升效率与易用性[12]; 工程落地上针对 SysML/UML/AADL/Simulink 等构建到时间自动机、Stateflow、PRISM、TEGG 等的显式映射与域化抽取, 为下游验证提供可验证模型[13]. 然而, 以语法为主的点对点映射难以保证语义一致性与性质保持, 尤其在时序、概率、故障等非功能语义上; 跨工具互操作依赖个案适配, 呈现 $N\times M$ 复杂度与高维护成本, 缺乏可追溯的转换证据链以支撑安全约束的端到端映射.

模型验证方面正由单一技术走向“集成验证”. 通过统一前端承载一阶、时序与超属性, 编排归纳证明、有界模型检查 (bounded model checking, BMC)、测试生成与合成等多后端以形成覆盖互补[14,15]; 在跨层闭环与运行时联动[16,17]中, 将设计级证明、仿真测试与运行时监测/合成耦合, 实现设计-验证-运行的动态保证; 并在特定领域构建可组合验证框架以提升规模化与适配性. 但是, 不同验证后端在抽象层级与语义假设上缺乏同构性, 反例、候选不变式与证明工件难以共享与复用, 证据的可迁移性与可累计性受限; 同时, 安全风险分析产出的约束与论据难以被验证链路以可计算方式刻画与验证, 导致安全目标与功能/时序/概率属性之间的系统化交叉验证机制不足.

综上, 现有方法仍面临 3 个关键及技术挑战: (1) 语义一致性不足: 缺乏统一的语义表示机制, 设计模型在转换为验证模型时容易出现信息丢失或语义偏差, 尤其在处理时序约束、概率属性与故障语义等定量与非功能特性时, 尚缺完整的理论与工程支撑; (2) 工具链互操作性受限: 多数转换方法针对特定设计/验证工具进行点对点适配, 缺乏可扩展、可追溯的通用机制以支撑跨平台的语义一致性转换与安全约束的端到端映射; (3) 验证方法系统性薄弱: 难以保证多视图 (功能、行为、时序、安全等) 间的一致性, 缺乏以安全约束为核心的系统化交叉验证与证据累计机制, 制约验证结果的可靠性提升与设计闭环质量的保障.

因此, 迫切需要建立统一的中间表示机制和多重交叉验证框架, 以实现异构模型间的语义保持转换, 并通过多

类型验证模型的互补校验确保复杂系统从设计到安全风险分析全过程的一致性与可靠性.

为解决这些问题, 本文提出一种基于统一中间表示 (unified intermediate representation, UIR) 的建模与验证方法, 如图 1 所示. 通过多样化需求分析支持多种设计模型, 利用统一中间表示对异构设计模型信息进行统一语义提取与分析, 并结合 STPA 理论进行多维属性的校验与补充; 在此基础上, 定义分层模型转换与追溯机制, 系统化映射到时序、逻辑与概率等互补的验证模型; 最后, 围绕安全约束开展源-中间-目标模型一致性检查与多视角交叉验证, 形成端到端证据链. 本文的主要贡献如下.

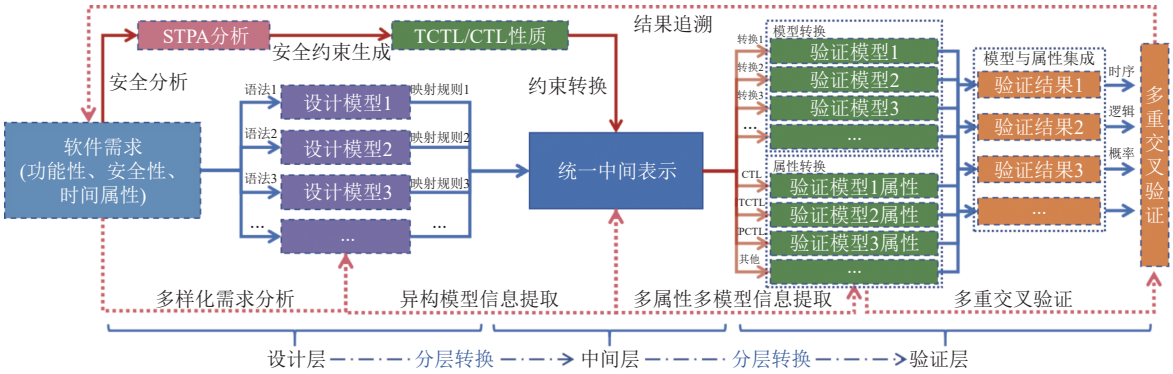

图 1 基于统一中间表示的异构模型转换与多重交叉验证方法框架

(1) 面向安全分析的 UIR 语义互操作性框架: 构建以 UIR 为核心的语义互操作性框架, 将基于 STPA 的危险分析与不安全控制行为识别前置于设计阶段, 并将由此提炼的安全约束纳入统一语法与语义域中, 与结构、行为与时序信息协同进行形式化建模与管理. UIR 中显式承载安全约束的语义内涵与元数据 (来源、触发条件、控制链路、风险场景), 从机制上抑制跨工具与语言转换导致的安全语义丢失与歧义传播, 从而确保设计-验证贯通过程的语义完整性与互操作性.

(2) 面向安全约束映射与追溯的分层模型转换机制: 提出“设计模型-UIR-验证模型”的分层转换方法, 系统化定义覆盖结构/行为/时序/安全约束的信息提取与语义映射规则, 并在规则层嵌入追溯性元数据, 维护从源要素到目标模型与可验证属性的证据链. 将 STPA 导出的安全约束及其上下文 (触发条件、控制路径、失效模式) 类型转换映射为不变性、时序必然性、可达性与概率阈值等可验证属性, 实现端到端的可映射、可追溯与可审计, 提升验证模型生成效率与语义一致性, 同时避免安全分析产物在转换链路中的语义失真.

(3) 以安全约束为核心的多视角交叉验证方法: 基于 UIR 派生互补的时序、逻辑与概率验证模型, 将 STPA 提炼的安全约束系统化转换为可验证属性, 贯穿源、中间与目标模型的一致性检查, 以及时序/逻辑性质验证与概率分析, 形成多视角的交叉一致性校验闭环. 通过安全约束驱动的验证组织方式, 与功能、时序及既有安全属性进行相互印证与冲突检测, 提升验证覆盖度与缺陷定位能力, 在生命周期早期识别并定位潜在安全问题, 从而增强验证的全面性与鲁棒性.

本文第 1 节介绍所需的基础知识. 第 2 节介绍从设计到安全风险分析的总体框架. 第 3 节介绍设计层模型的建立与安全风险分析. 第 4 节介绍本文构建的统一中间表示的设计逻辑及语法语义. 第 5 节介绍本文所提的分层模型转换机制. 第 6 节介绍多重交叉验证机制. 第 7 节对本文所提方法进行可扩展性分析. 第 8 节介绍模型转换工具的实现. 第 9 节选取自动驾驶汽车控制软件和多域协同无人机飞控系统实际案例进行实验研究. 第 10 节介绍相关工作. 最后总结全文.

1 基础知识

1.1 设计模型

设计模型是系统开发过程中用于描述和规范系统特征的抽象表示, 通过对系统的功能、行为和结构的多维度

建模, 能够有效捕获系统需求, 支持设计决策.

定义 1. 设计模型可以被定义为 $M_{dg} := (dm\_C, dm\_S, dm\_T, dm\_I, dm\_\phi)$, 其中,

- dm_C 表示类图集合, 描述系统的静态结构;
- $dm\_S = (dm\_Q, dm\_E, dm\_\delta, dm\_q_0)$ 表示状态图集合, 描述系统的动态行为; $dm\_Q = (dm\_q_1, dm\_q_2, \ldots, dm\_q_n)$ 表示系统状态的集合; $dm\_E = (dm\_e_1, dm\_e_2, \ldots, dm\_e_m)$ 表示事件的集合; dm_δ: $dm\_Q \times dm\_E \to dm\_Q$ 是状态转移函数, 表示从状态 dm_q_i 在事件 dm_e_j 的触发下转移到 dm_q_k: $dm\_\delta(dm\_q_i, dm\_q_j) = dm\_q_k$; $dm\_q_0 \in dm\_\phi$ 表示初始状态.
- $dm\_T = \{(dm\_e_1, dm\_t_1), (dm\_e_2, dm\_t_2), \ldots, (dm\_e_n, dm\_t_n)\}$ 是序列图集合, 表示组件间的交互; $dm\_e_i \in dm\_E$ 表示事件; $dm\_t_2 \geqslant 0$ 表示事件触发的时间戳;
- $dm\_I = (I_{\text{in}}, I_{\text{out}})$ 表示系统输入输出的接口集合; I_{in} 表述输入接口集合; I_{out} 表示输出接口集合;
- dm_ϕ 表示属性的集合.

1.2 验证模型

验证模型通过严格的数学语义、目标导向性、抽象性和工具依赖性等特点, 采用模型检测等方法验证系统的关键性质 (如安全性、实时性和可靠性).

定义 2. 验证模型可以被定义为一个四元组 $VM = (TA, P, V, \Upsilon)$, 其中,

- TA 表示时间自动机集合, 用于形式化建模系统的时序行为;
- P 表示性质集合, 可以表示为一个多层次的元组: $P = (P_s, P_r, P_l, P_p)$; P_s 表示安全性质集合; P_r 表示实时性质集合; P_l 表示可靠性质集合; R_p 表示性质之间的依赖关系;
- V 表示验证方法集合;
- Υ 表示映射函数.

定义 3. 验证性质可以表示为一个三元组 $Prop = (id, spec, type)$, 其中, id 表示性质的唯一标识符; $spec$ 表示性质的形式化规约; $type$ 表示性质的类型.

定义 4. 时间自动机 (timed automata, TA) 是一种扩展的有限状态机, 用于建模具有时序行为的系统, 可以被定义为: $T_A = (L, l_0, C, A, E, I)$, 其中, $L = \{l_1, l_2, \ldots, l_n\}$ 表示位置的有限集合; $l_0 \in L$ 表示初始位置; $C = \{c_1, c_2, \ldots, c_m\}$ 表示时钟变量的有限集合; $A = \{a_1, a_2, \ldots, a_p\}$ 表示动作的有限集合; $E \subseteq L \times A \times G \times 2^C \times L$ 表示边的有限集合, 即状态之间的转移关系; $G = \{g \mid g : C \to \{\text{true}, \text{false}\}\}$ 表示守卫条件; 2^C 表示时钟重置集合, 即在状态转移时需要重置的时钟变量; I: $L \to G$ 表示位置不变性函数.

1.3 模型转换

模型转换是实现异构模型间语义映射和结构转换的关键技术, 通过提供独立于具体建模语言的抽象层, 采用统一中间表示作为模型转换的核心机制, 可以实现源模型到目标模型的正确映射.

定义 5. 模型转换可以定义为一个五元组: $MT = (S, T, R, \Phi, \Psi)$, 其中,

- S 表示源模型集合;
- T 表示目标模型集合;
- R 表示统一中间表示, 作为异构模型之间的桥梁;
- Φ 表示源模型到中间表示的转换函数;
- Ψ 表示中间表示到目标函数的转换函数.

定义 6. 统一中间表示可以表示为一个四元组: $R = (E, A, C, M)$, 其中,

- E 表示实体集合, 包含模型中的基本元素;
- A 表示属性集合, 描述实体的关系和限制条件;
- C 表示约束集合, 定义实体间的关系和限制条件;
- M 表示元数据集合, 包含模型的语义信息和转换规则.

定义 7. 模型转换的语义保持性可以表示为: 对于转换 $\tau: S \to T$, 若存在语义等价关系 $\equiv_{sem}$, 使得对于任意源模型 $m_s \in S$ 和对应的目标模型 $m_t \in T$, 都有 $m_s \equiv_{sem} m_t$, 则称转换 τ 是语义保持的.

1.4 安全风险分析

安全风险分析基于系统理论的事故模型和过程 (systems-theoretic accident model and process, STAMP), 通过层次化控制结构分析控制系统中的安全风险.

定义 8. 基于 STAMP 的安全风险控制模型可以定义为一个六元组 $SRC = (E, K, N, P, T, Z)$, 其中,

- E 表示层次化控制结构;
- K 表示控制动作集合;
- N 表示反馈信息集合;
- P 表示安全约束集合;
- T 表示事故致因分析集合;
- Z 表示风险环节措施集合.

定义 9. 不安全控制动作 (unsafe control action, UCA) 可以表示为一个四元组 $UCA = (ca, ctx, tp, haz)$, 其中,

- $ca \in K$ 表示具体的控制动作;
- ctx 表示系统上下文或环境条件;
- tp 表示 UCA 的类型;
- haz 表示可能导致的危险状态.

定义 10. 安全约束可以表示为一个三元组 $SC = (sc\_id, sc\_constraint, sc\_level)$, 其中,

- sc_id 表示约束的唯一标识符;
- $sc\_constraint$ 表示约束的形式化描述;
- $sc\_level \in E$ 表示约束所属的控制层级.

2 总体框架

当前复杂软件系统开发过程中, UML、SysML、SCADE 等异构设计工具并存, 模型语义和数据格式差异显著, 导致跨工具协同与验证工作割裂. 传统的模型转换大多依赖一次性脚本, 实现单一模型至单一模型的转换, 语义保持受限, 且难以实现端到端的追溯. 另一方面, 功能、时间与安全属性相互耦合, 依赖单一验证模型或单一求解器覆盖不足, 验证可靠性较低, 难以及时暴露早期设计缺陷与安全隐患. 尽管 STPA 等方法能够输出危险与安全信息, 但其结果难以直接进行形式化验证, 导致安全属性与功能和时序属性分离.

为解决这些局限, 本文提出了一种基于统一中间表示的总体框架, 结合分层模型转换与多模型交叉验证策略, 构建从设计到安全风险分析的可追溯流程. 框架围绕状态、行为、时间与规则, 实现多异构设计模型向 UIR 的信息提取与转换, 同时通过数据解析、数据规范和信息整合, 使得 UIR 通过“单一事实源”的方法转换为多种验证模型进行多重交叉验证, 提供设计“元素-UIR-验证元素”的双向追溯, 确保了标识一致性、结构设计多样性与验证多样性. 如图 2 所示, 在此基础上, 构建 3 层管线: 其一, 设计层至中间层进行元素识别、语义对齐, 确保语义保持与信息完整; 其二, 中间层至验证层, 面向不同验证目标派生互补模型族, 例如基于时间自动机的实时验证 (如 UPPAAL 模型)、基于时序逻辑的逻辑验证 (如 NuSMV 模型)、过程代数与可达性验证 (如 SPIN 模型) 以及概率模型检测 (如 PRISM 模型) 等; 其三, 属性同步层, 将需求与 STPA 产出的安全约束模板化为 LTL/CTL/TCTL 等属性, 并统一注入至中间层, 而后传递至验证层以实现属性一致性校验与潜在问题发现. 在安全分析方面, 以系统控制结构为脉络, 按照“识别系统危险-识别不安全控制-生成约束-输出安全属性”的流程自动化地产生安全属性, 与功能/时间属性共同参与验证. 验证阶段采用多模型、跨工具自动地进行交叉验证与反例回溯, 即任一模型产生的反例均可映射至 UIR 与原始设计, 以支撑故障定位与恢复, 从而降低单一模型的检测局限性.

通过 UIR 统一承载异构设计与验证语义, 系统性地降低信息丢失与歧义, 提升验证模型生成的一致性与可追溯

性, 实现语义一致的互操作性; 多模型、多工具与多属性的多重交叉验证提高覆盖度与结论的稳健性, 实现全面而稳健的验证; 将 STPA 生成的安全约束与功能/时间属性在同一闭环内统一校验, 使复杂软件系统的潜在问题可在设计早期被发现、定位与修复, 降低生命周期后期成本与风险, 实现早期的安全保障与缺陷定位; UIR 与分层管线可以在实际环境中自动化部署与增量式运行, 同时适配主流的设计工具与验证工具, 支持复杂系统的持续验证与复用.

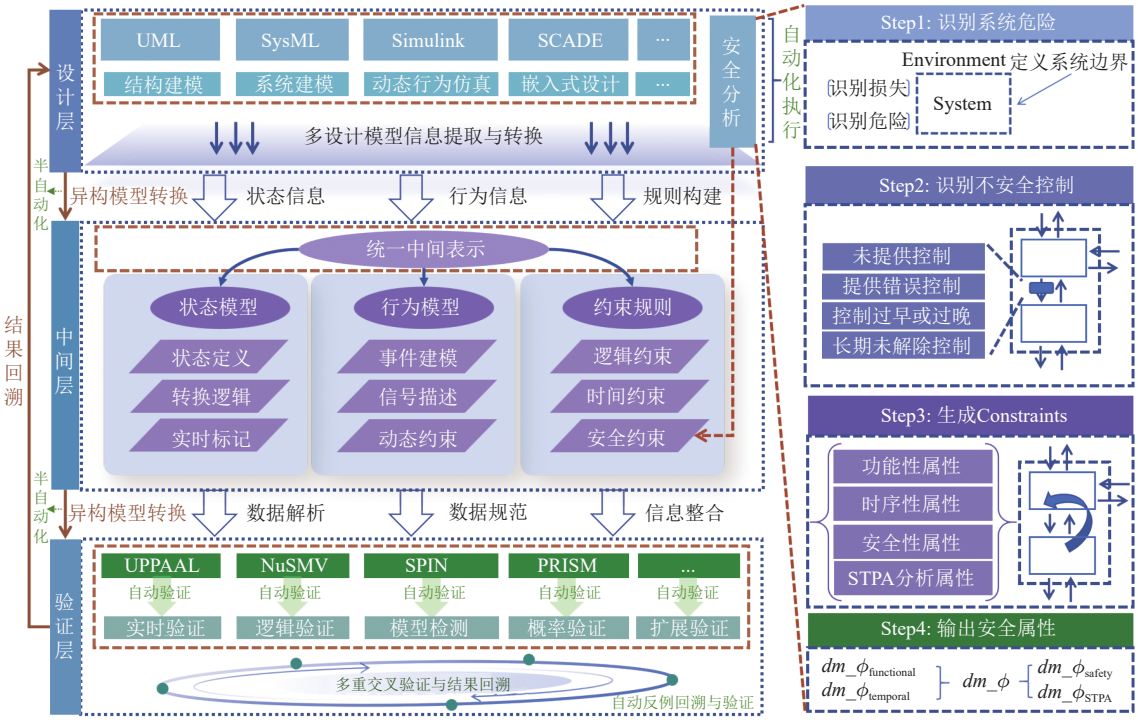

图 2 基于统一中间表示的异构模型转换与多重交叉验证整体架构图

3 设计层模型建立与安全风险分析

设计层一方面依据软件需求 (功能性、安全性、时间属性等) 构建形式化设计模型; 另一方面基于同一需求开展 STPA 分析, 并在 STAMP 框架下提炼系统级约束. 通过设计模型刻画系统结构与行为边界, STPA/STAMP 识别控制失效与不安全控制动作并生成约束, 从而形成结构-约束的互补视图.

3.1 设计层模型建立

在设计阶段, 使用统一建模语言等工具建立系统的设计模型, 表征系统的需求和相关约束. 为进一步增强系统的安全属性分析与描述, 在设计阶段添加系统理论过程分析 STPA 以识别系统危险及自动化地生成规约.

定义 11. 系统控制行为可以被定义为 $G_{\text{control}} := (C_{\text{control}}, P_{\text{process}}, UC, L)$, 其中,

- $C_{\text{control}} = (N, R)$ 表示系统的控制结构, 描述控制器、被控对象及二者之间的交互; N 表示控制器与被控对象的节点集合; $R \subseteq N \times N$ 表示控制信号与反馈信息的边集合;
- $P_{\text{process}} = \{p_1, p_2, \ldots, p_k\}$ 表示控制行为过程中的环境变量集合;
- $UC = \{uc_1, uc_2, \ldots, uc_l\}$ 表示可能导致系统危险的控制行为的集合;
- $L = \{l_1, l_2, \ldots, l_m\}$ 表示系统可能造成的失效或不可恢复的危险集合.

3.2 安全风险分析

采用系统层次性分析方法, 识别系统中可能导致危险的控制行为.

