软件学报 ISSN 1000-9825, CODEN RUXUEW
2025,36(4):1413−1434 [doi: 10.13328/j.cnki.jos.007223] [CSTR: 32375.14.jos.007223]

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

复杂嵌入式系统需求一致性的组合验证方法*

杨 晓[1], 王小齐[1], 陈小红[1], 金 芝[2,3]

[1](上海市高可信计算重点实验室 (华东师范大学), 上海 200062)
[2](高可信软件技术教育部重点实验室 (北京大学), 北京 100871)
[3](北京大学 计算机学院, 北京 100871)
通信作者: 陈小红, E-mail: xhchen@sei.ecnu.edu.cn; 金芝, E-mail: zhijin@pku.edu.cn

摘　要: 形式化方法在需求一致性验证领域已经取得了显著的成就. 然而, 随着嵌入式系统需求复杂度的不断提升, 需求一致性验证面临着状态空间过大的挑战. 为了有效约减验证的状态空间, 同时考虑到嵌入式系统需求所涉及的设备强依赖性, 提出一种复杂嵌入式系统需求一致性的组合验证方法. 它基于需求分解, 识别需求间的依赖关系, 通过这些依赖关系组装验证子系统, 从而实现对复杂嵌入式系统需求的组合验证, 并能初步定位到不一致的需求. 具体而言, 采用问题框架方法对需求进行建模和分解, 并预设领域设备知识库对设备的物理特性进行建模. 在验证子系统的组装过程中, 生成预期软件的行为模型, 并结合物理设备的模型进行动态组装. 最后, 采用航空领域机载侦查系统进行了实例研究, 验证了方法的可行性和有效性, 并通过 5 个案例评估证实了验证状态空间的显著减小. 此方法为复杂嵌入式系统需求的验证提供了一种切实可行的解决方案.
关键词: 复杂嵌入式系统; 需求一致性; 组合验证
中图法分类号: TP311

中文引用格式: 杨晓, 王小齐, 陈小红, 金芝. 复杂嵌入式系统需求一致性的组合验证方法. 软件学报, 2025, 36(4): 1413–1434. http://www.jos.org.cn/1000-9825/7223.htm
英文引用格式: Yang X, Wang XQ, Chen XH, Jin Z. Compositional Verification for Requirements Consistency of Complex Embedded Systems. Ruan Jian Xue Bao/Journal of Software, 2025, 36(4): 1413–1434 (in Chinese). http://www.jos.org.cn/1000-9825/7223.htm

Compositional Verification for Requirements Consistency of Complex Embedded Systems

YANG Xiao[1], WANG Xiao-Qi[1], CHEN Xiao-Hong[1], JIN Zhi[2,3]
[1](Shanghai Key Laboratory of Trustworthy Computing (East China Normal University), Shanghai 200062, China)
[2](Key Lab of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing 100871, China)
[3](School of Computer Science, Peking University, Beijing 100871, China)

Abstract: Formal methods have made significant strides in the field of requirements consistency verification. However, as the complexity of embedded system requirements continues to increase, verifying requirements consistency faces the challenge of dealing with an excessively large state space. To effectively reduce the verification state space, while also considering the strong dependency among devices in embedded system requirements, this study proposes a compositional verification method for ensuring the consistency of requirements in complex embedded systems. This method is based on requirement decomposition and identification of dependencies among requirements. By leveraging these dependencies, it assembles verification subsystems, enabling the compositional verification of complex embedded system requirements and facilitating the initial identification of inconsistencies. Specifically, the problem frames approach is employed for requirement modeling and decomposition, while a domain-specific device knowledge base is utilized for modeling the physical characteristics of devices. During the assembly of verification subsystems, models of expected software behavior are generated and dynamically integrated with physical device models. Finally, the feasibility and effectiveness of this method are validated through a case study of an airborne reconnaissance control system, demonstrating a significant reduction in the verification state space through five

* 收稿时间: 2023-12-31; 修改时间: 2024-02-20, 2024-04-02; 采用时间: 2024-05-10; jos 在线出版时间: 2024-07-03
CNKI 网络首发时间: 2024-07-11

case evaluations. This method thus provides a practical solution for verifying the requirements of complex embedded systems.