(1) 基于控制图 C_{control}, 通过枚举方法分析系统的 4 种类型的控制行为: 未提供必需控制、提供错误控制、过早或过晚的控制以及持续的控制. 通过分析不安全控制 (unsafe control, UC), 生成系统约束 $Constraints = \{C_1, C_2, \ldots, C_r\}$, 其中 C_i 表示一条约束, 即一条具体的控制规则, 保证相关活动不会导致危险. 若控制器未发送信号, 会导致 P_{process} 中变量 p_i 超出安全范围: $C_i = \neg(Signal(c_{\text{controller}}, c_{\text{process}})) \rightarrow (p_i \notin Range_{\text{safe}})$. 控制信号在时间窗口 T_{start}和 T_{end} 外被触发, 则控制行为无效: $C_j = (t \notin [T_{\text{start}}, T_{\text{end}}]) \rightarrow Invalid(c_{\text{controller}}, Signal(t))$.

(2) 基于建立的系统约束集合以及危险集合 L, 生成规约描述 dm_ϕ_{STPA}, 用于验证系统是否满足安全属性: $dm\_\phi_{\text{STPA}} = \bigwedge_i^r(\neg UC_i \rightarrow Safe(L))$. 使用自动化工具生成规约和控制约束并输出控制结构 C_{control}, 包括控制器、被控对象以及信号传递的逻辑关系, 添加逻辑变量 P_{process} 及失效条件 L, 自动分析控制信号的安全性, 生成潜在的不安全控制行为 UC, 并关联其导致的危险 L, 生成规约 $Constraints$ 以形式化逻辑描述形式导出, 导出的规约可直接嵌入到系统设计属性集合 dm_ϕ 中. 最终, 将原有的属性 dm_ϕ 扩展为: $dm\_\phi = dm\_\phi_{\text{functional}} \cup dm\_\phi_{\text{temporal}} \cup dm\_\phi_{\text{safety}} \cup dm\_\phi_{\text{STPA}}$. 通过引入 STPA 分析, 可以自动化生成安全要求规约, 系统设计模型能够扩展安全性和动态属性, 同时, 生成的属性 dm_ϕ_{STPA} 与原有的功能性和时序性属性相结合, 为后续的模型验证提供正确且全面的形式化描述.

(3) 值得注意的是, STPA 的分析过程本身也是一个系统化的风险分级与冲突消解过程. 其对不安全控制行为 (UCA) 的分类 (如“未提供控制”“提供但过早/过晚”等) 隐式构建了一个风险优先级框架, 通常“未提供控制”导致的危害具有最高严重等级. 此外, 从系统级危害 (system-level hazard) 向下追溯的分析路径, 确保了所有衍生出的安全约束都服务于顶层安全目标. 在此过程中, 不同约束间的潜在冲突会在系统层面被工程师识别和处理 (例如, 针对同一控制动作的不同 UCA), 并基于工程经验与安全标准进行消解, 最终输出一个内部一致、可用于验证的约束集. 因此, 在验证前段可通过系统分析, 最大程度预防约束间的逻辑冲突.

4 统一中间表示设计

统一中间表示以“单一事实源”统一承载异构设计与验证语义, 解耦设计多样性与验证多样性; 以状态、行为、规则方面等信息为输入, 在结构、行为、时间与安全这 4 个语义域上构建可扩展的统一中间表示模型, 如图 3 所示, 并为每一域匹配到目标验证形式 (如时间自动机/PRISM 与 LTL/CTL 等) 的映射瞄点与双向追溯关系. 通过规范化的元素抽取、语义对齐与缺省补全, 将设计模型映射为 UIR, 再以可证明的规则从 UIR 派生多种验证模型族与属性族, 实现“一次抽取, 多端映射, 可回溯验证”. 以统一标识与语义瞄点贯穿需求-设计-验证-反例回溯全链条, 保证语义完整性与一致性, 使功能、时间与安全属性在同一闭环内被协同校验.

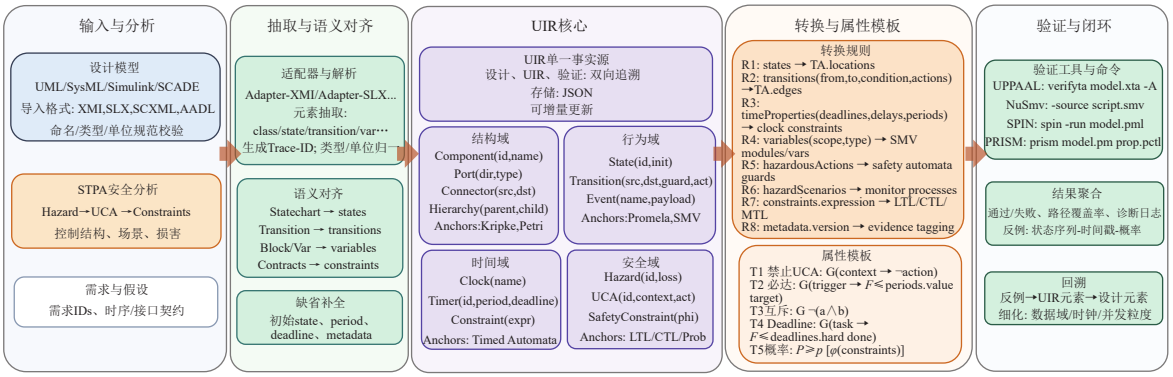

图 3 统一中间表示设计原理图

4.1 统一中间表示设计元素

UIR 一方面统一不同模型之间的数据结构, 另一方面, 支持安全约束的存储和转换. 如图 4 所示, UIR 包含模型 (model) 结构、状态 (states) 信息、转换 (transitions) 信息、变量 (variables) 信息、约束 (constraints) 信息、危险场景 (hazardScenarios) 信息以及时间属性 (timeProperties) 信息等内容.

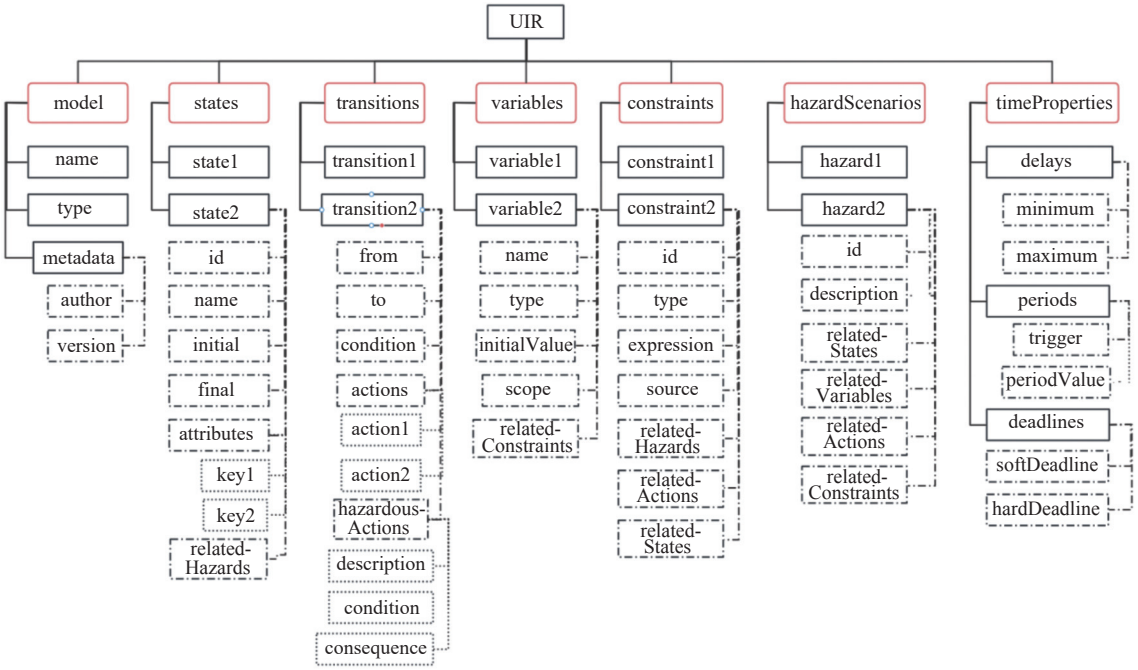

图 4 UIR 设计元素

● model 中 name 和 type 是必需字段, 分别表示模型的名称和类型; metadata 是可选字段, 用于描述作者、版本等开发背景, 提供模型管理和版本控制. states 字段包含唯一标识 id、状态名称 name, 并通过 initial 和 final 标记初始状态与终止状态. 可扩展的 attributes 字段采用键值对存储自定义属性, 实现对状态特征的灵活扩展. relatedHazards 可选, 用于标明该状态对应的潜在危险源或关联风险.

● transitions 每个 transition 定义状态间的迁移逻辑, 包括起始状态 from 和目标状态 to. condition 为转换触发条件, 采用布尔表达式描述. actions 字段下可细分为多个动作为状态迁移时执行的操作, 支持复合动作和 hazardous-Actions (危险操作), 以表达系统动态特性. description 用于补充文字说明, consequence 可用以记录迁移结果或者影响.

● variables 部分定义系统运行所需的全局或局部变量, 包括 name (变量名)、type (数据类型)、initialValue (初始值). scope 字段用于区分变量作用范围 (如全局/局部), relatedConstraints 帮助变量与约束建立直接关联.

● constraints 中约束对象包括 id、type、expression (约束表达式) 和 source (约束来源). 其中 type 区分约束类型, expression 采用逻辑表达式或数学式描述, relatedHazards、relatedActions 和 relatedStates 用于标识该约束关联的危险、动作与状态.

● hazardScenarios 中的每个 hazard 元素包括唯一标识 id、危险描述以及与该危险相关的状态 (related States)、变量 (relatedVariables)、动作 (relatedActions) 和约束 (relatedConstraints).

● timeProperties 中 delays 定义延迟相关属性, periods 规定周期任务的触发条件与周期时长, deadlines 用于约束任务执行的硬截止时间或软截止时间: 前者为绝对不可违反的时间限制, 后者允许一定范围内的滞后.

4.2 统一中间表示语法

UIR 采用元模型的方式定义其核心组成元素及其关系.

定义 12. UIR 模型可以被定义为 $UIR :=< uir\_M, uir\_S, uir\_T, uir\_V, uir\_C, uir\_D, uir\_P >$, 其中,

● uir_M 表示模型信息集合, 描述 UIR 模型的全局信息 (如模型名称等);

● $uir\_S = \{s_i \mid s_i = (id, isInit, isFinal, attributes)\}$ 表示状态集合, 定义系统的离散状态空间; id 表示状态标识;

isInit 表示初始状态标记; *isFinal* 表示最终状态标记; *attributes* 表示自定义的状态属性集合;

- $uir\_T = \{t_j \mid t_j = (src, tgt, cond, action)\}$ 表示转换集合, 定义系统从一个状态到另一个状态的转移规则; *src* 表示源状态; *tgt* 表示目标状态; *cond* 表示迁移条件, 是布尔表达式; *action* 是迁移执行动作;
- $uir\_V = \{v_k \mid v_k = (name, type, range, initial)\}$ 表示变量集合, 定义系统的全局变量和局部变量; *name* 表示变量名称; *type* 表示变量类型; *range* 表示变量范围; *initial* 表示变量初始值;
- *uir_C* 表示约束集合, 定义系统的静态约束条件 (如变量范围约束、逻辑约束等);
- $uir\_D = \{d_n \mid d_n = (trigger, event, resources, actions)\}$ 表示动态行为集合, 定义系统的动态行为 (如事件、资源竞争等); *trigger* 表示触发行为的条件; *event* 表示被触发的事件类型; *resources* 表示发生动作涉及的系统资源; *actions* 表示触发行为后执行的动作序列;
- $uir\_P = \{p_m \mid p_m = (trigger, delay, period)\}$ 表示时间约束集合, 定义系统的时间属性 (如延迟、周期等); *trigger* 表示触发行为的条件; *delay* 表示延迟执行任务; *period* 表示周期执行任务.

UIR 状态定义描述 UIR 中的离散状态, 支持初始状态和最终状态的标记; 转移规则定义语法描述状态之间的转换条件和动作, 支持条件表达式和动作语句的嵌套; 变量定义语法描述 UIR 中的全局或局部变量, 包括类型和初始值; 时间约束定义语法描述 UIR 模型的时间属性.

同时, UIR 引入范畴理论的“共极限 (colimit)”和“态射 (morphism)”以支持组件组合语法. 共极限统一表示多个模块化模型之间的组合关系, 定义模块间的组合规则, 将各模块建模为范畴中的对象, 并通过共极限操作得到全局系统的统一建模结果. 多个状态组件被看作是范畴中的对象, 各状态之间的转换 (包括触发条件和动作) 被视为态射. 态射表示不同模型元素之间的关系映射 (mapping), 可以表示组件之间的交互映射、模型层次的功能映射或验证工具中的语义转换映射. 模型的某一变量在不同工具中存在不同的表达形式, 这种映射关系可用态射表示.

4.3 统一中间表示语义

UIR 的语义涵盖静态语义和动态语义两部分.

(1) 静态语义

静态语义描述 UIR 模型的结构性约束及其组成元素的合法性, 主要包括状态语义、变量语义、约束语义等. 其中, 状态语义表征每个状态必须具有唯一标识符, 且初始状态和最终状态的数量满足两个条件: 其一, 至少存在一个初始状态, 即 $\exists s \in uir\_S,\ s.initial = \text{true}$; 同时至少存在一个最终状态, 即 $\exists s \in uir\_S,\ s.final = \text{true}$. 变量语义的定义必须满足类型一致性, 且变量的初始值必须在其定义范围内, 即 $\forall v \in uir\_V,\ v.initial\_value \in v.range$. 约束语义是布尔类型的逻辑表达式, 且约束的作用范围需要明确, 即 $\forall c \in uir\_C,\ c.expression: bool$. 变量语义表征变量初始化必须严格对应定义的类型和值域的范围, 即 $\exists v_i \in uir\_V: initial(v_i) \in range(v_i)$, 其中 $type(v_i)$ 有效. 约束语义表征约束语句应明确地作用于指定变量或状态集合的布尔类型逻辑表达式, 即: $\exists c \in uir\_C, dom(c) = \{uir\_V, uir\_S\},\ eval(c) = bool$.

(2) 动态语义

动态语义描述 UIR 模型的运行行为, 包括状态转移、时间约束和动态行为. 其中, 状态转移语义表征状态转移的条件必须满足布尔表达式, 且状态的转换必须是确定性的, 即 $\forall t \in uir\_T,\ t.condition: bool$; 转移规则的执行会引起状态的更新, 即 $(s, v) - [t.condition] - > (s', v')$, 其中, s 和 s' 分别表示转移前后的状态, v 和 v' 表示转移前后的变量值; 同时, 状态转移应明确指定迁移条件表达式 (condition) 和迁移动作 (action), 即: $\forall t \in uir\_T,\ t = (src, tgt, cond, action),\ eval(cond) = \text{true} \rightarrow exec(action)$, 并且将状态由 *src* 转移至 *tgt*. 时间约束语义表征时间约束, 定义了系统的时间行为, 必须满足时序逻辑公式: $\forall p \in uir\_P,\ G(p.expression)$, 其中 G 表示全局时序逻辑操作符, 确保时间约束在系统运行期间始终成立. 同时, UIR 的时间约束被表示为时序逻辑约束表达, 需满足约束: $\forall p \in (trigger, delay, period) \in uir\_P,\ G(trigger \rightarrow F(delay \leqslant t \leqslant delay + period))$. 动态行为语义表征包括事件触发和资源竞争等, 使用事件驱动模型描述语义: $trigger \rightarrow action$, 其中 *trigger* 表示事件触发条件, *action* 表示触发后的行为. 对于动态行为 (事件触发和资源竞争), 采用过程代数的行为轨迹定义, 如: $event(trigger)$: *resource competition* $\rightarrow$ $action(settled)$, *resources mutually-exclusive*.

5 分层模型转换机制

分层模型转换机制涵盖模型的分层映射、属性的分层映射以及转换正确性检查. 整个转换框架的核心是一套预定义的分层映射规则集. 该规则集采用模块化与增量式的设计理念, 旨在平衡转换过程的可靠性与面对不同建模语及验证工具时的扩展灵活性.

5.1 模型分层映射规则

5.1.1 设计层到中间层模型映射规则

解析设计模型的各元素, 按照预定义的映射规则逐一转换为 UIR 中对应的元素, 实现设计模型到 UIR 的映射. 定义通用的映射算法 (算法 1) 系统化地完成设计模型到 UIR 结构的转换过程.

算法 1. 设计模型到 UIR 的映射.

输入: 设计模型 $M_{dg} := (dm\_C, dm\_S, dm\_T, dm\_I, dm\_\phi)$;
输出: 统一中间表示 $UIR :=< uir\_M, uir\_S, uir\_T, uir\_V, uir\_C, uir\_D, uir\_P >$.

1. $elementArr \leftarrow parse(M_{dg})$ //解析设计模型 M_{dg}, 提取各类元素
2. **for each** $E \in elementArr$ **do**
3. $T \leftarrow Trans(E)$ //*Trans* 为基础映射
4. $UIR.add(T)$
5. **end for**
8. **return** *UIR*

定义 13. 设计模型到 UIR 的映射规则可以被定义为 $Trans(M_{dg}) :=< M, S, T, V, C, D, P >$, 其中, M 表示模型信息的映射; S 表示状态集合的映射; T 表示转换集合的映射; V 表示变量集合的映射; C 表示约束集合的映射; D 表示动态行为的映射; P 表示时间约束的映射.

为方便区分, 添加前缀“*sys_*”表示设计模型的元素, 添加前缀“*uir_*”表示 UIR 的元素, 如表 1 所示.

表 1 设计模型与 UIR 映射规则表

编号	元素	UIR映射对象	公式表达	规则编号	规则	说明
1	sys_M	uir_M	$sys\_M \rightarrow uir\_M$	规则1	$Trans_1(sys\_M) := uir\_M$	全局信息
2	sys_S	uir_S	$sys\_s_i \rightarrow uir\_s_i$	规则2	$Trans_2(sys\_S) := uir\_S$	状态集合
3	sys_T	uir_T	$sys\_t_j \rightarrow uir\_t_j$	规则3	$Trans_3(sys\_T) := uir\_T$	转换集合
4	sys_V	uir_V	$sys\_v_k \rightarrow uir\_v_k$	规则4	$Trans_4(sys\_V) := uir\_V$	变量集合
5	sys_C	uir_C	$sys\_c_q \rightarrow uir\_c_q$	规则5	$Trans_5(sys\_C) := uir\_C$	约束集合
6	UC	uir_C, uir_D	$uc_i \rightarrow (uir\_d_{uc_i}, uir\_c_{uc_i})$	规则6	$Trans_{\mathrm{STAMP}_1}(UC) := \{uir\_D, uir\_C\}$	不安全控制映射
7	L	uir_V, uir_C	$I_j \rightarrow uir\_v_{l_j}$	规则7	$Trans_{\mathrm{STAMP}_2}(L) := \{uir\_V, uir\_C\}$	损失变量与阻断约束
8	C_{control}	uir_C	$c_k^{\mathrm{stamp}} \rightarrow uir\_c_k^{\mathrm{stamp}}$	规则8	$Trans_{\mathrm{STAMP}_3}(C_{\mathrm{control}}) := uir\_C$	自动生成安全约束
9	P_{process}	uir_V, uir_D	$p_i \rightarrow uir\_v_{p_j}$	规则9	$Trans_{\mathrm{STAMP}_4}(P_{\mathrm{process}}) := \{uir\_V, uir\_D\}$	环境过程变量

5.1.2 中间层到验证层模型结构映射

通过解析 UIR 模型中的核心元素实现中间层 UIR 到验证模型的结构映射, 并按照预定义的映射规则逐一映射为验证模型中对应元素实现. 定义通用的映射算法 (算法 2) 系统化地完成 UIR 到验证模型结构的转换过程.

算法 2. UIR 到验证模型的映射.

输入: UIR 模型 $UIR =< uir\_M, uir\_S, uir\_T, uir\_V, uir\_C, uir\_D, uir\_P >$;
输出: 验证模型 $VM =< V, T, P, Q, S >$.