Key words: complex embedded system; requirements consistency; compositional verification

众所周知, 软件系统成功开发的先决条件是高质量的需求文档[1]. 一致性验证就是一种确保需求文档高质量的技术. 它也是解决 Zowghi 等人[2]提出的需求的完整性和正确性的关键. 因此, 需求的一致性验证是软件开发中非常重要的一步[3]. 目前, 很多形式化方法被广泛用于需求的一致性验证[3−12], 各种各样的形式化验证语言和工具, 比如, Z3[13]、UPPAAL[14]、Rodin[5], 已经被应用于需求形式化验证中. 形式化方法在需求验证领域已经展示出它们强大的能力[15]. 特别是在安全攸关系统中, 这些方法的应用已经取得了显著的验证效果.

近些年来, 嵌入式系统规模和复杂度呈现出持续增长的趋势[16]. 这一现象给需求一致性验证带来了新的挑战, 即验证状态空间巨大, 出现状态空间爆炸的问题, 使得一致性验证迎来的挑战. 特别地, 在嵌入式系统中, 其软件就是要控制各种各样的物理设备, 以满足用户的需求, 其软件需求描述中涉及大量与物理设备的接口, 由于这些设备存在严格的物理约束, 如控制信号之间的先后顺序关系, 这就要求在需求的一致性验证中必须考虑这些物理约束. 这将使得验证的状态空间进一步扩大, 加剧了验证的复杂度.

鉴于复杂的需求常常被分解为多个子需求, 一种可能实用的降低验证系统状态空间的方法就是使用 Clarke 等人提出的组合验证[11]. 通过将大型系统的验证分解为其子系统的验证, 组合验证可以在需求阶段处理更大规模的形式验证, 检测潜在的错误, 从而防止在后期阶段进行昂贵的修改. 我们前期提出了对需求先分解再进行组合验证的方法[17,18]. 它针对轨道交通的区域控制器的安全需求采用了问题框架[19,20]建模, 并基于投影进行了安全需求分解, 最后再根据子安全需求生成了待验证性质及验证子系统. 这些工作通过需求分解降低了验证模型的复杂度, 减少了验证空间, 缓解了需求验证中状态空间爆炸的问题. 然而, 它们主要是对需求可满足性的验证.

在前期工作的基础上, 本文提出对复杂嵌入式系统进行需求一致性的组合验证, 面临的主要问题是如何有效地构建验证子系统. 理论上来说, 每个验证子系统都应该可以独立运行, 都应该对应一组相关子需求, 以确保需求整体的一致性. 但是由于子需求之间可能存在设备与数据的共享, 每个子需求可能都会隐式地依赖于其他子需求. 因此, 在进行验证子系统构建时, 相关需求组必须考虑到每个子需求及其依赖的需求. 另外, 针对嵌入式系统的特征, 每个验证子系统都必须包含软件构件与设备构件, 以保证设备的物理约束得到满足.

为了解决上述问题, 我们提出了一种复杂嵌入式系统需求一致性的组合验证方法. 它基于需求分解, 识别需求间的依赖关系, 通过这些依赖关系组装验证子系统, 利用现有的一致性验证方法实现对复杂嵌入式系统需求的组合验证, 并能初步定位到不一致的需求. 特别地, 为了保证每个验证子系统中物理设备的特性的不变性, 我们预定义了其行为模型, 将其做成构件, 并通过不同的构件组装所有的验证子系统. 具体而言, 我们采用问题框架方法中的问题图以及前期提出的情景图[21]对需求进行建模和分解, 并预设领域设备知识库对设备的物理特性进行建模. 其中, 由于物理设备的因果性及时间敏感性, 我们采用时间自动机[14]对其进行行为建模与验证. 在验证子系统的组装过程中, 根据需求生成预期软件的行为模型, 并结合物理设备的模型进行构件级的动态组装. 这种组合方法有效地约减了每个验证子系统的验证状态空间, 并能定位到不一致性.

本文的主要贡献包括: 提出了复杂嵌入式系统需求一致性的组合验证方法框架. 首先进行子需求依赖关系的识别, 并根据依赖关系和预设的知识库动态组装验证子系统, 能够有效约减需求一致性验证系统中的状态空间, 从而使原本无法进行验证的复杂需求也能够进行有效验证. 根据嵌入式系统子需求间的设备和数据共享情况, 我们提出了一种基于模型的嵌入式系统需求依赖关系识别方法. 这种方法可以有效地识别相关需求组, 为需求一致性的组合验证中验证子系统的有效构建提供了依据.