1. *elementArr* ← *parse*(*UIR*) //解析 UIR 模型, 提取模型元素
2. **for each** $E \in elementArr$ **do**
3. *template* ← *Trans*(*E*) //调用映射规则, 将元素转换为验证模型模板
4. *VM.add*(*template*) //将模板添加至对应的验证模型集合中
5. **end for**
6. **return** ***VM***

算法 2 中的 UIR 模型包含模型信息 *uir_M*、状态集合 *uir_S*、转换集合 *uir_T*、变量集合 *uir_V*、约束集合 *uir_C*、动态行为集合 *uir_D*、时间约束集合 *uir_P*. 验证模型包含状态变量集合 *V*、转移规则集合 *T*、时间约束集合 *P*、安全属性集合 *Q* 以及 STAMP 安全约束集合 *S*.

定义 14. UIR 到验证模型的映射规则可以被定义为 $Trans(UIR) :=< V_{uir}, T_{uir}, P_{uir}, Q_{uir}, S_{uir} >$, 其中, V_{uir} 表示状态变量集合的映射; T_{uir} 表示转移规则集合的映射; P_{uir} 表示时间约束集合的映射; Q_{uir} 表示安全属性集合的映射; S_{uir} 表示 STAMP 安全规约集合的映射.

为方便区分, 添加前缀“*uir_*”表示 UIR 的元素, 添加前缀“*vm_*”表示验证模型的元素, 如表 2 所示.

表 2 UIR 与验证模型映射规则表

编号	UIR元素	设计模型映射对象	公式表达	规则编号	规则	说明
1	uir_M	vm_V	$uir\_M \to vm\_V$	规则10	$Trans_6(uir\_M) := vm\_V$	模型信息
2	uir_T	vm_T	$uir\_t_i \to vm\_t_i$	规则11	$Trans_7(uir\_T) := vm\_T$	转移集合
3	uir_P	vm_P	$uir\_p_j \to vm\_p_j$	规则12	$Trans_8(uir\_P) := vm\_P$	时间约束集合
4	uir_C	vm_Q	$uir\_c_k \to vm\_q_k$	规则13	$Trans_9(uir\_C) := vm\_Q$	安全约束
5	uir_D	vm_S	$uir\_d_\xi \to vm\_s_\xi$	规则14	$Trans_{\mathrm{STAMP}_5}(uir\_D) := vm\_S$	不安全控制行为集合
6	uir_S, uir_V	vm_S	$(uir\_s_\rho, uir\_s_\vartheta) \to (vm\_s_\varpi)$	规则15	$Trans_{\mathrm{STAMP}_6}(uir\_S, uir\_V) := vm\_S$	损失监控变量集合
7	uir_C	vm_S	$uir\_c_\mu \to vm\_s_\mu$	规则16	$Trans_{\mathrm{STAMP}_7}(uir\_C) := vm\_S$	安全约束集合

5.2 属性分层映射规则

5.2.1 设计层到中间层属性映射规则

定义 15. 系统约束 ϕ_{total} 由 3 个不相交的子集组成: $\phi_{\mathrm{total}} = \phi_{\mathrm{req}} \cup \phi_{\mathrm{intrinsic}} \cup \phi_{\mathrm{STAMP}}$, 其中 ϕ_{req} 表示由于需求分析得出的约束集合, $\phi_{\mathrm{intrinsic}}$ 表示设计模型固有的约束集合, ϕ_{STAMP} 表示从 STAMP 分析结果派生的安全约束集合.

规则 17. 需求分析约束映射规则: $Trans_{\mathrm{req}}(\phi_{\mathrm{req}}) := uir\_C_{\mathrm{req}}$.

规则 17 表示需求分析得出的约束集合 ϕ_{req} 映射为 UIR 的需求约束集合 uir_C_{req}, 映射过程将每个需求约束 $\phi_{\mathrm{req}_i} \in \phi_{\mathrm{req}}$ 转换为对应的 UIR 约束 $uir\_C_{\mathrm{req}_i}$, 并确保约束的标识符、类型、形式化表达式以及关联属性等信息的完整性与一致性. 映射结果满足如下约束.

- $\phi_{\mathrm{req}_i}.id = uir\_C_{\mathrm{req}_i}.id$;
- $\phi_{\mathrm{req}_i}.type = uir\_C_{\mathrm{req}_i}.type$;
- $\phi_{\mathrm{req}_i}.expression = uir\_C_{\mathrm{req}_i}.expression$;
- $\phi_{\mathrm{req}_i}.source =$ “*requirement_analysis*”;
- $\phi_{\mathrm{req}_i}.relatedActions = \mathfrak{R}_A(\phi_{\mathrm{req}_i})$;
- $\phi_{\mathrm{req}_i}.relatedStates = \mathfrak{R}_S(\phi_{\mathrm{req}_i})$;
- $\phi_{\mathrm{req}_i}.relatedHazards = \varnothing$.

其中, $\mathfrak{R}_A(\phi_{\mathrm{req}_i})$ 表示需求约束关联动作提取函数, 通过自然语言处理与语义分析识别需求约束 ϕ_{req_i} 中涉及的系统动作集合; $\mathfrak{R}_S(\phi_{\mathrm{req}_i})$ 表示需求约束关联状态提取函数, 通过依赖性分析识别需求约束 ϕ_{req_i} 中涉及的系统状态集合; $\varnothing$ 表示需求约束初始时不关联危险场景.

规则 18. 设计模型固有约束映射规则: $Trans_{\text{int}}(\phi_{\text{intrinsic}}) := uir\_C_{\text{int}}$.

规则 18 表示设计模型的固有约束集合 $\phi_{\text{intrinsic}}$ 映射为 UIR 的固有约束集合 uir_C_{int}. 映射过程将每个固有约束 $\phi_{\text{int}_i} \in \phi_{\text{intrinsic}}$ 转化为对应的 UIR 约束 $uir\_C_{\text{int}_i}$, 并保证约束的标识符、类型、表达式、来源以及关联属性等信息. 映射结果满足如下约束.

- $\phi_{\text{int}_i}.id = uir\_C_{\text{int}_i}.id$;
- $\phi_{\text{int}_i}.type = uir\_C_{\text{int}_i}.type$;
- $\phi_{\text{int}_i}.expression = uir\_C_{\text{int}_i}.expression$;
- $uir\_C_{\text{int}_i}.source =$ "$design\_model$";
- $uir\_C_{\text{int}_i}.relatedActions = \phi_{\text{int}_i}.related\_actions$;
- $uir\_C_{\text{int}_i}.relatedStates = \phi_{\text{int}_i}.related\_states$;
- $uir\_C_{\text{int}_i}.relatedHazards = \varnothing$.

其中, $\phi_{\text{int}_i}.related\_actions$ 表示设计模型中固有约束直接关联的动作集合; $\phi_{\text{int}_i}.related\_states$ 表示设计模型中固有约束直接关联的状态集合.

规则 19. STAMP 控制器约束映射规则: $Trans_{\text{ctrl}}(\phi_{\text{STAMP}}) := uir\_C_{\text{ctrl}}$.

规则 19 表示 STAMP 控制器集合的映射规则. STAMP 分析的控制器集合 ϕ_{STAMP} 映射为 UIR 的控制器约束集合 uir_C_{ctrl}. 映射过程将每个控制器 $\phi_{\text{ctrl}_i} \in \phi_{\text{STAMP}}$ 转换为对应的 UIR 约束 $uir\_C_{\text{ctrl}_i}$, 并保证控制器的安全控制逻辑、受控危险以及控制动作等信息. 映射结果满足约束:

- $\phi_{\text{ctrl}_i}.id = uir\_C_{\text{ctrl}_i}.id$;
- $uir\_C_{\text{ctrl}_i}.type =$ "$safety\_control$";
- $uir\_C_{\text{ctrl}_i}.expression = G_{\text{ctrl}}(c_i)$;
- $uir\_C_{\text{ctrl}_i}.source =$ "$STAMP\_controller$";
- $uir\_C_{\text{ctrl}_i}.relatedHazards = c_i.controlled\_hazards$;
- $uir\_C_{\text{ctrl}_i}.relatedActions = c_i.controlled\_actions$;
- $uir\_C_{\text{ctrl}_i}.relatedStates = c_i.control\_states$.

其中, $G_{\text{ctrl}}(c_i)$ 表示控制器安全约束生成函数, 根据控制器 ϕ_{ctrl_i} 的控制逻辑和安全责任, 自动生成相应的形式化安全约束表达式, 确保控制器在所有操作条件下产生安全的控制输出.

5.2.2 中间层到验证层模型属性映射

通过解析 UIR 中的约束集合、危险场景和时间属性实现 UIR 到验证属性的映射, 并按照验证工具的属性规范进行转换实现. 定义通用的映射算法 (算法 3) 系统化地完成 UIR 到验证模型属性的转换过程.

算法 3. UIR 到验证模型属性的映射.

输入: UIR 约束 $uir\_C =< id, type, expression, source, relatedHazards, relatedActions, relatedStates >$; 目标验证模型属性语言集合 $ToolSet$ = {CTL, TCTL, PCTL, Promela, TLA+};

输出: 多语言验证属性集合 $VM\_properties =< vm\_Q\_tool \mid tool \in ToolSet >$.

1. **for each** $tool \in ToolSet$ **do**
2. $syntax\_analyzer \leftarrow get\_syntax\_analyzer(tool)$
3. $constraint\_template \leftarrow parse\_UIR\_constraint(uir\_C, syntax\_analyzer)$
4. **switch**(*tool*):
5. **case** CTL: $vm\_Q\_CTL \leftarrow T\_CTL(constraint\_template)$
6. **case** TCTL: $vm\_Q\_TCTL \leftarrow T\_TCTL(constraint\_template, extract\_time\_bound(uir\_C))$
7. **case** PCTL: $vm\_Q\_PCTL \leftarrow T\_PCTL(constraint\_template, extract\_probability(uir\_C))$

8. **case** Other: *vm_Q_Other* ← *T_Other*(*constraint_template*,*tool_specific_params*)
9. *VM_properties.add*(*vm_Q_tool*)
10. **end for**
11. **return** ***VM_properties***

从 UIR 到不同验证工具约束语法的转换是属性映射的核心, 需要根据目标验证工具的特性选择合适的转换策略.

规则 20. UIR 约束到 CTL 的转换规则: $Trans_{\mathrm{CTL}}(uir\_C) := vm\_Q_{\mathrm{CTL}}$.

规则 20 表示 UIR 统一约束到计算树逻辑 (computation tree logic, CTL) 的转换规则. 转换过程基于 UIR 表达式的语法结构, 将统一约束表达式转换为 CTL 时态逻辑公式, 映射结果满足约束: $vm\_Q_{\mathrm{CTL}} = \mathfrak{I}_{\mathrm{CTL}}(uir\_C.expression)$, 若 $uir\_C.type \in \{safety\_control\}$, 则 $vm\_Q_{\mathrm{CTL}} = AG(\phi_{\mathrm{safety}})$; 若 $uir\_C.type \in \{\mathrm{functional}, \mathrm{performance}\}$, 则 $vm\_Q_{\mathrm{CTL}} = AF(\phi_{\mathrm{goal}}) \vee AG(\phi_{\mathrm{invariant}})$; 若 $uir\_C.type \in \{\mathrm{structural}, \mathrm{behavioral}\}$, 则 $vm\_Q_{\mathrm{CTL}} = AG(\phi_{\mathrm{structure}})$. 其中, $\mathfrak{I}_{\mathrm{CTL}}(expr)$ 表示 UIR 表达式到 CTL 公式的语法转换函数; $AG(\cdot)$ 表示 CTL 中的“在所有路径上总是成立”算子; $AF(\cdot)$ 表示“在所有路径上最终成立”算子; ϕ_{safety}、ϕ_{goal}、$\phi_{\mathrm{invariant}}$、$\phi_{\mathrm{structure}}$ 分别表示安全性谓词、目标谓词、不变式谓词和结构性谓词.

$$\mathfrak{I}_{\mathrm{CTL}}(expr) = \begin{cases} AG(\neg hazard\_state), & \text{if } expr \text{ is safety constraint} \\ AF(goal\_state), & \text{if } expr \text{ is liveness constraint} \\ AG(invariant), & \text{if } expr \text{ is invariant constraint} \\ A[precond\,\mathcal{U}\,postcond], & \text{if } expr \text{ is until constraint} \end{cases}$$

其中, *hazard_state* 表示危险状态谓词; *goal_state* 表示目标状态谓词; *invariant* 表示系统不变式; *precond* 和 *postcond* 分别表示前置条件和后置条件; $\mathcal{U}$ 表示 CTL 的“直到”算子; $A(\cdot)$ 表示在“在所有路径上”的路径谓词.

规则 21. UIR 约束到 TCTL 的转换规则: $Trans_{\mathrm{TCTL}}(uir\_C) := vm\_Q_{\mathrm{TCTL}}$.

规则 21 表示 UIR 统一约束到时间计算树逻辑 (timed computation tree logic, TCTL) 的转换规则. UIR 约束 *uir_C* 映射为 TCTL 公式 vm_Q_{TCTL}. 转换过程包含将时间属性的 UIR 约束转换为带时间界限的 TCTL 公式, 映射结果需满足约束: $vm\_Q_{\mathrm{TCTL}} = \mathfrak{I}_{\mathrm{TCTL}}(uir\_C.expression)$, 若约束包含截止时间 *deadline*, 则 $vm\_Q_{\mathrm{TCTL}} = AG_{\leqslant deadline(\varnothing)}$; 若约束包含响应时间 *response*, 则 $vm\_Q_{\mathrm{TCTL}} = AG(req \Rightarrow AF_{\leqslant response}(resp))$; 若约束包含周期性 *period*, 则 $vm\_Q_{\mathrm{TCTL}} = AG(trigger \Rightarrow AF_{\leqslant period}(AX(trigger)))$. 其中 $\mathfrak{I}_{\mathrm{TCTL}}(expr)$ 表示带时间界限的转换函数; $AG_{\leqslant t}(\cdot)$ 表示“在时间界限 t 内总是成立”的时间约束算子; $AF_{\leqslant t}(\cdot)$ 表示“在时间界限 t 内最终成立”的时间约束算子; *deadline* 表示任务截止时间; *response* 表示系统响应时间界限; *period* 表示周期性约束的时间周期; *req* 表示请求条件; *resp* 表示响应条件; *trigger* 表示出发条件; $AX(\cdot)$ 表示“下一个状态”算子; $\Rightarrow$ 表示逻辑蕴含关系.

$$\mathfrak{I}_{\mathrm{TCTL}}(expr,t) = \begin{cases} AG_{\leqslant t}(\neg hazard), & \text{if safety with deadline } t \\ AF_{\leqslant t}(goal), & \text{if reachability with deadline } t \\ AG(req \Rightarrow AF_{\leqslant t}(resp)), & \text{if response with bound } t \end{cases}$$

规则 22. UIR 约束到 PCTL 的转换规则: $Trans_{\mathrm{PCTL}}(uir\_C) := vm\_Q_{\mathrm{PCTL}}$.

规则 22 表示 UIR 统一约束到概率计算树逻辑 (probabilistic computation tree logic, PCTL) 的转换规则. UIR 约束 *uir_C* 映射为 PCTL 公式 vm_Q_{PCTL}. 转换过程将包含概率信息的 UIR 约束转换为概率时态逻辑公式, 映射结果满足约束: $vm\_Q_{\mathrm{PCTL}} = \mathfrak{I}_{\mathrm{PCTL}}(uir\_C.expression)$, 若约束为概率安全性, 则 $vm\_Q_{\mathrm{PCTL}} = P_{\geqslant p}[G(\neg hazard)]$, 若约束为概率可达性, 则 $vm\_Q_{\mathrm{PCTL}} = P_{\geqslant p}[F(goal)]$; 若约束为概率响应性, 则 $vm\_Q_{\mathrm{PCTL}} = P_{\geqslant p}[G(req \Rightarrow F(resp))]$. 其中, $\mathfrak{I}_{\mathrm{PCTL}}(expr)$ 表示带概率界限的转换函数; $P_{\geqslant p}[\cdot]$ 表示“概率至少为 p”的概率算子; p 表示概率阈值, $p \in [0,1]$; $G(\cdot)$ 表示“全局成立”算子; $F(\cdot)$ 表示“最终成立”算子.

$$\mathfrak{I}_{\mathrm{PCTL}}(expr,p) = \begin{cases} P_{\geqslant p}[G(\neg hazard)], & \text{if probabilistic safety with bound } p \\ P_{\geqslant p}[F(goal)], & \text{if probabilistic reachability with bound } p \\ P_{\geqslant p}[G(req \Rightarrow F(resp))], & \text{if probabilistic response with bound } p \end{cases}$$

规则 23. UIR 约束到其他验证语言的转换规则: $Trans_{\text{Other}}(uir\_C) := vm\_Q_{\text{Other}}$.

规则 23 表示 UIR 统一约束到其他特定验证工具语言的转换规则, 包括 Promela、TLA+等. 转换过程根据目标工具的语法规范, 将 UIR 约束转换为相应的规约语言. 对于 Promela 语言, 满足约束: $vm\_Q_{\text{Promela}} = \Im_{\text{Promela}}(uir\_C) = assert(\phi) \vee ltl(\varphi)$; 对于 TLA+语言, $vm\_Q_{\text{TLA+}} = \Im_{\text{TLA+}}(uir\_C) =$ THEOREM Spec $\Rightarrow$ Property. 其中 $\Im_{\text{Promela}}(\cdot)$、$\Im_{\text{TLA+}}(\cdot)$ 分别表示到 Promela、TLA+语言的转换函数; $assert(\phi)$ 表示 Promela 中的断言语句; $ltl(\varphi)$ 表示 Promela 中的线性时态逻辑属性; THEOREM 表示 TLA+中的定理声明关键字; Spec 表示 TLA+规约; Property 表示待验证的属性.

5.3 转换正确性检查

5.3.1 设计模型到 UIR 转换的正确性检查

对生成的 UIR 模型进行无歧义性、完备性、语义一致性检查以确保模型转换正确. 具体说明如下.

(1) 无歧义性检查: 每个设计模型及 STAMP 安全约束在映射到 UIR 时有且仅有一个对应的 UIR 元素, 确保映射规则 $Trans_1$、$Trans_2$、$Trans_3$、$Trans_4$、$Trans_5$ 以及 $Trans_{\text{STAMP}}$ 是单射的. 即对于任意的设计模型元素 $c_1, c_2 \in DM$, 若 $c_1 \neq c_2$, 则其映射结果 $Trans_i(c_1) \neq Trans_i(c_2)$. 特别地, 需验证 STAMP 自动生成的约束表示与原设计约束不重复.

(2) 完备性检查: 设计模型中的所有核心元素、交互关系和附加信息都能被映射到 UIR 中, 确保映射规则的覆盖性. 即对于设计模型中的所有元素集合 E_M、E_S、E_T、E_V、E_C 以及 STAMP 元素集合 E_{UC}、E_L、$E_{\text{Constraints}}$、E_P, 有 $Trans_i(E_M) \subseteq Dom(UMR)$、$Trans_i(E_S) \subseteq Dom(UMR)$、$Trans_i(E_T) \subseteq Dom(UMR)$、$Trans_i(E_V) \subseteq Dom(UMR)$、$Trans_i(E_C) \subseteq Dom(UMR)$、$Trans_{\text{STAMP}}(E_{\text{UC}}) \subseteq Dom(UMR)$、$Trans_{\text{STAMP}}(E_L) \subseteq Dom(UMR)$、$Trans_{\text{STAMP}}(E_{\text{Constraints}}) \subseteq Dom(UMR)$、$Trans_{\text{STAMP}}(E_P) \subseteq Dom(UMR)$.

(3) 语义一致性检查: 映射能够保留设计模型的语义, 保证 UIR 中的元素能够完整表达设计模型的行为和结构, 即需要检验正向映射的语义一致性与逆向映射的可还原性. 即 $\forall e \in DM$, $Trans^{-1}(Trans_i(E_j)) = E_j$. STAMP 分析产生的安全约束在 UIR 中得到完整表达, 且其逻辑语义与原 STPA 分析结果等价. 即对于每个不安全控制行为 UC_i 及其对应的 UIR 约束 $UIR\_c_{UC_i}$, 需满足: $Semantic(uc_i) \equiv Semantic(UIR\_c_{uc_i})$.