本文第 1 节对相关工作进行比较. 第 2 节介绍预备知识. 第 3 节给出方法框架与设备知识库. 第 4 节描述方法细节. 第 5 节以航天领域机载侦查系统为例进行案例研究, 对本文方法进行评估和讨论. 最后, 第 6 节总结全文并展望未来工作.

1 相关工作

目前有很多关于嵌入式系统需求一致性验证的工作, 它们使用各种形式的需求描述语言进行需求建模, 再将

其转换为形式化语言模型, 最后对其采用形式化手段进行验证. 这些方法关注如何将非形式化或半形式化需求模型转化为形式化模型. 一些研究聚焦于 UML 或 SysML 相关的模型与其扩展. 例如, 黄友能等人使用扩展后的 UML 顺序图对需求进行半形式化建模, 最后转换成线性混成自动机模型, 并使用 BACH 工具进行相关性质的验证[4]. Fotso 等人[5]使用 SysML/KAOS 对系统需求进行半形式化建模, 然后将其转化为 B 系统规范[22], 最后用 Rodin 工具验证需求一致性. 杨璐等人[6]采用半形式化方法消息顺序图 (MSC)[23]对需求进行建模, 然后再将 MSC 模型转换为时间自动机, 最后用 UPPAAL 工具[14]进行一致性验证. 我们前期的工作将 UML 顺序图映射至时钟约束规约语言 (clock constraint specification lanuage, CCSL), 并提出了基于 SMT 和基于时钟图的验证方法[7].

也有工作基于其他的描述. 如李腾飞等人[8]提出了一种转换方法, 将 SCADE[17]模型描述的功能需求内容转换为 Lustre[24]形式, 再使用 Jkind 工具进行验证. Vuotto 等人[9]提出的需求一致性检查的工具 ReqV, 使用一组结构化自然语言表达的需求作为输入, 将其翻译成 LTL 语言后使用 SPECPRO 进行验证. Wang 等人[10]将自然语言需求转换为命题投影时间逻辑 (PPTL), 再使用 PPTLSAT 工具对其进行冲突检测. Vidal 等人[12]提出了一种领域特定语言 GIRL, 用于描述软件需求结构不变量、实体以及它们之间的关系, 然后再将 GIRL 模型转换为 Alloy 规约, 使用 Alloy SAT 求解器自动验证需求一致性. 我们还提出了一种模式语言 SafeNL 来表达安全需求, 将 SafeNL 语句转换为 CCSL 后使用 MyCCSL 工具进行形式化验证[3]. 但是, 上述研究工作都聚焦于验证结果的正确性, 并未关注如何约减验证系统的状态空间.

形式化方法也在探索约减验证系统状态空间的方式. 例如, 统计模型检测方法不遍历整个状态空间, 而是通过蒙特卡罗模拟的方法估计性质满足或不满足的概率[25]. Ramesh 等人[26]提出一种新的采样方法为蒙特卡罗模拟提供包含知识更多的样本, 从而减少需要枚举的样本数量, 进而起到验证状态空间约减效果. Zhu 等人[27]针对线性时序逻辑 (LTL) 模型检查状态空间爆炸问题, 将机器学习分类算法引入进来, 提取 LTL 公式特征并输入机器学习二元分类器得到一个可接受的预测验证结果. Clarke 等人[11]主张使用组合模型验证思维约减验证状态空间. 组合模型检查不直接验证复杂的程序, 而是单独验证其并行过程, 然后通过检查单个过程的属性来推导组合的属性, 进而实现加速验证的目标[28,29]. Namjoshi 等人[30]将其扩展到参数化通信模型检查中, 以验证分布式网络协议和共享内存并发程序. Zhou 等人[31]提出基于验证性质来约减 SOPC 变量、分支数量的方法, 删减与待验证性质无关的变量、分支, 得到验证所需的最小变量、分支集合. Shen 等人[32]提出了一种通过值约减和转换关系约减的方法, 通过去除变量的冗余值, 去除与被验证变量无关的转换来实现. 但这些方法大部分并没有应用于需求的形式化验证中, 特别地, 其中的组合验证以分解为前提, 并不关心怎么分解.

在需求形式化验证中, 也已经存在一些组合验证方法来约减状态空间. 例如, Yuan 等人[17]分别基于问题框架和约束提出了两种投影方法. 基于问题框架的方法通过功能分解将系统分解为子属性, 而基于约束投影的方法则消除了冗余变量, 两种方法均可减小验证空间. 刘筱珊等人[18]提出对轨道交通区域控制器系统进行问题框架需求建模, 基于投影对需求进行分解, 并将子需求自动生成安全需求验证模型, 进行组合验证. 尽管这些工作都隶属组合验证的范畴且都有效减少了验证空间, 但它们都是需求的可满足性验证, 并没有针对需求的一致性验证.

2 预备知识

2.1 运行案例介绍

在详细阐述本文方法之前, 首先介绍一个简单的智能家居系统的案例, 以展示需求一致性验证的全过程, 其问题陈述如下.

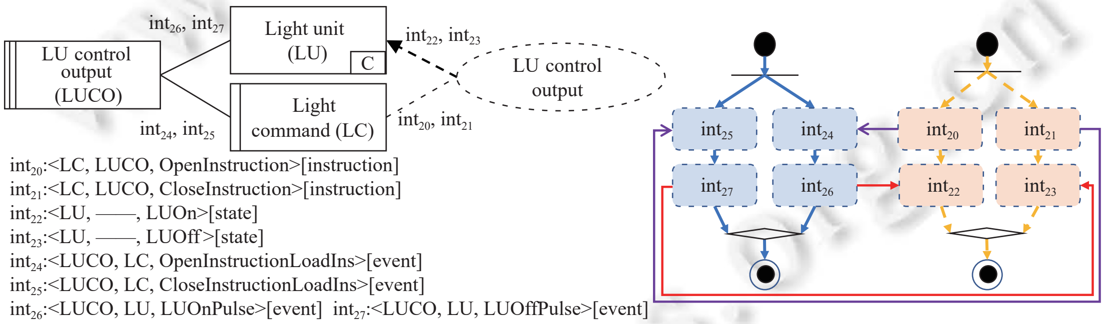
智能家居系统问题: 需要一个计算机系统来控制灯光和空调, 该系统涉及到灯光单元 (light unit, LU)、空调 (air condition, AC)、亮度传感器 (light sensor, LS)、温度传感器 (temperature sensor, TS)、脉冲生成器 (pulse generator, PG) 设备以及灯光命令 (light command, LC)、空调命令 (AC command, ACC) 数据存储. 该系统共有 6 个需求, 分别为初始化 (initialization)、自动存储 (automatic save)、命令存储 (commanded save)、灯光单元控制输出 (LU control output)、温度控制 (temperature control)、空调控制输出 (AC control output). 其中, 初始化需求用来

初始化灯光单元和空调设备; 自动存储需求要求软件从亮度传感器获取环境亮度信息, 根据预先设置的灯光控制规则, 将开灯或关灯指令保存至灯光命令数据存储中; 命令存储需求用来接收脉冲生成器发出的开关灯命令, 并将开灯或关灯指令保存至灯光命令数据存储中; 灯光单元控制输出需求负责将灯光命令数据存储中的开关灯指令转换为信号并发给灯光单元, 用来实现对灯光单元的开关控制; 温度控制需求要求软件从温度传感器获取环境温度信息, 根据预先设置的空调控制规则, 将制冷或制热指令保存至空调命令数据存储中; 空调控制输出需求负责将空调命令数据存储中的空调操作指令转换为信号并发给空调设备, 用来实现对空调的制冷/制热控制.

2.2 问题图和情景图

问题图[32]是问题框架方法中需求描述的结果, 用领域 (domain)、需求 (requirement) 和交互 (interaction) 来对问题进行表达. 其中, 领域可分为机器领域 (machine domain) 和问题领域 (problem domain), 机器领域就是待构建的软件系统, 用双竖线矩形框表示, 问题领域指与软件系统存在交互的外部实体. 根据外部实体的特性, 又可将问题领域分为因果领域、词法领域和自主领域. 因果领域指领域现象之间存在可预测的因果关系, 如受控设备. 词法领域指数据的物理存储, 如数据库. 自主领域的行为现象无法准确预测, 例如人类. 在问题图中, 因果领域和自主领域用矩形框表示, 词法领域用单竖线矩形框表示. 机器领域与问题领域之间的交互称为行为交互, 用五元组<Id, Ini, Rec, Phe, Typ> 表示, 其中, Id 代表交互编号, Ini 代表交互的发起方, Rec 代表交互的接收方 (可为空集), Phe 代表交互现象, Typ 代表现象类型, 包括事件 (event)、状态 (state) 和值 (value). 此外, 问题图用虚线椭圆表达需求、需求对问题领域现象的引用或约束. 需求引用 (reference) 表示需求仅仅“提到”现象, 而需求约束 (constraint) 则表示要要求该现象发生. 需求引用用无箭头虚线表示, 需求约束用带箭头虚线表示. 需求引用或约束上的交互称为期望交互, 表示与行为交互相同.