(4) 互模拟等价检查[6,7]: 在语义一致性检查的基础上, 进一步进入“互模拟等价 (bisimulation equivalence)”检查, 以验证设计模型到 UIR 的转换策略的可信性.

定义 16. 两个状态系统是互模拟等价的可以表示为 $M_1 \sim M_2$, 其中, $M_1 = (S_1, T_1)$ 表示第 1 个系统; $M_2 = (S_2, T_2)$ 表示第 2 个系统 $R \subseteq S_1 \times S_2$ 表示一个关系; 前向模拟: 若 $s_1 \xrightarrow{a} s_1'$ 且 $(s_1, s_2) \in R$, 则存在 $s_2 \xrightarrow{a} s_2'$ 且 $(s_1', s_2') \in R$; 后向模拟: 若 $s_2 \xrightarrow{a} s_2'$ 且 $(s_1, s_2) \in R$, 则存在 $s_1 \xrightarrow{a} s_1'$ 且 $(s_1', s_2') \in R$.

具体地, 对于设计模型 M_{dg}, 定义其状态集合为 $S_{\text{DM}} = \{s_1, s_2, \ldots, s_n\}$, 转换集合为 $T_{DM} = \{(s_1, a, s_2) \mid s_i, s_j \in S_{\text{DM}}, a \in A_{\text{DM}}\}$; UIR 模型为 UIR, 相应的状态集合为 $S_{\text{UIR}} = \{uir\_s_1, uir\_s_2, \ldots, uir\_s_n\}$, 转换集合为 $T_{\text{UIR}} = \{(uir\_s_i, a, uir\_s_j) \mid (uir\_s_i, uir\_s_j \in S_{\text{UIR}}, a \in A_{\text{UIR}})\}$; 定义互模拟关系 $R \subseteq S_{\text{DM}} \times S_{\text{UMR}}$, 且满足 $R = \{(s_i, uir\_s_i) \mid s_i \in S_{\text{DM}}, uir\_s_i \in S_{\text{UIR}}\}$.

(1) 前向模拟

假设 $s_i \xrightarrow{a} s_j$ 是设计模型中的一个转换, 即 $(s_i, a, s_j) \in T_{\text{DM}}$; 根据设计模型到 UIR 的映射规则, s_i 和 s_j 分别映射为 uir_s_i 和 uir_s_j, 且转换 $s_i \xrightarrow{a} s_j$ 映射为 $uir\_s_i \xrightarrow{a} uir\_s_j$; 因此, 存在 $uir\_s_i \xrightarrow{a} uir\_s_j$, 且 $(s_i, uir\_s_j) \in S_{\text{UIR}}$.

(2) 后向模拟

假设 $uir\_s_i \xrightarrow{a} uir\_s_j$ 是 UIR 模型中的一个转换, 即 $(uir\_s_i, a, uir\_s_j) \in T_{\text{UIR}}$; 根据 UIR 到设计模型的映射规则, uir_s_i 和 uir_s_j 分别映射为 s_i 和 s_j, 且转换 $uir\_s_i \xrightarrow{a} uir\_s_j$ 映射为 $s_i \xrightarrow{a} s_j$; 因此, 存在 $s_i \xrightarrow{a} s_j$, 且 $(s_j, uir\_s_j) \in R$.

由此可得, 设计模型 M_{dg} 和 UIR 模型 UIR 是互模拟等价的, 即 $M_{dg} \sim UIR$.

5.3.2 中间层到验证模型转换的正确性检查

对转换后的模型进行无歧义性、完备性以及语义一致性检查, 以确保从 UIR 到验证模型的转换正确性.

(1) 无歧义性检查: 每个 UIR 元素在映射到验证模型时有且仅有一个对应的验证模型元素, 确保映射规则 $Trans_6$、$Trans_7$、$Trans_8$、$Trans_9$ 以及 STAMP 相关的映射规则 $Trans_{\text{STAMP}_5}$、$Trans_{\text{STAMP}_6}$、$Trans_{\text{STAMP}_7}$ 是单射的. 即

对于任意 $e_1, e_2 \in UIR$, 若 $Trans_i(e_1) = Trans_i(e_2)$, 则 $e_1 = e_2$.

(2) 完备性检查: UIR 中的所有核心元素都能被映射到验证模型中, 确保映射规则的覆盖性. 即对于 UIR 中的所有元素集合 uir_M、uir_S、uir_T、uir_V、uir_C、uir_D、uir_P 都能被映射到验证模型, 保证证每个 UIR 元素都能被映射规则覆盖.

(3) 语义一致性检查: 确保验证模型中的状态变量、转移规则、时间约束和安全约束能够完整表达 UIR 行为和性质. 验证映射后的验证模型是否能够通过逆向映射还原为 UIR 模型, 即 $Trans_6^{-1}$、$Trans_7^{-1}$、$Trans_8^{-1}$、$Trans_9^{-1}$、$Trans_{\text{STAMP}_5}^{-1}$、$Trans_{\text{STAMP}_6}^{-1}$、$Trans_{\text{STAMP}_7}^{-1}$ 存在且可用. 同时, 通过验证模型的执行结果与 UIR 的预期行为进行对比, 确保语义一致性.

(4) 互模拟等价检查: 定义 UIR 模型为 UIR, 状态集合为 $S_{\text{UIR}} = (uir\_s_1, uir\_s_2, \ldots, uir\_s_n)$, 转换集合为 $T_{\text{UIR}} = \{(uir\_s_i, a, uir\_s_j) \mid uir\_s_i, uir\_s_j \in S_{\text{UIR}}, a \in A_{\text{UIR}}\}$; 定义验证模型 VM, 相应的状态集合为 $S_{\text{VM}} = (vm\_s_1, vm\_s_2, \ldots, vm\_s_n)$, 转换集合为 $T_{\text{VM}} = \{(vm\_s_i, a, vm\_s_j) \mid vm\_s_i, vm\_s_j \in S_{\text{VM}}, a \in A_{\text{VM}}\}$; 定义互模拟关系 $R \subseteq S_{\text{UIR}} \times S_{\text{VM}}$, 且满足 $R' = \{(uir\_s_i, vm\_s_i) \mid uir\_s_i \in S_{\text{UIR}}, vm\_s_i \in S_{\text{VM}}\}$. 通过前向模拟和后向模拟验证 UIR 模型 UIR 和验证模型 VM 和是互模拟等价的, 即 $UIR \sim VM$.

通过构建具备模块化与增量式理念的映射规则, 为各类系统模型中最常见的元模型元素定义了一套核心映射规则库, 涵盖了状态、转换、变量、约束等基础元素的通用转换. 同时分层模型转换机制亦支持增量扩展, 当需要引入一个新的设计语言或验证后端时, 只需针对其特有的语法元素和语义特征, 增量地开发并补充一组特定的映射规则, 以可插拔方式集成到现有框架中即可, 显著降低了扩展的复杂度.

6 多重交叉验证机制

为了全面验证系统的功能正确性与安全约束满足性, 同时克服单一验证工具在覆盖范围和验证深度上的局限性. 本文提出多重交叉验证机制. 从 UIR 出发自动转换为多种功能互补的验证模型 (时序验证模型、逻辑验证模型、概率验证模型), 将源于设计层 STPA 安全风险分析的安全约束系统化地转换为模型中的可验证属性, 并通过模型间的一致性检查、STPA 安全约束一致性校验以及验证结果的互补性分析, 实现多维度、多层次的交叉验证.

6.1 多类型验证

通过映射规则将 UIR 生成多个验证模型 (如时序验证模型、逻辑验证模型和概率验证模型等), 实现互补验证和一致性检查, 如后文图 5 所示.

定义 17. 验证模型的生成过程可表示为 $VM_i = f_i(UIR)$, $i \in \{1, 2, \ldots, n\}$, 其中, $VM_i = <vm_S^i, vm_T^i, vm_P^i, vm_Q^i, vm_{\text{STAMP}}^i>$ 表示生成的第 i 个验证模型; f_i 表示从 UIR 到特定验证模型的映射函数; vm_{STAMP}^i 表示嵌入到第 i 个验证模型中的 STAMP 安全约束集合. 每个映射函数 f_i 的定义依赖于 UIR 的核心元素和其在目标模型中的语义映射规则, 并统一集成 STAMP 安全约束作为待验证的安全属性.

6.1.1 时序验证模型

通过 UIR 中的状态集合 S、转换集合 T 和时间约束集合 P, 生成适配于时序验证工具的验证模型, 用于验证系统的实时性和时序行为.

定义 18. 时序验证模型的映射规则可以表示为 $f_{\text{timing}}(UIR) :=< vm_S, vm_T, vm_P, vm_{\text{STPA}}^{\text{timing}} >$, 其中,

- $vm_S = \{s_i \mid s_i \in uir_S, \forall s_i \in S, \exists t \in T, Pre(t) = s_i \vee Post(t) = s_i\}$ 表示生成时序验证模型状态集合的规则; s_i 表示 UIR 模型中的某个状态; uir_S 表示 UIR 模型中的状态集合; $Pre(t)$ 表示转换 t 的起始状态; $Post(t)$ 表示转换 t 的目标状态.
- $vm_T = \{t_j \mid t_j = (s_i, s_k, \lambda, \alpha), \forall t_j \in uir_T, \lambda \in Cond, \alpha \in Act\}$ 表示生成时序验证模型转换集合的规则; $t_j = (s_i, s_k, \varnothing, a)$ 表示一个转换 t_j, 包括起始状态 s_i、目标状态 s_k、转换条件λ和转换动作 α; $Cond$ 表示条件集合; Act 表示动作集合.
- $vm_P = \{p_k \mid p_k \in uir_P, Scope(p_k) \subseteq vm_S\}$ 表示生成时序验证模型时间约束集合的规则; p_k 表示 UIR 模型中的某个时间约束; uir_P 表示 UIR 模型中的时间约束集合; $Scope(p_k)$ 表示时间约束 p_k 的作用范围.

● $vm_{\text{STPA}}^{\text{timing}} = \{stpa\_l \mid stpa\_l \in uir\_C_{\text{STPA}}, Convert(stpa\_l) \rightarrow TL\_Formula\}$ 表示将 STPA 安全约束转换为时序逻辑公式的规则; *TL_Formula* 表示时序逻辑公式集合; 函数 $Convert(\cdot)$ 将 STPA 约束转换为相应的时序逻辑表达式.

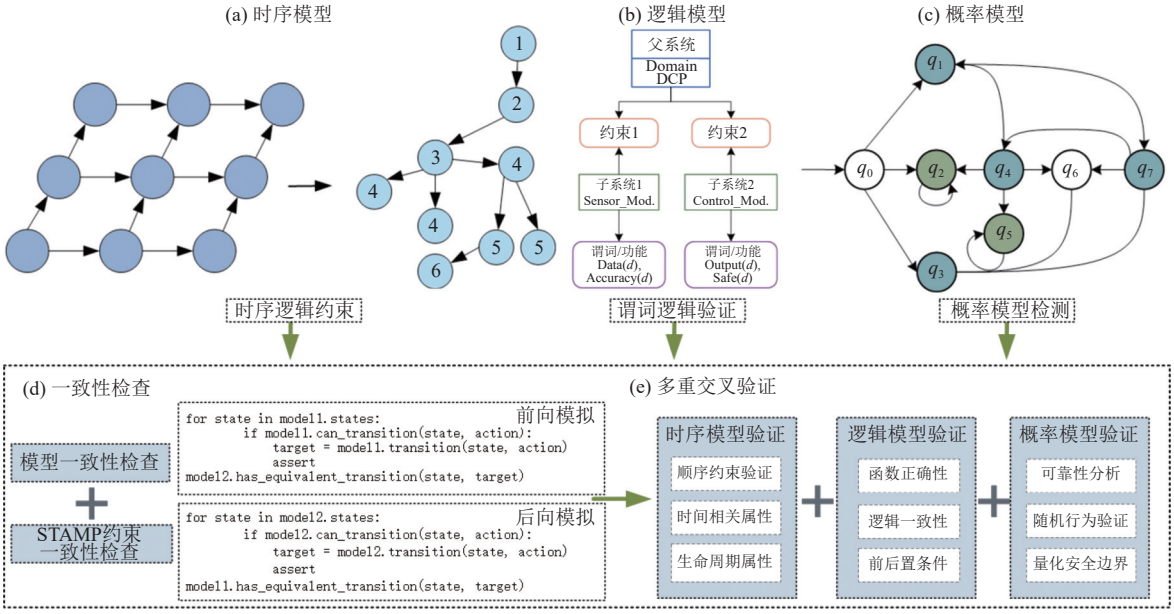

图 5 多重交叉验证机制

6.1.2 逻辑验证模型

通过映射 UIR 中的变量集合 V、约束集合 C 和安全属性集合 Q, 生成适配逻辑验证工具的验证模型, 用于验证系统的逻辑一致性和安全性.

定义 19. 逻辑验证模型的映射规则可以表示为 $f_{\text{logic}}(UIR) :=< vm_V, vm_C, vm_Q, vm_{\text{STPA}}^{\text{logic}} >$, 其中,

● $vm_V = \{v_i \mid v_i = (name, type, range, init), \forall v_i \in uir_V\}$ 表示生成的第 i 个验证模型; v_i 表示逻辑验证模型中的某个变量; *name* 表示变量的名称; *type* 表示变量的类型; *range* 表示变量的取值范围; *init* 表示变量的初始值.

● $vm_C = \{c_j \mid c_j = (\mathbb{C}, scope), \mathbb{C} \in Expr, scope \subseteq vm_V\}$; c_j 表示逻辑验证模型中的某个约束; $\mathbb{C}$ 表示约束的具体条件; *scope* 表示约束的作用范围; *Expr* 表示逻辑表达式集合; $scope \subseteq vm_V$ 表示约束的作用范围是逻辑验证模型中的变量集合的子集;

● $vm_Q = \{q_k \mid q_k = (\psi, priority), \psi \in SafetyExpr, priority \in \mathbb{N}\}$; q_k 表示逻辑验证模型中的某个安全属性; ψ 表示系统需要满足的安全性条件; *priority* 表示安全属性的优先级; *SafetyExpr* 表示安全属性表达式集合.

● $vm_{\text{STPA}}^{\text{logic}} = \{stpa\_m \mid stpa\_m \in uir\_C_{\text{STPA}}, Convert(stpa\_m) \rightarrow Logic\_Invariant\}$ 表示将 STPA 安全约束转换为逻辑不变式的规则; *Logic_Invariant* 表示逻辑不变式集合.

6.1.3 概率验证模型

通过扩展 UIR 中的状态集合 S 和转换集合 T, 引入概率转移矩阵 $\mathbf{P}$ 和随机变量 $\mathbf{X}$, 同时将 STPA 安全约束转换为概率逻辑属性, 用于概率验证工具的验证, 以此来验证系统的可靠性和随机行为以及概率安全性.

定义 20. 概率验证模型的映射规则可以表示为 $f_{\text{prob}}(UIR) :=< vm_S, vm_T, \mathbf{P}, \mathbf{X}, vm_{\text{STPA}}^{\text{prob}} >$, 其中,

● $\mathbf{P} = [p_{ij}], p_{ij} = P(s_i \xrightarrow{t} s_j), \forall s_i, s_j \in uir_S, \sum_j p_{ij} = 1$ 表示生成的第 i 个验证模型; p_{ij} 表示矩阵的元素, 表示从状态 s_i 转移到状态 s_j 的概率; $P(s_i \xrightarrow{t} s_j)$ 表示从状态 s_i 经转换 t 转移到状态 s_j 的概率; $\sum_j p_{ij} = 1$ 表示每个状态的转移概率总和为 1.

● $\mathbf{X} = \{x_k \mid x_k \in uir_V, type(x_k) = RandomVar\}$; x_k 表示 UIR 模型中的某个随机变量; uir_V 表示 UIR 模型中的变量

集合; $type(x_k) = RandomVar$ 表示变量 x_k 的类型是随机变量.

- $vm^{prob}_{STAMP} = \{stamp\_n \mid stamp\_n \in uir\_C_{STAMP}, Convert(stamp\_n) \rightarrow Prob\_Property\}$ 表示将 STAMP 安全约束转换为概率属性的规则; $Prob\_Property$ 表示概率属性集合.

6.2 动态交叉验证

时序、逻辑和概率等模型生成后, 针对不同模型的验证结果进行交叉验证, 使验证结果相互印证与补充. 本框架在 STPA 分析的基础上, 通过形式化验证工具提供属性验证的第 2 道防线. 当输入的性质规约 (LTL/CTL 等公式) 存在逻辑冲突时, 验证器会返回“不满足”并通常提供反例, 从而精确标识出相互矛盾的约束条款. 这实现了对约束冲突的自动化、精确化检测. 本框架要求所有约束必须被同时满足, 任何违反都将产生反例.

6.2.1 验证模型一致性检查

验证模型之间的一致性通过前向模拟和后向模拟进行检查. 定义验证模型 $VM_i =< vm^i_S, vm^i_T, vm^i_{STAMP} >$ 和 $VM_j =< vm^j_S, vm^j_T, vm^j_{STPA} >$, 若存在互模拟关系 $\mathcal{R} \subseteq vm^i_S \times vm^j_S$, 则满足:

- 前向模拟: 若 $(s_i, s_j) \in \mathcal{R}$, 且 $s_i \xrightarrow{t_i} s'_i$, 则存在 $s_j \xrightarrow{t_j} s'_j$ 使得 $(s'_i, s'_j) \in \mathcal{R}$;
- 后向模拟: 若 $(s_i, s_j) \in \mathcal{R}$, 且 $s_j \xrightarrow{t_j} s'_j$, 则存在 $s_i \xrightarrow{t_i} s'_i$, 使得 $(s'_i, s'_j) \in \mathcal{R}$.

若上述条件同时满足, 则验证模型 VM_i 和 VM_j 是互模拟等价的, 验证结果一致. 其中 $s_i \xrightarrow{t_i} s'_i$ 表示状态 s_i 经转换 t_i 转移到状态 s'_i.

6.2.2 STPA 安全约束一致性检查

对于每个状态对 $(s_i, s_j) \in \mathcal{R}$, 其相关的 STPA 安全约束需保持语义等价:

$$\forall stpa_i \in Active(s_i, vm^i_{STPA}),\ \exists stpa_j \in Active(s_j, vm^j_{STPA}):\ Semantic(stpa_i) \equiv Semantic(stpa_j),$$

其中, $Active(s, vm_{STPA})$ 表示在状态 s 下激活的 STPA 安全约束集合.

6.2.3 验证结果的一致性检查

交叉验证方法针对对不同模型之间的结果可靠性进行一致性检查. 定义验证模型之间的互模拟关系 $R \subseteq vm^i_S \times vm^j_S$, 如果两个验证模型 VM_i 和 VM_j 满足条件:

$$\forall (s_i, s_j) \in \mathcal{R}, s_i \xrightarrow{t_i} s'_i \Rightarrow \exists s'_j \in vm^j_S, s_j \xrightarrow{t_j} s'_j \wedge (s'_i, s'_j) \in \mathcal{R},$$

同时满足 STPA 安全约束验证结果一致性:

$$\forall stpa \in STPA_{constraints}:\ Verify_i(stpa) \equiv Verify_j,$$

则可认为其验证结果一致. 其中, $Verify_i(stpa)$ 表示验证模型 VM_i 对 STPA 安全约束 $stpa$ 的验证结果, $\xrightarrow{t}$ 表示状态间的转移关系; $\Rightarrow$ 是逻辑蕴含关系.

6.2.4 验证结果的互补性

将不同验证模型的结果进行交叉验证, 以避免单一模型存在的验证盲区. 即 $R = \bigcup_{i=1}^{n} R_i \cup R_{STAMP}$, $R_i = Verify(VM_i)$, 其中 R_i 表示第 i 个验证模型的验证结果, R 表示综合验证结果, $R_{STAMP} = \bigcup_{i=1}^{n} Verify_{STAMP}(VM_i)$ 表示所有验证模型中 STAMP 安全约束的验证结果集合. 以 STPA 的系统性分析作为先导预防, 以形式化验证的自动化检测作为事后保障, 通过交叉对比所有模型的验证结果, 确保验证约束在时序、逻辑、概率等多个维度上的一致性和互补性, 强化系统安全验证的可信性和鲁棒性.

7 可扩展性分析

从计算复杂度理论角度, 对本文提出的基于统一中间表示 (UIR) 的异构模型转换与验证框架进行可扩展性分析, 证明该方法具备处理大规模系统的理论潜力.

7.1 分析框架与基本假设

为了准确分析本文方法的可扩展性, 我们首先建立系统化的分析框架. 采用随机访问机 (random access

machine, RAM) 作为计算模型. 在该模型下, 我们假设每个基本操作 (如赋值、比较、算术运算等) 可在常数时间内完成. 我们以设计模型中元素数量 n 作为系统规模的主要度量指标. 这里的“元素”包括状态、转移、变量、组件等所有需要被处理和验证的实体.

基于本文方法的核心特征, 我们设定如下基本假设.

- 转换局部性: UIR 转换过程具有局部性特性, 每个设计元素的处可以独立或半独立进行, 产生 O(1) 数量的元素.
- 结构保持性: 转换过程保持系统的模块化层次结构, 避免产生全局性的组合爆炸.
- 验证正交性: 多重验证过程可以相对独立地进行, 其复杂度具有可加性.

这些假设在实际工程实践中合理且可实现, 特别是在具有良好架构设计的大型系统中.

7.2 各组件复杂度分析

通过分析 UIR 转换、模型派生以及验证过程的复杂度, 评估本文方法的整体计算复杂度.

7.2.1 UIR 转换过程复杂度分析

定理 1. 线性转换复杂度. 设设计模型规模为 n, UIR 转换算法的时间复杂度为 $T_{\text{trans}}(n) = \text{O}(n)$.

证明: UIR 转换过程的核心思想是将复杂的设计模型分解为相对独立的元素集合, 然后通过规则驱动的方法将每个元素映射到统一的中间表示. 这个过程可以形式化描述如下.

对于设计模型中的每个元素 e_i ($i = 1,2,\ldots,n$), 转换算法执行以下操作:

- 解析元素 e_i 的语义信息.
- 根据预定义的转换规则集, 生成对应的 UIR 元素.
- 建立元素间的追踪链接.

由于每个元素的处理时间基于独立于系统总规模, 且处理过程中涉及的操作数量有上限 (由转换规则集的有限性保证), 因此每个元素的处理时间为常数 O(1). 整体转换时间可表示为:

$$T_{\text{trans}}(n) = \sum_{i=1}^{n} \text{O}(1) = \text{O}(n).$$

这种线性复杂度的实现基于本文设计的增量式转换机制和规则映射方法, 避免了传统方法中常见的全局分析和重构过程.

7.2.2 模型派生过程复杂度分析

定理 2. 参数化派生复杂度. 从 UIR 派生验证模型的时间复杂度取决于目标模型的特征维度 d ($d \geqslant 1$), 满足:

$$T_{\text{derive}}(n) = \text{O}(n^d).$$

证明: 模型派生过程的复杂度不是固定的, 而是与目标验证模型的语义维度和结构特征密切相关. 我们引入维度参数 d 来刻画这种关系.

- 一维模型派生 ($d = 1$): 适用于结构简单的验证模型, 如基本状态机或简单的逻辑公式. 派生过程主要是元素间的一对一或一对多映射, 复杂度为 O(n);
- 二维模型派生 ($d = 2$): 适用于需要处理二元关系的模型, 如时间自动机 (状态迁移关系) 等需要处理元素间的两两关系, 复杂度为 O(n^2);
- 高维模型派生 ($d \geqslant 3$): 适用于需要分析多元关系的模型, 如性能模型 (多资源竞争) 等, 复杂度可能达到 O(n^3) 或更高.

值得注意的是, 虽然理论上 d 可能很大, 但在实际应用中, UIR 提供的结构化信息和语义约束避免了派生过程中最坏情况下的组合爆炸.

推论 1. 对于本文涉及的主要验证模型, 派生复杂度满足:

$$T_{\text{derive}}(n) = \text{O}(n^3).$$

7.2.3 验证过程复杂度

定理 3. 验证复杂度分解. 多重验证过程的总体时间复杂度可分解为:

$$T_{\text{verify}}(n)=\sum_{i=1}^{k}T_i(n)+T_{\text{coord}}(n),$$

其中, $T_i(n)$ 是第 i 种验证方法的复杂度, $T_{\text{coord}}(n)$ 是协调验证结果的复杂度.

分层复杂度主要来源于基础验证复杂度和协调复杂度, 具体说明如下.

- 基础验证复杂度取决于单个验证方法的范围, 可包括模拟测试、模型检测以及定理证明等.
- 协调过程主要包括验证结果的交叉检查和冲突消解, 复杂度主要来自结果间的比较和整合.

综上, 总体复杂度是由最复杂的单个验证方法和协调复杂度共同决定:

$$T_{\text{verify}}(n)=\mathrm{O}(\max_i T_i(n))+\mathrm{O}(n^2).$$

定理 4. 整体复杂度上界. 在合理假设下, 本文方法的总体时间复杂度上界为:

$$T_{\text{total}}(n)=\mathrm{O}(n^c),$$

其中, $c=\max(3,d,e)$, d 是派生复杂度维度, e 是验证复杂度维度. 此上界为保守估计, 因为不同阶段不会同时达到最坏情况, UIR 的结构化信息有效约束了复杂度的增长, 同时实际系统的模块化特性也进一步降低了复杂度.

8 统一中间表示转换平台原型系统实现

统一中间表示转换平台选用 Python 作为开发语言, 并在 VSCode 环境中完成编码. 软件架构设计分为 3 层结构: 界面层、处理逻辑层和工具调用层, 如图 6 所示. 其中, 界面层主要包含基于 RestfulAPI 搭建的 API 服务以及网页端访问界面, 规模约 800+行, 处理逻辑层主要包含设计解析、UIR 处理和模型验证这 3 个模块. 设计解析模块集成包含 SysML、Stateflow、UPPAAL、Modelica 以及 XSTAMPP 模型的解析功能, 并基于从设计模型中提取的状态、状态变迁等信息, 转换为 UIR 格式的对象; UIR 处理模块集成 UIR 的数据定义、导入导出功能, 同时提供数据解析和生成接口供其他模块调用; 模型验证模块抽取 UIR 数据结构中的有关信息, 生成相应的验证模型并进行验证, 规模约 2 700+行; 工具调用层基于命令行封装了多种工具, 供处理逻辑层调用, 规模约 200+行.

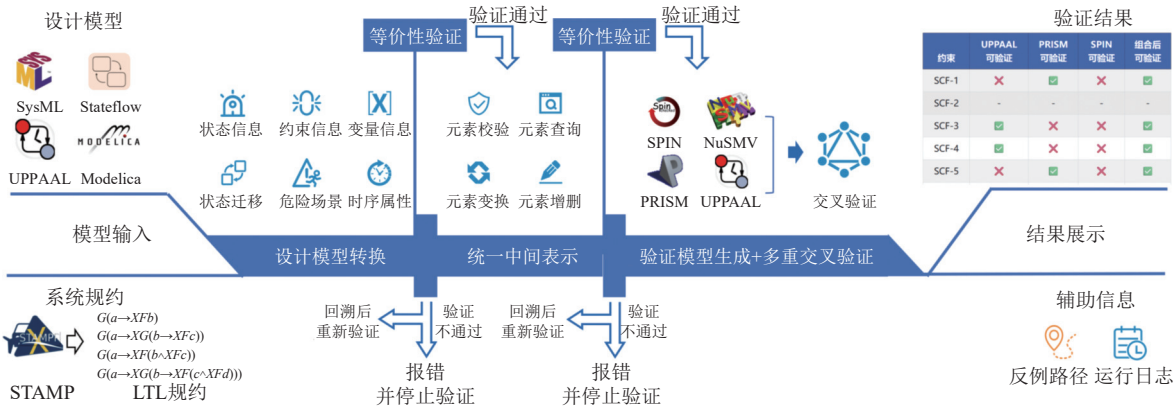

图 6 模型转换工具模块架构图

本平台目前支持 4 类主流设计建模语言到 5 种形式化验证工具的自动化转换, 如表 3 所示. 在设计模型输入界面, 平台能够解析 UML 状态图的状态转换结构和 OCL 约束规则, 提取 SysML 活动图的控制流和需求约束, 处理 Stateflow 的状态机模型和条件约束, 以及转换 Modelica 组件模型的行为方程和时间约束. 这些异构设计信息通过专用解析器统一转换为 UIR 标准表示, 形成包含状态集合、转换关系和约束条件的中间模型. 在验证工具输出界面, 平台可根据具体的任务逻辑转换为 SPIN、PRISM、NuSMV、UPPAAL 以及 STORM 等模型以验证 LTL、CTL、PCTL 以及 TCTL 等属性. 该转换框架的核心优势在于将传统的 $N\times M$ 复杂度转换问题简化为 $N+M$ 线性复杂度,

显著降低了系统的开发和维护成本. 当需要扩展新的设计模型语言时, 仅需开发一个到 UIR 的转换器; 当集成新的验证工具时, 仅需实现一个从 UIR 的代码生成器, 从而提升可扩展性和可维护性.

表 3　软件可支持转换列表

转换内容	设计模型转UIR			UIR转验证模型	
	状态提取	转换提取	约束提取	结构转换	属性转换
UML	状态图提取	转换提取	OCL提取	统一结构	统一约束
SysML	活动提取	流提取	需求提取	统一结构	统一约束
Stateflow	状态提取	转换提取	条件提取	统一结构	统一约束
Modelica	模式提取	事件提取	方程提取	统一结构	统一约束
UIR统一接口	getStates()	getTransitions()	getConstraints()	UIR结构	UIR约束
SPIN	接口读取	接口读取	接口读取	proctype	LTL
PRISM	接口读取	接口读取	接口读取	module	LTL、PCTL
NuSMV	接口读取	接口读取	接口读取	VAR	LTL、CTL
UPPAAL	接口读取	接口读取	接口读取	locations	TCTL
STORM	接口读取	接口读取	接口读取	states	PCTL
新工具	接口读取	接口读取	接口读取	标准接口	自动生成

用户交互模块主要基于 HTML 技术开发, 并与 Python 开发的后端服务进行交互. 工具界面如图 7 所示, 其中左侧为项目的文件结构, 分别表示设计解析、用户交互、中间表示和模型验证 4 大模块. 右侧上方呈现了模型转换及验证系统的主要界面. 设计模型文件上传后, 则可点击按钮转换为 UIR, 转换成功后会弹出新弹窗, 展示生成的 UIR 内容, 并支持在该弹窗中下载. 右侧下方的分析流程为 UIR 的验证, 主页面可选择不同的验证方法, 被验证的 UIR 文件可直接使用上方流程生成的结果, 或上传其他 UIR 文件. 点击相应验证方法后, 生成对应类型的验证模型; 运行验证流程, 可获得不同类型的验证模型的验证结果. 基于多种验证模型的验证结果, 可对生成的验证模型进行多重交叉验证.

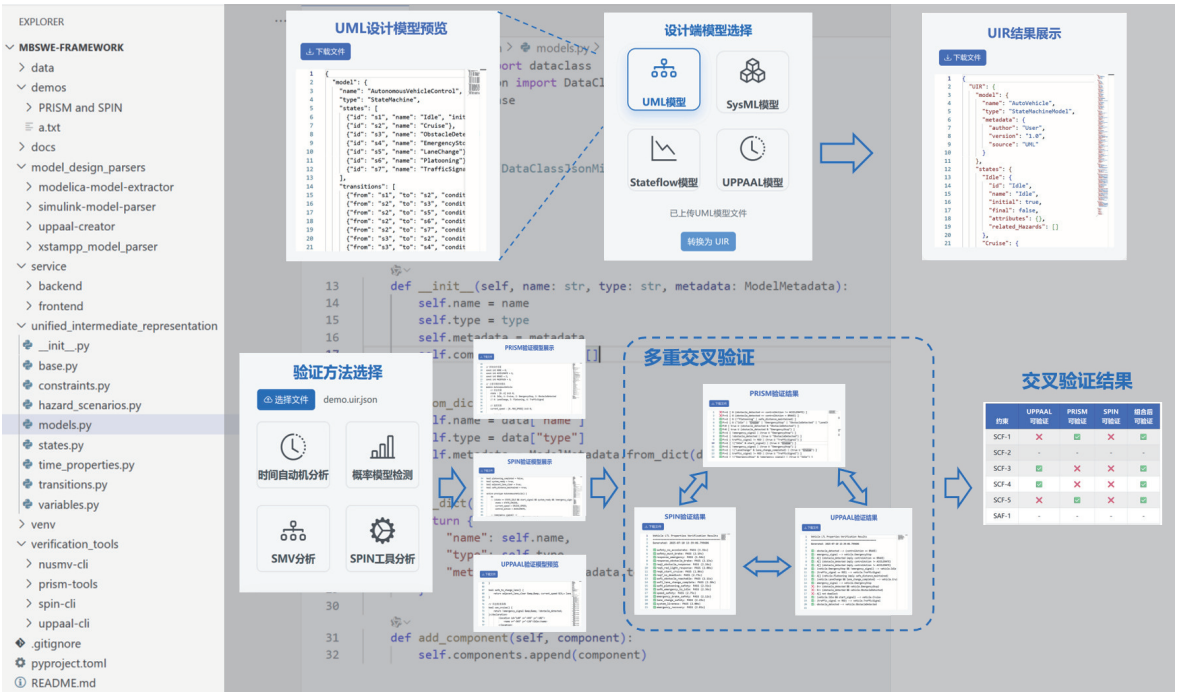

图 7　工具界面示意图

9 实例研究

本节以自动驾驶汽车控制软件 (autonomous vehicle control software, AVCS) 以及多域协同无人机飞控系统为例, 验证本文提出的方法的可行性.

9.1 自动驾驶汽车控制软件需求确认

AVCS 主要负责管理车辆的驾驶行为, 包括路径规划、避障、速度控制和紧急处理, 系统需要在动态环境中实时响应外部传感器数据, 同时确保车辆的安全性和高效性. 本案例主要对自动驾驶汽车控制软件的巡航、躲避障碍物等进行分析建模. 如图 8 所示.

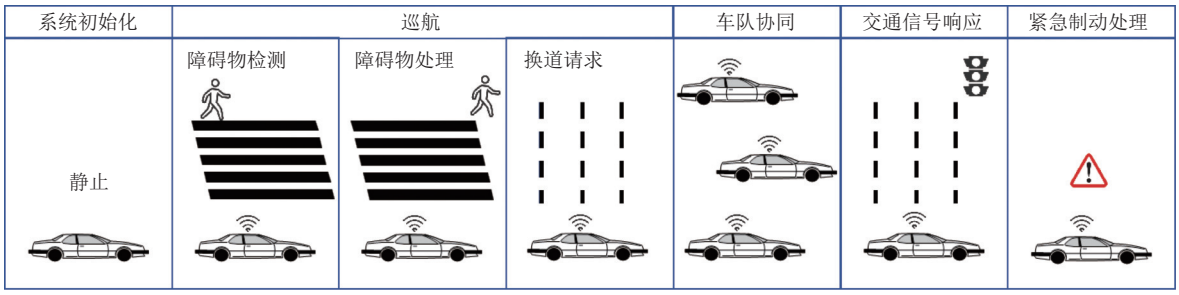

图 8 自动驾驶汽车控制软件工作图

通过对软件进行分析, 得到自动驾驶汽车控制软件的功能性需求如下.

- REQ-1: 自动驾驶汽车控制软件应具备静止状态和巡航模式的基本运行能力;
- REQ-2: 软件应能够检测到障碍物并执行避障操作;
- REQ-3: 自动驾驶汽车控制软件可以控制车辆进行车道切换;
- REQ-4: 软件应具备车队协同驾驶模式, 保持安全距离和同步速度;
- REQ-5: 自动驾驶汽车控制软件应能够识别并响应交通信号灯;
- REQ-6: 软件应根据启动信号自动从静止状态切换至巡航模式;
- REQ-7: 软件应能够根据环境变化和驾驶需求在不同状态间进行切换.

得到自动驾驶汽车控制软件的安全性需求如下.

- SAF-1: 当检测到障碍物时, 软件必须立即进入避障状态或紧急停车;
- SAF-2: 在紧急情况下, 软件应能够立即执行紧急停车操作;
- SAF-3: 车道切换过程中, 软件必须确保安全条件完成后才能返回巡航状态;
- SAF-4: 车队协同模式下, 软件必须维持与其他车辆的安全距离;
- SAF-5: 软件应严格遵守交通信号灯规则, 红灯时必须停车;
- SAF-6: 紧急情况解除后, 软件应安全地返回到静止状态等待进一步指令.

基于 UML 对 AVCS 的驾驶行为进行建模与验证, 软件的状态以及状态表示如后文表 4 所示, 包括静止状态、巡航模式状态以及检测到障碍物等情况.

9.2 设计层模型建立与安全约束生成

在本实例中, 控制行为主要分为两类: 状态管理和车辆控制, 如表 5 所示. 根据自动驾驶汽车控制软件的状态和状态切换逻辑, 建立如图 9 所示的 UML 状态图. 基于 STPA 系统论事故模型的分析框架, 本研究通过构建 AVCS 的层次控制结构模型, 识别出 A-1 (变道时车身不稳定, 严重时可能发生事故)、A-2 (追尾前车)、A-3 (不遵守交通信号, 本实例中主要体现为闯红灯)、A-4 (紧急制动时刹车失效) 这 4 种需要重点防范的高风险事故类型, 这些事故类型分布在感知层、决策层和执行层的不同控制层次中. 基于上述 A-1–A-4 事故类型分析结果, 识别出的事故类型转化为具体的系统风险项: H-1 (变道时车速过快)、H-2 (距离前车过近)、H-3 (无法在红灯前规定距离停止)、H-4 (不响应紧急制动指令) 和 H-5 (状态不稳定). 在风险项识别的基础上, 分析导致风险的关键控制行为

以及控制行为与系统安全的关联性, 识别安全关键控制行为.

表 4 自动驾驶汽车软件状态表示与描述

序号	状态集S状态标识	软件所处状态说明
1	Idle	车辆处于静止状态
2	Cruise	车辆处于巡航模式, 按照规划路径行驶
3	ObstacleDetected	检测到障碍物, 车辆减速并尝试避障
4	EmergencyStop	紧急情况, 车辆立即停止
5	LaneChange	车辆切换车道
6	Platooning	车队协同模式, 车辆保持安全距离和同步速度
7	TrafficSignal	响应交通信号灯, 红灯停车, 绿灯通行

表 5 AVCS 典型控制行为

类别	序号	ID	控制行为	是否安全关键
车辆控制指令	1	CA-8	油门指令	是
	2	CA-9	刹车指令	是
	3	CA-10	方向指令	是
状态管理指令	4	CA-1	设置状态为Idle	—
	5	CA-2	设置状态为Cruise	—
	6	CA-3	设置状态为ObstacleDetected	是
	7	CA-4	设置状态为LaneChange	是
	8	CA-5	设置状态为Platooning	—
	9	CA-6	设置状态为TrafficSignal	是
	10	CA-7	设置状态为EmergencyStop	是

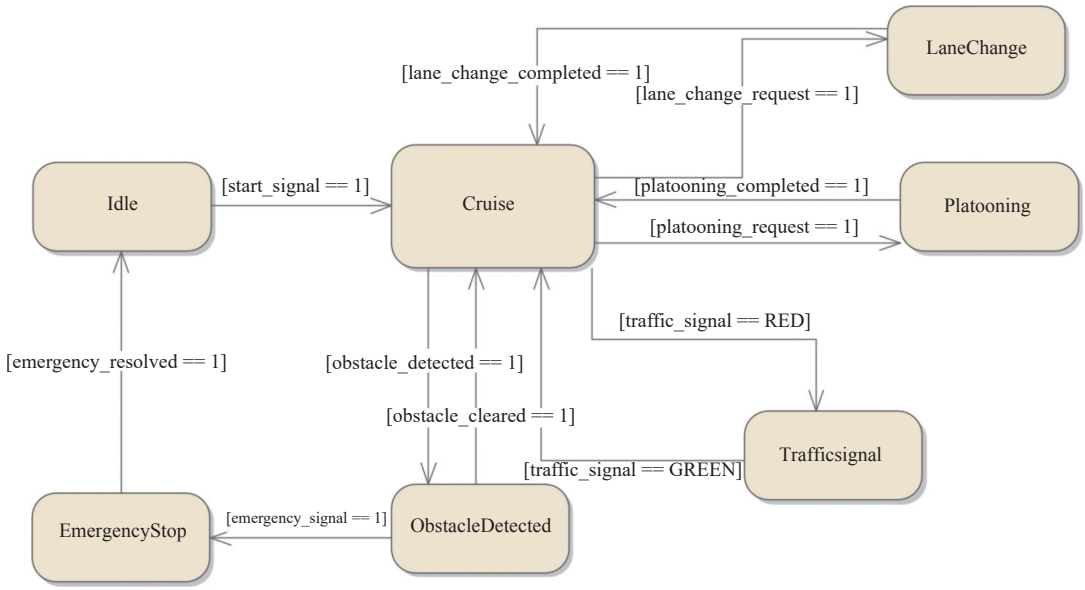

图 9 自动驾驶汽车控制软件 UML 状态图

基于风险识别和控制行为分析的结果, 构建基于 STAMP 的 AVCS 模型, 如图 10 所示, 将识别到的安全关键控制行为和风险项映射到具体的软件组件和状态变量中, 实现从抽象风险分析到具体系统建模的转化.

基于 STPA 方法的控制结构分析, 分析控制指令的语义和执行逻辑, 建立控制行为与系统状态变量之间的映射关系, 如表 6 所示, “—”表示不涉及此内容.

对 AVCS 中的控制行为进行系统性的安全分析, 识别必要控制行为未提供、不安全控制行为提供、控制行为提供过早或过晚以及控制行为停止过早或持续过长, 如表 7 所示, “—”表示不涉及此内容, 进而系统性地生成所有可能的不安全控制行为组合.

基于前述对不安全控制行为的分析, 安全约束规则采用 CTL 逻辑表达式进行建模, 确保约束的可验证性和无歧义性. 采用自动化软件对建模得到如下的安全约束.

(1) 当检测到障碍物时, 系统禁止执行加速控制动作

A[] (obstacle_detected imply controlAction != ACCELERATE), 记为 SCF1.

(2) 当接收到紧急信号时, 系统禁止执行加速控制动作

A[] (emergency_signal imply controlAction != ACCELERATE), 记为 SCF2.

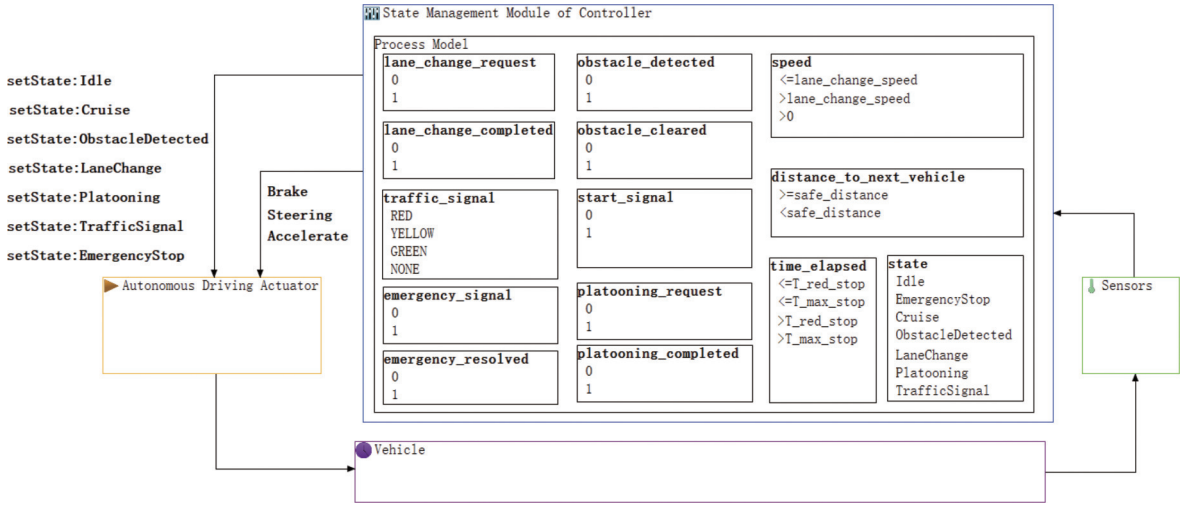

图 10　AVCS STAMP 模型

表 6　控制行为与状态变量映射关系

序号	控制指令	未提供控制行为可能导致风险	提供控制行为可能导致风险	功能执行相关的变量	其他安全约束得到的变量
1	设置状态为Idle	—	—	—	—
2	设置状态为Cruise	—	√	state, start_signal, platooning_completed, lane_change_completed, obstacle_cleared, traffic_signal	—
3	设置状态为ObstacleDetected	√	—	state, obstacle_detected	—
4	设置状态为LaneChange	—	√	state, lane_change_request	speed
5	设置状态为Platooning	—	√	state, platooning_request	—
6	设置状态为TrafficSignal	√	—	state, traffic_signal	—
7	设置状态为EmergencyStop	√	—	state, emergency_signal	—
8	油门指令	—	√	state, distance_to_next_vehicle, obstacle_detected, traffic_signal, emergency_signal	—
9	刹车指令	√	—	state, distance_to_next_vehicle, obstacle_detected, traffic_signal, emergency_signal	—
10	方向指令	—	√	—	—

(3) 当检测到障碍物时, 系统必须切换到障碍物检测状态

A[] (obstacle_detected imply controlAction = = BRAKE), 记为 SCF3.

(4) 当接收到紧急信号时, 系统必须切换到紧急停车状态

emergency_signal --> vehicle.EmergencyStop, 记为 SCF4.

(5) 当检测到障碍物时, 系统必须执行制动控制动作

obstacle_detected --> (controlAction = = BRAKE), 记为 SCF5.

对部分待验证的需求进行形式化提取, 得到如下约束.

- REQ-2: obstacle_detected --> vehicle.ObstacleDetected.
- REQ-5: (traffic_signal = = RED) --> vehicle.TrafficSignal.
- REQ-6: (vehicle.Idle && start_signal) --> vehicle.Cruise.
- REQ-7: A[] not deadlock.

- SAF-1: E<> (obstacle_detected && vehicle.ObstacleDetected); E<> (obstacle_detected && vehicle.EmergencyStop).
- SAF-2: emergency_signal --> vehicle.EmergencyStop.
- SAF-3: (vehicle.LaneChange && lane_change_completed) --> vehicle.Cruise.
- SAF-4: A[] (vehicle.Platooning imply safe_distance_maintained).
- SAF-5: (traffic_signal = = RED) --> vehicle.TrafficSignal.
- SAF-6: (vehicle.EmergencyStop && !emergency_signal) --> vehicle.Idle.

表 7　不安全控制行为识别

序号	系统级别危险与控制	控制指令	未提供控制行为可能导致风险	提供控制行为可能导致风险		
				任何时间下提供	控制行为提供过早或过晚	控制行为时间过长或过短
1	发现障碍物后未能及时减速或停车	油门指令	—	√	—	—
2		设置状态为ObstacleDetected	√	—	提供过晚	—
3		设置状态为EmergencyStop	√	—	提供过晚	—
4		刹车指令	√	—	提供过晚	控制行为时间过短
5	红灯之前未能及时减速停车	油门指令	—	√	—	—
6		设置状态为TrafficSignal	√	—	提供过晚	—
7		设置状态为EmergencyStop	√	—	提供过晚	—
8		刹车指令	√	—	提供过晚	控制行为时间过短
9	变道速度过快	刹车指令	√	—	提供过晚	—
10		方向指令	—	—	提供过早	—
11	距离前车过近	油门指令	—	—	—	—
12		刹车指令	√	—	提供过晚	控制行为时间过短
13	紧急制动无法生效	油门指令	—	√	—	—
14		设置状态为EmergencyStop	√	—	提供过晚	—
15		刹车指令	√	—	提供过晚	控制行为时间过短

9.3 统一中间表示生成

将 UML 状态图使用自动化工具转换为 UIR 进行下一步的安全约束验证, 如后文图 11 所示. 在确定 UIR 的结构后可以将 UIR 转换为 JSON 以进行后续的转换操作. UIR 到 JSON 的转换采用递归访问者模式, 结合类型映射表和约束验证机制, 确保转换的完整性和正确性. 转换框架包含以下核心组件.

(1) 语法分析器 (Parser): 解析 UIR 树状结构, 识别节点类型和层级关系.

(2) 类型映射器 (Type Mapper): 根据 UIR 节点类型确定 JSON 数据类型.

(3) 约束验证器 (Constraint Validator): 验证转换结果的语义一致性.

(4) 序列化器 (Serializer): 生成符合 JSON Schema 的标准输出.

根据 UIR 的语义模型, 定义如后文表 8 所示转换规则.

9.4 验证层模型转换

利用模型转换工具, 将 UIR 分别转换为 UPPAAL、PRISM 以及 SPIN 进行实时性、概率特性以及逻辑一致性的多重交叉验证.

9.4.1 UPPAAL 模型

通过模型转换工具将 UIR 的状态映射为 UPPAAL 的状态节点, 状态间的转换映射为 UPPAAL 的边 (带条件守卫), 变量映射为 UPPAAL 的全局或局部变量, 约束条件映射为 UPPAAL 的时钟、同步或守卫条件. 同时, UIR

中的实时性约束通过 UPPAAL 的时钟和时间边界表达, 安全性约束映射为 UPPAAL 中的待验证条件, 如图 12 所示.

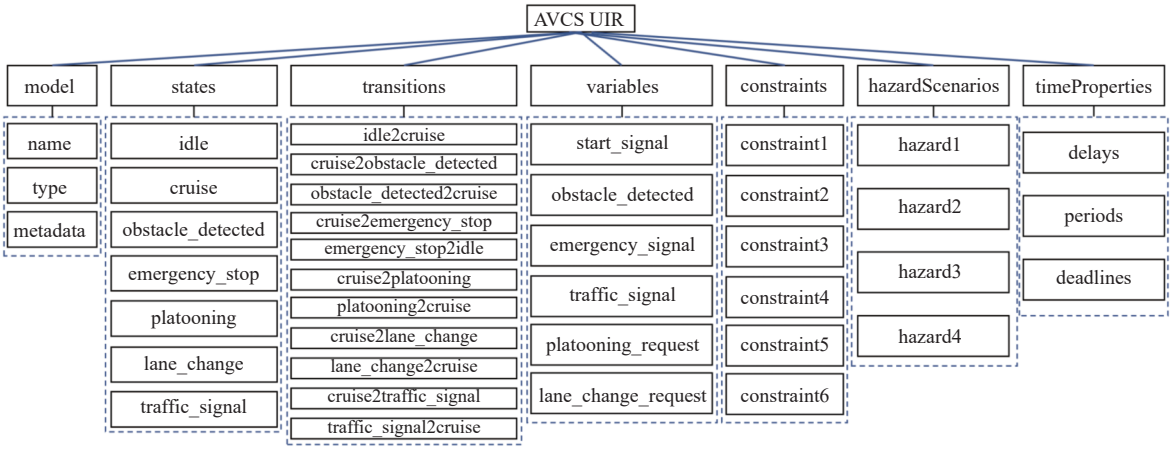

图 11　自动驾驶汽车控制系统 UIR

表 8　UIR 节点到 JSON 的映射规则

编号	转换规则	解释
1	object2object	UIR复合节点映射为JSON对象, 递归映射其所有子节点
2	list2array	UIR同类型节点集合映射为JSON数组
3	primitive2value	UIR叶子属性 (如string、number、Boolean)直接映射为JSON值
4	reference2string/array	UIR引用关系映射为ID字符串或字符串数组
5	attribute2pair	UIR属性键值对映射为JSON对象属性
6	expression2string	UIR表达式转为JSON中的字符串
7	initial/final2boolean	UIR初始/终止状态映射为JSON布尔型 initial/final 字段
8	metadata2object	metadata等嵌套属性整体映射为内嵌JSON对象
9	empty2null	无内容节点映射为JSON的null

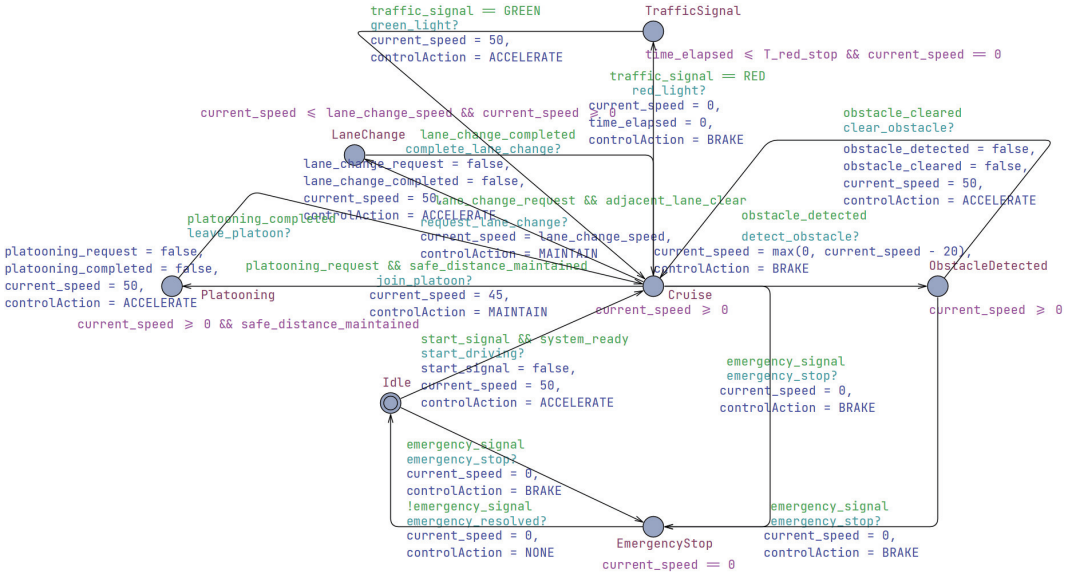

图 12　自动驾驶汽车控制系统 UPPAAL 模型

9.4.2 PRISM 模型

通过模型转换工具实现 UIR 至 PRISM 模型的转换, 算法 4 描述了从 UIR 的 JSON 表示转换为 PRISM 模型文件的过程, 输入为 UIR 的 JSON 表示, 输出为 PRISM 模型文件. 将 UIR 转换为 PRISM 模型以验证系统的概率行为和随机性.

算法 4. UIR2PRISM 模型转换.

输入: UIR_JSON (UIR 的 JSON 表示);
输出: PRISM_Model (PRISM 的模型文件).

1. Parse UIR_JSON
2. Extract States and Transitions
3. Initialize PRISM_Model with “dtmc” keyword
4. Define State Variable:
5. Add “state : [0…*N*–1] init 0;” where *N* is the number of states
6. Add variable declaration to <declaration> section
7. Add Transitions:
8. For each transition in Transitions:
9. Add <location> element with state ID and name
10. Add Initial State:
11. Find the initial state and add <init> element
12. Add Transitions:
13. For each transition in Transitions:
14. Add “[] state=source -> 1.0 : (state′=target);” to PRISM_Model
15. Return PRISM_Model

9.4.3 SPIN 模型

通过模型转换工具, 将 UIR 模型转换为 SPIN 模型, 算法 5 描述了从 UIR 的 JSON 表示转换为 SPIN 的 Promela 模型的过程, 输入为 UIR 的 JSON 表示, 输出为 SPIN 的 Promela 模型文件. 通过将 UIR 转换为 SPIN 模型以验证系统的逻辑一致性和并发问题.

算法 5. UIR2SPIN 模型转换.

输入: UIR_JSON (UIR 的 JSON 表示);
输出: Spin_Model (Spin 的 Promela 模型).

1. Parse UIR_JSON
2. Extract States and Transitions
3. Define State Enumeration:
4. Add “mtype = {state1, state2,…, state*N*};”
5. Initialize State Variable:
6. Add “mtype state = initial_state;”
7. Add Transitions:
8. Add “do...od” loop:
9. For each transition in Transitions:
10. Add “state == source -> state = target;”

11. Return Spin_Model

9.5 多重交叉验证

9.5.1 UPPAAL 模型验证

UPPAAL 的验证结果如图 13 所示. 在 UPPAAL 模型的验证结果中, A[] not deadlock、E<> (obstacle_detected && vehicle.ObstacleDetected)、E<> (obstacle_detected && vehicle.EmergencyStop) 验证不通过, 说明系统存在死锁且有条件不满足, 通过反例路径分析, 当进入 ObstacleDetected 状态后, obstacle_detected 被设置为 false, 这导致 obstacle_detected && vehicle.ObstacleDetected 永远不可能为真, 导致了系统死锁, 也导致了 SAF-1 验证不通过. 后续增加环境模型管理 obstacle_detected 信号, 死锁问题消失.

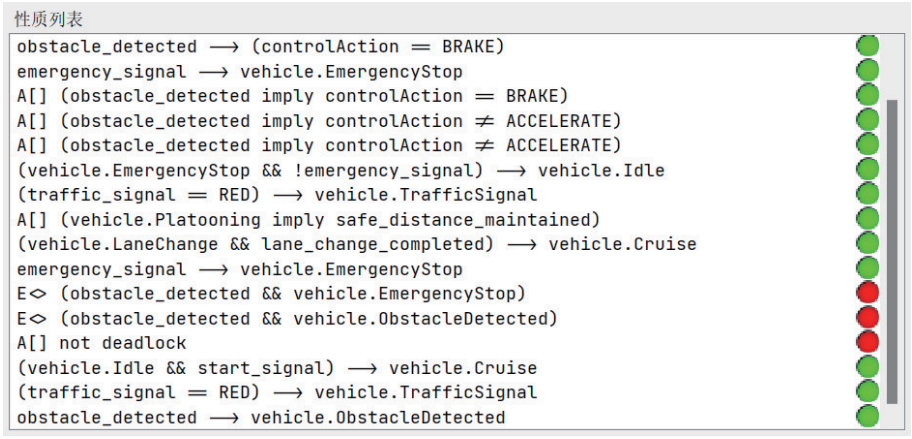

图 13 UPPAAL 模型验证结果

9.5.2 PRISM 模型验证

PRISM 模型的验证结果如图 14 所示. PRISM 验证结果显示所有状态可达性、响应性以及活性属性均通过验证, 表明模型具备完整的行为逻辑和状态转换能力; 然而, 以下两个关键安全属性, 均验证失败, 揭示了控制策略中存在非确定性执行路径.

(1) P>=1[G(obstacle_detected=>controlAction!=ACCELERATE)];

(2) P>=1[G(obstacle_detected=>dcontrolAction=BRAKE)].

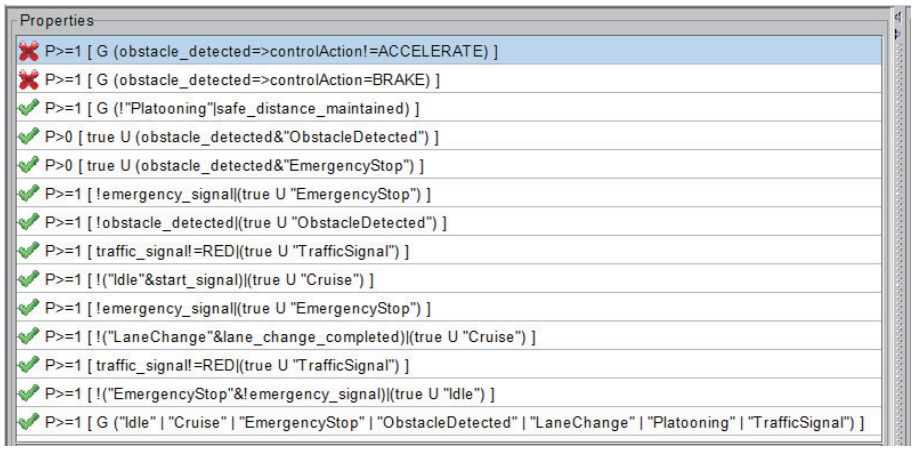

图 14 PRISM 模型验证结果

通过反例轨迹分析发现, 当 obstacle_detected=true 时, 系统存在状态转换非原子性问题, 导致 controlAction 在某些执行路径上保持先前值, 违反了全局算子 G 的语义要求, 即在瞬时安全约束方面存在漏洞, 直接影响车辆在危险情况下的即时响应能力.

9.5.3 SPIN 模型验证

SPIN 验证结果如图 15 所示. SPIN 验证的 22 个线性时序逻辑 (linear temporal logic, LTL) 属性均通过, 未发现系统相关问题, 包括安全性属性 9 个 (如障碍物检测时禁止加速、紧急情况下强制制动)、活性属性 6 个 (如系统状态可达性、无死锁保证) 和响应性属性 7 个 (如交通信号响应、车道变换响应), 平均验证时间 2.31 s, 总验证时间 50.82 s.

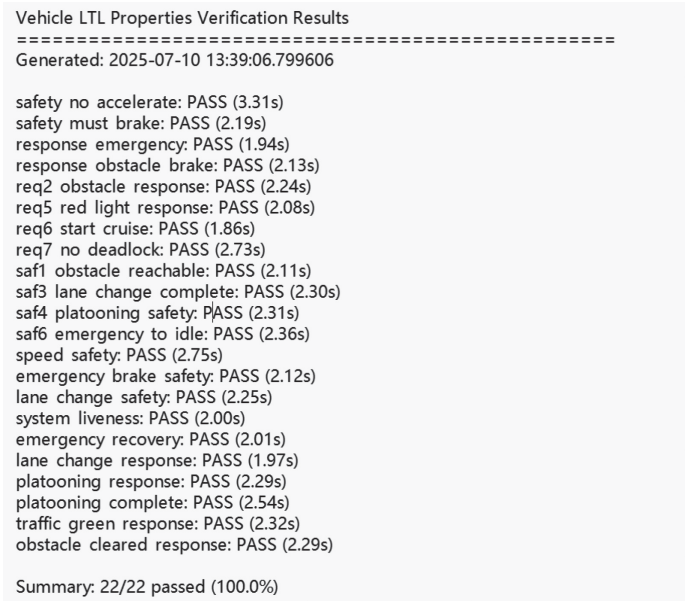

图 15 SPIN 模型验证结果

9.5.4 多重交叉验证分析

对同一 UIR 模型, UPPAAL、PRISM 和 SPIN 这 3 种模型检测工具分别从时序逻辑、概率验证和线性时序逻辑这 3 个维度对系统功能性、安全性和可靠性进行检测与评估. 如表 9 所示, 验证结果显示, UPPAAL、PRISM 和 SPIN 这 3 种检测工具问题发现率分别是 60%、40%、0%, 而本文所提方法问题发现率为 100%, 这说明了本文所提方法的有效性. 统一中间表示融合了验证模型中的语义信息并且包含 STPA 理论分析得到的安全约束, 分层转换模型保证了转换过程中语义的一致性和正确性, 而多重交叉验证机制确保了检验的全面性和可验证性以及互补性, 有效提高了问题检出率, 保障了系统安全.

表 9 自动驾驶汽车控制系统多重交叉验证结果

约束	UPPAAL	UPPAAL反例路径	PRISM	PRISM反例路径	SPIN	SPIN反例路径	本文方法	本文方法反例路径
REQ-2	—	—	—	—	—	—	—	—
REQ-5	—	—	—	—	—	—	—	—
REQ-6	—	—	—	—	—	—	—	—
REQ-7	√	存在	×	○	×	○	√	存在
SAF-1	—	—	—	—	—	—	—	—
SAF-2	—	—	—	—	—	—	—	—
SAF-3	—	—	—	—	—	—	—	—
SAF-4	—	—	—	—	—	—	—	—

表 9 自动驾驶汽车控制系统多重交叉验证结果 (续)

约束	UPPAAL	UPPAAL反例路径	PRISM	PRISM反例路径	SPIN	SPIN反例路径	本文方法	本文方法反例路径
SAF-5	—	—	—	—	—	—	—	—
SCF-1	×	○	√	存在	×	○	√	存在
SCF-2	—	—	—	—	—	—	—	—
SCF-3	√	存在	×	○	×	○	√	存在
SCF-4	√	存在	×	○	×	○	√	存在
SCF-5	×	○	√	存在	×	○	√	存在

注: “√”表示检出问题; “存在”表示可提供相应的反例路径描述, 从而定位问题; “×”表示有问题但未检出; “○”表示无反例路径信息; “—”表示没有问题

9.6 执行复杂度实证分析

9.6.1 AVCS 实例规模度量

AVCS 实例的规模参数分为状态数量参数 7 个主要状态 (Idle、Cruise、ObstacleDetected、LaneChange、Platooning、TrafficSignal、EmergencyStop)、10 个安全关键控制行为、8 个核心变量 (speed、distance_to_next_vehicle、obstacle_detected、traffic_signal、state、emergency_signal、lane_change_request、start_signal), 总模型元素数量 n 约为 25 个核心建模元素.

9.6.2 派生过程复杂度实证分析

AVCS 实例派生复杂度包括 UPPAAL 模型派生、PRISM 模型派生以及 SPIN 模型派生. 其中, UPPAA 模型派生过程主要构建状态迁移关系, 处理二元关系 (状态-迁移-状态), 复杂度为 $T_{\text{derive}}^{\text{UPPAAL}}(n)=\text{O}(n^2)\approx\text{O}(25^2)=\text{O}(625)$; PRISM 模型需要处理概率转移和随机性, 复杂度为 $T_{\text{derive}}^{\text{PRISM}}(n)=\text{O}(n^{2.5})\approx\text{O}(25^{2.5})=\text{O}(3125)$; SPIN 模型主要处理进程间通信和并发关系, 建立进程与通信通道之间的二元关系, 派生生成 Promela 模型的复杂度为 $T_{\text{derive}}^{\text{SPIN}}(n)=\text{O}(n^2)\approx\text{O}(625)$. 因此, AVCS 实例的实际派生复杂度维度 $d_{\text{actual}}\approx 2.2<3$, 这得益于 UIR 提供的结构化约束信息.

9.6.3 验证过程复杂度实证分析

AVCS 实例的验证复杂度来源于基础验证复杂度、协调复杂度两类. 对模型规模和实测执行时间的双对数数据点进行线性回归分析后, 可以得到基于验证复杂度包含模型检测 (UPPAAL) 复杂度: $T_{\text{UPPAAL}}(n)=\text{O}(n^{2.8})$; 概率验证 (PRISM) 复杂度: $T_{\text{PRISM}}(n)=\text{O}(n^{3.2})$; 并发验证 (SPIN) 复杂度: $T_{\text{SPIN}}(n)=\text{O}(n^{2.5})$. 进一步地, 协调 3 种验证结果, 处理 22 个 LTL 属性: $T_{\text{coord}}(n)=\text{C}_n^2=\dfrac{n(n-1)}{2}\approx\text{O}(n^2)=\text{O}(625)$.

总体验证复杂度为: $T_{\text{verify}}(n)=\text{O}(\max(n^{2.8},n^{3.2},n^{2.5}))+\text{O}(n^2)=\text{O}(n^{3.2})$, 即 $e_{\text{actual}}=3.2$.

9.6.4 整体复杂度实证分析

AVCS 整体复杂度为: $T_{\text{total}}(n)=\text{O}(n^c)$, 其中 $c=\max(2,d_{\text{actual}},e_{\text{actual}})=\max(3,2.2,3.2)=3.2$. 基于分析结果, 可以发现, 实际复杂度 ($\text{O}(n^{3.2})$) 显著低于传统方法的指数复杂度 ($\text{O}(2^n)$), 同时验证过程是复杂度主导因素 ($e=3.2>d=2.2$). 实例复杂度低于理论最坏情况 ($c=3.2<4$), 证实了理论分析的保守性.

根据 AVCS 的复杂度实证分析可以发现, 实例数据支持了参数化复杂度理论的正确性, 实际复杂度 ($\text{O}(n^{3.2})$) 处于理论范围内 ($\text{O}(n^c), c\leqslant 4$); 多项式复杂度使得验证时间从传统方法的数小时降低到秒级, 具有处理大规模系统的潜力.

9.7 多域协同无人机飞控系统案例

9.7.1 目标系统简介

多域协同无人机飞控系统 (简称“飞控系统”) 主要负责管理和控制无人机集群中每一个无人机个体的行为, 包括任务规划、编队控制、自主避障和应急迫降等. 系统需要在复杂的动态环境中实时响应多源传感器数据与集群通信信息, 同时确保整个蜂群的安全性、协同性与任务高效性. 本案例主要对智能蜂群飞控软件的航线执行、协同规避、编队飞行等关键功能进行分析建模, 其状态示意图如图 16 所示, 其中 G1–G5 代表地面上的状态, F1–F7

为飞行中的状态.

9.7.2 目标系统安全约束

基于 STPA 方法的控制结构分析, 分析控制指令的语义和执行逻辑, 建立控制行为与系统状态变量之间的映射关系, 并最终得到 19 条安全约束, 其中 GSR 代表无人机位于地面情况下的安全约束, 共 9 条; FSR 代表无人机处于飞行情况下的安全约束, 共 10 条. 安全约束及其形式化表示如下.

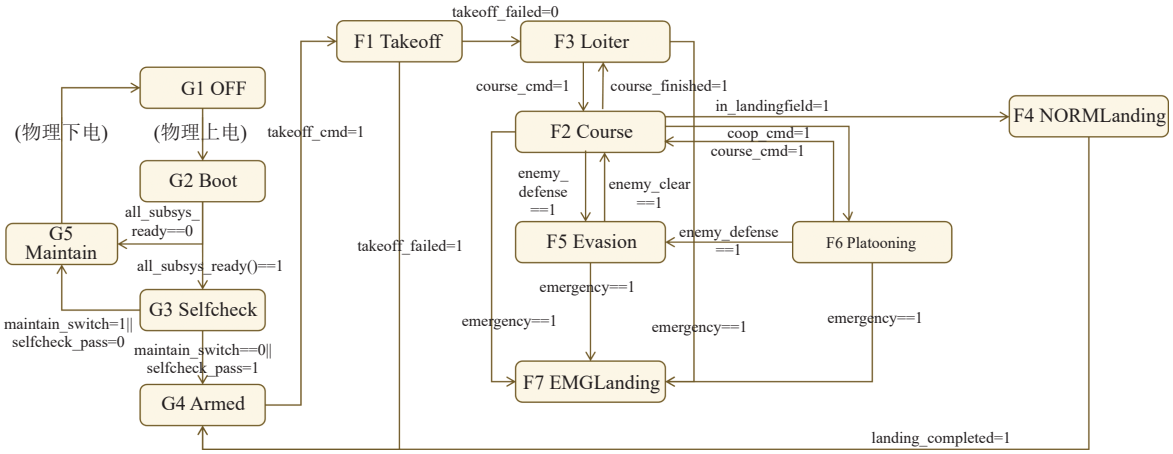

图 16 多域协同无人机飞控系统 UML 状态图

(1) [GSR-1] 当动力、控制、导航等子系统未启动完成时, 不得进行自检.

A[] (!all_subsys_ready) --> !(state= =G3).

(2) [GSR-2] 当电量不足、通信异常时禁止从 G4 待命状态进入 F1 起飞状态.

A[] (power_low= =1 || communication_abnormal= =1) --> !(state= =F1).

(3) [GSR-3] 当处于 G2 的状态超过启动超时, 则须直接进入维护状态.

A[] (state = = G2 && time > startup_timeout) --> (state= =G5).

(4) [GSR-4] 当处于 G3 自检状态的时间超过自检超时, 则说明可能存在外设自检超时等问题, 须直接进入维护状态.

A[] (state = = G3 && time > selfcheck_timeout) --> (state= =G5).

(5) [GSR-5] 当维护标志位引脚 (maintain_switch) 为高电平 1 时, 自检完成后应进入 G5 维护状态, 否则进入 G4 待命状态.

A[] (maintain_switch = = 1) --> (state=G5).

A[] (selfcheck_flag = 0 && selfcheck_pass) --> state = = G4.

(6) [GSR-6] 在 G4 状态下, 当电池放电电流为负 (充电) 时, 不应切换到 F1 状态.

A[] (state = = G4 && discharge_current < 0) --> !(state = = F1).

(7) [GSR-7] 在 G4 待命状态下, 若关键传感器 (IMU、GPS 等) 数据异常或失效, 不得响应起飞指令, 须重新进入 G3 自检状态.

A[] (state = = G4 && (IMU_abnormal=1 || GPS_abnormal=1)) --> (state = = G3).

(8) [GSR-8] 在 G5 维护状态下, 若维护操作被非预期中断, 再次上电后应直接进入 G5 维护状态.

A[] (state = = G5 && power_interrupt) --> (power_on --> state = = G5).

(9) [GSR-9] 在 G4 待命状态下, 若 GPS 定位精度低于阈值且持续超过 T2, 不得进入 F1 起飞状态.

A[] (state = = G4 && GPS_accuracy < threshold && time > T2) --> !(state= =F1).

(10) [FSR-1] 在 F1 起飞状态下, 若超过起飞超时后, 空速 (airspeed) 和高度 (altitude) 都很小, 则说明可能起飞失败, 应退回 G4 待命模式.

A[] (state = = F1 && time > takeoff_timeout && airspeed < AirspeedMin && altitude < heightMin) -->
(state = = G4 && error_report).

(11) [FSR-2] 在 F2 航线/任务状态下, 接收新航线或任务航点时, 应保证剩余电量 (battery_remaining) 在足以覆盖预估所需电量 (estimated_consumption) 的同时还有剩余 (reserved_pwr).

A[] (state = = F2 && new_waypoint_received) --> (battery_remaining > estimated_consumption + reserved_pwr).

(12) [FSR-3] 在 F3 盘旋/悬停状态下, 当剩余电量 (battery_remaining) 即将不满足飞往最近降落场所需 (power_needed_to_nearest_landing) 时, 应自动进入 F2 航线模式.

A[] (state = = F3 && battery_remaining < power_needed_to_nearest_landing) --> state = = F2.

(13) [FSR-4] 在 F2、F3 和 F4 状态下, 当通信中断 (communication_lost) 时, 应立即进入 F2 航线模式, 飞回最近的降落场.

A[] ((state = = F2 || state = = F3 || state = = F4) && communication_lost) --> state = = F2.

(14) [FSR-5] 在 F2、F3、F5 和 F6 状态下, 当定位信号中断 (positioning_lost) 且持续时间超过 T1 (positioning_lost_time>T1) 时, 应立即进入 F7 迫降状态.

A[] ((state = = F2 || state = = F3 || state = = F5 || state = = F6) && positioning_lost && positioning_lost _time > T1) -->
state = = F7.

(15) [FSR-6] 在 F6 状态下, 编队飞行的两无人机距离 (uav_distance) 应不低于 d_min.

A[] (state = = F6) --> (uav_distance >= d_min).

(16) [FSR-7] 在 F5 规避状态下, 无人机间距应在进入规避模式后 20 s 内将间距 (uav_distance) 散开到不低于规避模式下最小间距 d_circ_min.

A[] (state = = F5 && time < 20s) --> (uav_distance >= d_circ_min).

(17) [FSR-8] 在 F2、F3 和 F5、F6 状态下, 若飞机的动力、控制出现异常 (control_abnormal), 应及时发送遭受火力打击的广播报文 (broadcast_fire_attack=1).

A[] ((state = = F2 || state = = F3 || state = = F5 || state = = F6) && control_abnormal) --> broadcast_fire_attack.

(18) [FSR-9] 在 F6 状态下, 若接收到遭受火力打击广播报文 (fire_broadcast_received) 或集群中任一无人机因非自身原因下线 (any_peer_unexpected_offline), 应进入 F5 规避飞行模式.

A[] (state = = F6 && (fire_broadcast_received || any_peer_unexpected_offline)) --> state=F5.

(19) [FSR-10] 在 F2、F3 和 F5、F6 状态下, 当无人机因电量不足、通信或导航异常等自身原因 (self_fatal_exc) 下线前, 应广播自身出现致命异常 (self_fatal_broadcast).

A[] ((state = = F2 || state = = F3 || state = = F5 || state = = F6) && self_fatal_exc) --> self_fatal_broadcast.

9.7.3 多重交叉验证

针对多域协同无人机飞控系统地面状态和飞行状态下的安全约束, 进行多重交叉验证, 最终部分安全约束的结果如后文表 10 所示. 对同一 UIR 模型, UPPAAL、PRISM 和 SPIN 这 3 种模型检测工具分别从时序逻辑、概率验证和线性时序逻辑这 3 个维度对系统功能性、安全性和可靠性进行检测与评估. 验证结果显示, UPPAAL、PRISM 和 SPIN 这 3 种检测工具的问题发现率分别是 33%、33%、66%, 而本文所提方法问题发现率为 100%. 以上结果证明, 本文提出的方法有效提高了问题检出率, 保障了系统安全.

9.7.4 执行复杂度实证分析

飞控系统实例的规模参数分为状态数量参数 12 个主要状态、12 个安全关键控制行为、29 个核心状态变量, 总模型元素数量 n 约为 53 个核心建模元素.

多域协同无人机飞控系统实例派生复杂度包括 UPPAAL 模型派生、PRISM 模型派生以及 SPIN 模型派生.

其中, UPPAAL 模型派生的复杂度为 $T_{\text{derive}}^{\text{UPPAAL}}(n) = \text{O}(n^2) \approx \text{O}(53^2) = \text{O}(2809)$; PRISM 模型派生复杂度为 $T_{\text{derive}}^{\text{PRISM}}(n) = \text{O}(n^{2.5}) \approx \text{O}(53^{2.5}) = \text{O}(20449.83)$; SPIN 模型派生复杂度为 $T_{\text{derive}}^{\text{SPIN}}(n) = \text{O}(n^2) \approx \text{O}(53^2) = \text{O}(2809)$. 因此, 飞控系统实例的实际派生复杂度维度 $d = \log_{53}((2809+204449.83+2809)/3) \approx 2.3 < 3$.

表 10　多域协同无人机飞行控制系统多重交叉验证结果

约束	UPPAAL	UPPAAL反例路径	PRISM	PRISM反例路径	SPIN	SPIN反例路径	本文方法	本文方法反例路径
GSR-1	—	—	—	—	—	—	—	—
GSR-2	—	—	—	—	—	—	—	—
GSR-3	—	—	—	—	—	—	—	—
GSR-4	—	—	—	—	—	—	—	—
GSR-5	×	○	√	存在	√	存在	√	存在
GSR-6	×	○	×	○	√	存在	√	存在
GSR-7	—	—	—	—	—	—	—	—
GSR-8	—	—	—	—	—	—	—	—
GSR-9	—	—	—	—	—	—	—	—
FSR-1	—	—	—	—	—	—	—	—
FSR-2	—	—	—	—	—	—	—	—
FSR-3	—	—	—	—	—	—	—	—
FSR-4	—	—	—	—	—	—	—	—
FSR-5	√	存在	×	○	×	○	√	存在
FSR-6	—	—	—	—	—	—	—	—
FSR-7	—	—	—	—	—	—	—	—
FSR-8	—	—	—	—	—	—	—	—
FSR-9	—	—	—	—	—	—	—	—
FSR-10	—	—	—	—	—	—	—	—

注: “√”表示检出问题; “存在”表示可提供相应的反例路径描述, 从而定位问题; “×”表示有问题但未检出; “○”表示无反例路径信息; “—”表示没有问题

飞控系统实例的验证复杂度来源于基础验证复杂度、协调复杂度两类. 基础验证复杂度中, 模型检测 (UPPAAL) 复杂度为 $T_{\text{UPPAAL}}(n) = \text{O}(n^{2.8})$; 概率验证 (PRISM) 复杂度为 $T_{\text{PRISM}}(n) = \text{O}(n^{3.2})$; 并发验证 (SPIN) 复杂度为 $T_{\text{SPIN}}(n) = \text{O}(n^{2.5})$. 进一步地, 协调复杂度为 $T_{\text{coord}}(n) = \text{C}_n^2 = (n(n-1))/2 \approx \text{O}(n^2)$, 总体验证复杂度为 $T_{\text{verify}}(n) = \text{O}(\max(n^{2.8}, n^{3.2}, n^{2.5})) + \text{O}(n^2) = \text{O}(n^{3.2})$, 即 $e_{\text{actual}} = 3.2$.

通过以上分析, 可得出飞控系统整体复杂度为: $T_{\text{total}}(n) = \text{O}(n^c)$, 其中 $c = \max(2, d_{\text{actual}}, e_{\text{actual}}) = \max(3, 2.3, 3.2) = 3.2$. 基于分析结果, 可以发现, 在飞控系统中, 实际复杂度 ($\text{O}(n^{3.2})$) 同样显著低于传统方法的指数复杂度 ($\text{O}(2^n)$), 同时验证过程是复杂度主导因素 ($e = 3.2 > d = 2.3$).

10　相关工作

随着安全关键系统复杂性的不断增长, 模型驱动工程已成为系统设计与验证的重要技术路径. 该技术路径的核心挑战在于: 如何在异构建模环境中实现高效且可追溯的模型转换, 如何确保转换过程的语义一致性与性质保持, 以及如何将形式化验证与安全风险分析深度融合. 下面就安全风险分析、模型转换、集成验证这 3 个方面进行介绍.

10.1　安全风险分析

当前安全风险分析的研究方法主要聚焦于 3 个方向: (1) 模型检查驱动的形式化验证: 以航空飞控模式转换[6]为代表, 借助 SPIN/NuSMV/SAL 等在给定抽象下对并发与时序行为进行系统性 (近似穷尽) 探索, 并延伸出 SysML 至 NuSMV 的转换[18]、AADL Safety Annex+JKind[19]以及以 SMV/NuSMV 为核心的基于模型的安全分析

(model-based safety analysis, MBSA) 将反例回填至 FMEA/FTA 的工程化链路[20], 亦有 smartiflow[7]等专用语言通过“仿真+模型检查”的两阶段流程提高早期分析效率; (2) 系统理论/MBSE 集成: 将 STPA 工件属性化 (如映射至 Event-B/Rodin[8], 或在 SysML 中以 OCL 约束验证[9]), 在架构层引入系统级安全约束; (3) 概率化与数据驱动扩展: 包括面向人机协作的 TRIO+Zot 危险识别[10]、DryVR[21]的数据驱动可达性方法, 以及海洋场景下 FSA/贝叶斯网络的风险评估[22]. 整体上, 现有研究在域内取得了阶段性进展, 但跨平台应用仍受制于 3 方面的瓶颈: 第一, 跨语言/跨工具的统一语义承载相对薄弱, 缺乏可操作的统一语义载体以对齐结构、行为、时序、概率与故障语义并确保性质保持; 第二, 多工具常为并列使用而非协同流程, 结果的交叉验证与融合支撑有限; 第三, 从设计模型到验证/安全分析模型的性质保持与可追溯转换尚待加强, 影响端到端覆盖与工程闭环.

10.2 模型转换技术

现有模型转换研究主要围绕理论方法、工具生态与工程实践这 3 个方面展开. 理论上, Mens 等人[11]对转换技术空间、方向与执行机制 (如 QVT 双向规则) 进行了体系化刻画; 工具层面, 从通用性、风格与协作支持等维度比较的研究[12], 提示跨语言互操作普遍依赖点对点适配, 易引发“$N\times M$”扩张; 工程上, 既有在 EMF 生态内强调性能优化的转换引擎 (如 YAMTL[13], 在特定基准上取得显著加速) 与面向最终用户的可视化语言 (VMTL)[23], 也有服务验证的显式映射与形式化承载 (如 SysML 到时间自动机/Stateflow 的转换[24]、UML 到 TEGG 图的转换[25]、AADL 到 PRISM 的转换[26]), 以及面向平台或领域的适配与抽取 (PIM 转换 EJB[27]、Simulink 架构/操作模式抽取[28]、SimMechanics 直转[29]). 由此可见, 现有工作仍以单一源/目标语言或局部语义为中心, 针对时序、概率、故障等非功能语义的统一承载与性质保持, 尚缺乏通用且可证明的方法论. 此外, 双向/增量转换与细粒度可追溯能力在实际工具链中稳定性不足. 转换与下游验证/安全分析之间的属性传递与结果互证机制亦未形成系统化方案. 与此同时, 跨工具互操作能力主要依赖个案适配, 易导致跨平台扩展与维护成本偏高.

10.3 集成验证技术

近年来, 模型验证正从单一技术应用走向以“集成验证”为核心的协同与闭环, 研究重点在于将多种验证技术、工具链与系统层级整合进同一流程, 以实现结果互证与设计回注. 一些研究人员通过技术集成实现端到端覆盖. S3 工具链[14]整合归纳证明、BMC、测试用例生成和等价性证明, 覆盖系统、设计、代码这 3 个层面的属性验证; UCLID5[15]采用统一前端承载一阶逻辑、时序与超属性, 并集成归纳证明、BMC 与语法制导合成; 同时, 研究者们尝试通过跨层闭环与运行时联动实现动态的闭环. 航天领域的 ASSURE[16]将确定性证明 (Dafny/TLAPS/Agda)、概率性验证、运行时监控与 DDDAS 反馈结合, 形成覆盖设计、验证、运行的动态保证闭环; 铁路联锁[17]以“设计语言、组合语义、多工具验证 (UPPAAL/Spin/Theta)、测试回注”的链路实现模型-测试往返; CPS[30]设计采用“AADL、UPPAAL、HIL 测时、二次模型检查、运行时监测合成”, 以硬件在环数据回馈验证结论并生成检测器; 进一步地, 一些研究人员尝试通过领域化集成进行可组合验证. 汽车软件[31]在 ISO 26262 场景下并行引入抽象解释 (Polyspace/Astrée)、SMT-驱动模型检查 (SCADE Design Verifier/JKind) 与演绎证明 (Frama-C WP), 以提升覆盖面; BPEL 工作流[32]结合 PRISM 的概率分析, 群体自适应与动态组件系统通过分解/抽象与 PCTL/PRISM 进行规模化验证, 协议组合采用 Promela/SPIN[33], 分析组合属性并生成攻击反例. 其他方向还包括 DL_CS/Isabelle[34]的架构模式证明 (强调可组合规范)、ASM 至 NuSMV 的转换[35]的自适应系统验证以及 TEE/SGX[36]的状态连续性验证 (面向中断/异常情形的一致性保障). 综上所述, 集成验证在多技术协同、工具链整合、跨层级覆盖与闭环迭代方面取得显著进展, 但关键瓶颈依然存在: 跨语言语义互操作与性质保持基础薄弱, 且设计验证与安全验证之间缺乏一致的贯通建模, 从而难以支撑可审计的端到端证据链.

11 总 结

本文针对复杂系统开发中设计模型与验证模型互操作性不足的问题, 提出了一种基于统一中间表示 (UIR) 的分层转换与多重交叉验证机制. 该机制在设计层引入基于 STPA 的系统性风险分析, 通过危险识别与不安全控制行为分析, 将安全约束作为核心要素融入 UIR 构建过程, 实现了从设计阶段到验证阶段的安全属性全流程追溯.

进一步地, 通过构建统一中间表示有效弥合了设计工具与验证工具在建模语言、语义表达和数据结构等方面的差异, 系统性地解决了语义不一致、工具链适配性差以及动态行为支持不足等关键问题. 与现有方法相比, 所提方法不仅实现了设计模型与验证模型的准确语义映射, 还通过分层转换机制系统化定义了包含安全约束在内的高效转换生成规则. 最后, 在验证端采用多模型的多重交叉验证策略, 将源于安全风险分析的约束转化为可验证属性, 能够发现并定位传统单一验证工具难以识别的潜在问题. 在自动驾驶汽车控制软件以及多域协同无人机飞控系统案例上进行方法验证, 验证结果表明, 该机制能够显著提高模型转换效率和验证准确性, 能够在软件开发生命周期的早期阶段发现并定位问题, 提高系统的可靠性. 目前, 所提方法在处理大规模复杂系统时的验证效率和转换规则的自适应性方面仍存在提升空间. 在未来的工作中, 将重点研究以下几个方向: 开发基于人工智能的自适应转换规则生成机制, 以降低对人工预定义规则的依赖, 提升方法的通用性与智能化水平; 探索面向大规模复杂系统的分布式验证技术, 进一步拓展本方法的工程应用边界.

References

[1] Lee D, Nam T, Park D, Kim Y, Na J. Enhanced soft error rate estimation technique for aerospace electronics safety design via emulation fault injection. Applied Sciences, 2024, 14(4): 1470. [doi: 10.3390/app14041470]

[2] Neelofar N, Aleti A. Identifying and explaining safety-critical scenarios for autonomous vehicles via key features. ACM Trans. on Software Engineering and Methodology, 2024, 33(4): 94. [doi: 10.1145/3640335]

[3] Ferrari A, Mazzanti F, Basile D, ter Beek MH. Systematic evaluation and usability analysis of formal methods tools for railway signaling system design. IEEE Trans. on Software Engineering, 2022, 48(11): 4675–4691. [doi: 10.1109/TSE.2021.3124677]

[4] Mejía-Granda CM, Fernández-Alemán JL, Carrillo-de-Gea JM, García-Berná JA. Security vulnerabilities in healthcare: An analysis of medical devices and software. Medical & Biological Engineering & Computing, 2024, 62(1): 257–273. [doi: 10.1007/s11517-023-02912-0]

[5] Holzmann GJ. The analysis of safety critical software systems. IEEE Trans. on Software Engineering, 2025, 51(3): 774–777. [doi: 10.1109/TSE.2025.3532391]

[6] Meenakshi B, Das Barman K, Babu KG, Sehgal K. Formal safety analysis of mode transitions in aircraft flight control system. In: Proc. of the 26th IEEE/AIAA Digital Avionics Systems Conf. Dallas: IEEE, 2007. 2.C.1-1–2.C.1-11. [doi: 10.1109/DASC.2007.4391854]

[7] Hönig P, Lunde R, Holzapfel F. Model based safety analysis with smartiflow. Information, 2017, 8(1): 7. [doi: 10.3390/info8010007]

[8] Howard G, Butler M, Colley J, Sassone V. Formal analysis of safety and security requirements of critical systems supported by an extended STPA methodology. In: Proc. of the 2017 IEEE European Symp. on Security and Privacy Workshops (EuroS&PW). Paris: IEEE, 2017. 174–180. [doi: 10.1109/EuroSPW.2017.68]

[9] Ahlbrecht A, Durak U. Integrating safety into MBSE processes with formal methods. In: Proc. of the 40th IEEE/AIAA Digital Avionics Systems Conf. San Antonio: IEEE, 2021. 1–9. [doi: 10.1109/DASC52595.2021.9594315]

[10] Askarpour M, Mandrioli D, Rossi M, Vicentini F. SAFER-HRC: Safety analysis through formal verification in human-robot collaboration. In: Proc. of the 35th Int'l Conf. on Computer Safety, Reliability, and Security. Trondheim: Springer, 2016. 283–295. [doi: 10.1007/978-3-319-45477-1_22]

[11] Mens T, Van Gorp P. A taxonomy of model transformation. Electronic Notes in Theoretical Computer Science, 2006, 152: 125–142. [doi: 10.1016/j.entcs.2005.10.021]

[12] Kahani N, Bagherzadeh M, Cordy JR, Dingel J, Varró D. Survey and classification of model transformation tools. Software & Systems Modeling, 2019, 18(4): 2361–2397. [doi: 10.1007/s10270-018-0665-6]

[13] Boronat A. Expressive and efficient model transformation with an internal DSL of Xtend. In: Proc. of the 21st ACM/IEEE Int'l Conf. on Model Driven Engineering Languages and Systems. Copenhagen: ACM, 2018. 78–88. [doi: 10.1145/3239372.3239386]

[14] Ge N, Jenn E, Breton N, Fonteneau Y. Integrated formal verification of safety-critical software. Int'l Journal on Software Tools for Technology Transfer, 2018, 20(4): 423–440. [doi: 10.1007/s10009-017-0475-0]

[15] Polgreen E, Cheang K, Gaddamadugu P, Godbole A, Laeufer K, Lin SK, Manerkar YA, Mora F, Seshia SA. UCLID5: Multi-modal formal modeling, verification, and synthesis. In: Proc. of the 34th Int'l Conf. on Computer Aided Verification. Haifa: Springer, 2022. 538–551. [doi: 10.1007/978-3-031-13185-1_27]

[16] Paul S, Cruz E, Dutta A, Bhaumik A, Blasch E, Agha G, Patterson S, Kopsaftopoulos F, Varela C. Formal verification of safety-critical aerospace systems. IEEE Aerospace and Electronic Systems Magazine, 2023, 38(5): 72–88. [doi: 10.1109/MAES.2023.3238378]

[17] Graics B, Majzik I. Integration test generation and formal verification for distributed controllers. In: Proc. of the 30th Minisymposium of

the Department of Measurement and Information Systems, 2023. 1–4. [doi: 10.3311/minisy2023-001]

[18] Wang HL, Zhong DM, Zhao TD, Ren FC. Integrating model checking with SysML in complex system safety analysis. IEEE Access, 2019, 7: 16561–16571. [doi: 10.1109/ACCESS.2019.2892745]

[19] Stewart D, Liu J, Cofer D, Heimdahl M, Whalen MW, Peterson M. AADL-based safety analysis using formal methods applied to aircraft digital systems. Reliability Engineering & System Safety, 2021, 213: 107649. [doi: 10.1016/j.ress.2021.107649]

[20] Chen L, Jiao J, Wei QX, Zhao TD. An improved formal failure analysis approach for safety-critical system based on MBSA. Engineering Failure Analysis, 2017, 82: 713–725. [doi: 10.1016/j.engfailanal.2017.06.034]

[21] Fan CC, Qi BL, Mitra S. Data-driven formal reasoning and their applications in safety analysis of vehicle autonomy features. IEEE Design & Test, 2018, 35(3): 31–38. [doi: 10.1109/MDAT.2018.2799804]

[22] Asuelimen G, Blanco-Davis E, Wang J, Yang ZL, Matellini DB. Formal safety assessment of a marine seismic survey vessel operation, incorporating risk matrix and fault tree analysis. Journal of Marine Science and Application, 2020, 19(2): 155–172. [doi: 10.1007/s11804-020-00136-4]

[23] Acreţoaie V, Störrle H, Strüber D. VMTL: A language for end-user model transformation. Software & Systems Modeling, 2018, 17(4): 1139–1167. [doi: 10.1007/s10270-016-0546-9]

[24] Xiao SH, Liu Q, Huang YH, Shi JQ, Guo X. Hierarchical refined modeling and verification method of airborne software using SysML. Ruan Jian Xue Bao/Journal of Software, 2022, 33(8): 2851–2874 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6602.htm [doi: 10.13328/j.cnki.jos.006602]

[25] Shi Z, Han L, Qian Y. A formalization and transformation method of UML model. In: Proc. of the 3rd Int'l Conf. on Computing, Networks and Internet of Things. Qingdao: IEEE, 2022. 191–196. [doi: 10.1109/CNIOT55862.2022.00041]

[26] Wei XM, Dong YW, Sun C, Li XH, Ma JF. Safety analysis for mixed-criticality system with random errors and burst errors based on AADL. Ruan Jian Xue Bao/Journal of Software, 2024, 35(9): 4287–4309 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/7137.htm [doi: 10.13328/j.cnki.jos.007137]

[27] Zhang T, Zhang Y, Yu XF, Wang LZ, Li XD. MDA based design patterns modeling and model transformation. Ruan Jian Xue Bao/Journal of Software, 2008, 19(9): 2203–2217 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/19/2203.htm [doi: 10.3724/sp.j.1001.2008.02203]

[28] Passarini RF, Becker LB, Farines JM. The assisted transformation of models: Supporting cyber-physical systems design by extracting architectural aspects and operating modes from Simulink functional models. In: Proc. of the 2013 III Brazilian Symp. on Computing Systems Engineering. Niteroi: IEEE, 2013. 47–52. [doi: 10.1109/SBESC.2013.25]

[29] Shi YK, Wang Y, Zou N, Yu C, Li L, Li P. Design and practice of model conversion method of multibody mechanical systems based on Simulink. Aerospace Control and Application, 2023, 49(1): 90–97 (in Chinese with English abstract). [doi: 10.3969/j.issn.1674-1579.2023.01.010]

[30] Misson HA, Gonçalves FS, Becker LB. Applying integrated formal methods on CPS design. In: Proc. of the 2019 IX Brazilian Symp. on Computing Systems Engineering. Natal: IEEE, 2019. 1–8. [doi: 10.1109/SBESC49506.2019.9046084]

[31] Todorov V, Boulanger F, Taha S. Formal verification of automotive embedded software. In: Proc. of the 6th Conf. on Formal Methods in Software Engineering. Gothenburg: ACM, 2018. 84–87. [doi: 10.1145/3193992.3194003]

[32] Bouchekir R, Boukhedouma S, Boukala MC. Automatic compositional verification of probabilistic safety properties for inter-organisational workflow processes. In: Proc. of the 6th Int'l Conf. on Simulation and Modeling Methodologies, Technologies and Applications. Lisbon: SciTePress, 2016. 244–253. [doi: 10.5220/0005978602440253]

[33] Xiao MH, Zhao HY, Yang K, Ouyang R, Song WW. A formal analysis method for composition protocol based on model checking. Scientific Reports, 2022, 12(1): 8493. [doi: 10.1038/s41598-022-12448-2]

[34] Marmsoler D. Hierarchical specification and verification of architectural design patterns. In: Proc. of the 21st Int'l Conf. on Fundamental Approaches to Software Engineering. Thessaloniki: Springer, 2018. 149–168. [doi: 10.1007/978-3-319-89363-1_9]

[35] Arcaini P, Riccobene E, Scandurra P. Formal design and verification of self-adaptive systems with decentralized control. ACM Trans. on Autonomous and Adaptive Systems, 2017, 11(4): 25. [doi: 10.1145/3019598]

[36] Jangid MK, Chen GX, Zhang YQ, Lin ZQ. Towards formal verification of state continuity for enclave programs. In: Proc. of the 30th USENIX Security Symp. USENIX Association, 2021. 573–590.

附中文参考文献

[24] 肖思慧, 刘琦, 黄滟鸿, 史建琦, 郭欣. 基于 SysML 的机载软件分层精化建模与验证方法. 软件学报, 2022, 33(8): 2851–2874. http://

www.jos.org.cn/1000-9825/6602.htm [doi: 10.13328/j.cnki.jos.006602]

[26] 魏晓敏, 董云卫, 孙聪, 李兴华, 马建峰. 基于 AADL 的混合关键系统随机错误与突发错误安全性分析. 软件学报, 2024, 35(9): 4287–4309. http://www.jos.org.cn/1000-9825/7137.htm [doi: 10.13328/j.cnki.jos.007137]

[27] 张天, 张岩, 于笑丰, 王林章, 李宣东. 基于 MDA 的设计模式建模与模型转换. 软件学报, 2008, 19(9): 2203–2217. http://www.jos.org.cn/1000-9825/19/2203.htm [doi: 10.3724/sp.j.1001.2008.02203]

[29] 石永康, 王垚, 邹楠, 于超, 李梁, 李鹏. 基于 Simulink 的多体机械系统模型转换方法设计与实践. 空间控制技术与应用, 2023, 49(1): 90–97. [doi: 10.3969/j.issn.1674-1579.2023.01.010]

作者简介

吴梦丹, 博士生, 主要研究领域为软件工程, 软件安全性, 图神经网络.

杨顺昆, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为复杂软件系统可靠性与安全性.

侯展意, 博士生, 主要研究领域为软件缺陷预测, 软件安全性, 机器学习.

佘志坤, 博士, 教授, 博士生导师, CCF 专业会员, 主要研究领域为混成系统自动验证, 智能系统协同控制, 微分方程.

曾福萍, 博士, 助理研究员, 主要研究领域为软件缺陷预测, 软件安全性, 软件测试.

冀振燕, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为人工智能, 区块链, 数据安全, 软件工程.