图 1(a) 是一个问题图例子, 该例子来自智能家居领域. 灯光单元控制输出需求要求软件从词法领域灯光命令获取存储的灯光开关指令, 然后将指令转换为脉冲发给灯光单元. 其中, 控制输出 (control output) 是待开发的软件, 灯光单元属于因果领域, 灯光命令属于词法领域. 图中有 4 个行为交互 int_{24}–int_{27}、4 个期望交互 int_{20}–int_{23}. 需求控制输出引用灯光命令领域, 约束灯光单元领域.



(a) 灯光单元控制输出问题图　　(b) 灯光单元控制输出情景图

图 1　问题图与情景图例子

情景图[25]是交互流, 描述了每个需求下各交互之间的关系, 其基本表示采用活动图的形式. 情景图节点可分为控制节点与交互节点, 其中控制节点包括开始节点 (实心圆)、结束节点 (圆环)、决策节点 (菱形)、分支节点 (短线)、合并节点 (菱形)、延时节点 (沙漏型)、指定时间节点 (沙漏型). 交互节点包括行为交互节点 (实线边框圆角矩形) 与期望交互节点 (虚线边框圆角矩形). 例如图 1(b) 中的 int_{24}–int_{27} 为行为交互节点, int_{20}–int_{23} 为期望交互节点. 情景图边可分为行为序关系、期望序关系、行为使能关系、需求期望关系、同步关系. 其中, 行为序关系表示行为交互节点之间或行为交互节点与控制节点之间的先序关系, 用行为交互节点/控制节点指向行为交互节点/控制节点的实线箭头表示, 如图 1(b) 中 int_{25} 与 int_{27} 之间的关系. 期望序关系表示期望交互节点之间或期望交互节点与控制节点之间的先序关系, 用期望交互节点/控制节点指向期望交互节点/控制节点的虚线箭头表示, 如图 1(b)

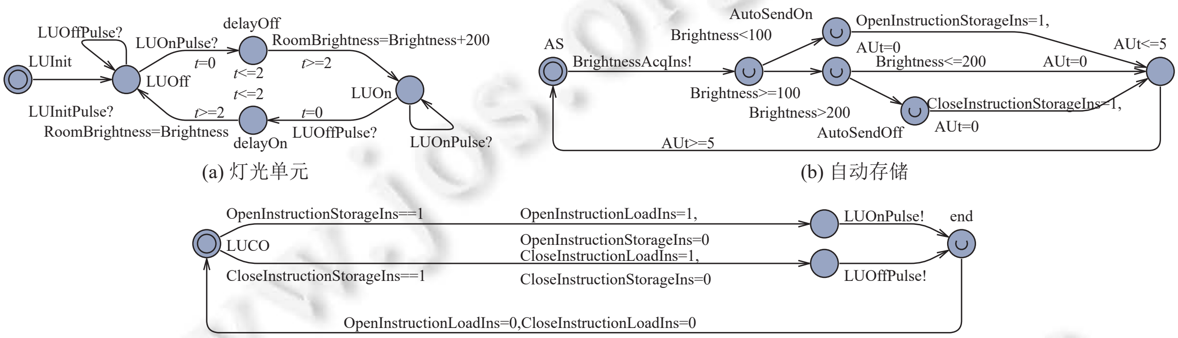
中 int_{20} 与 int_{22} 之间的关系. 行为使能关系表示行为交互节点使能期望交互节点, 用行为交互节点指向期望交互节点的实线箭头表示, 如图 1(b) 中 int_{26} 与 int_{22} 之间的关系. 需求期望关系表示期望交互节点要求行为交互节点发生, 用期望交互节点指向行为交互节点的实线箭头表示, 如图 1(b) 中 int_{20} 与 int_{24} 之间的关系. 同步关系表示行为交互节点与期望交互节点同时发生, 用无箭头实线表示.

2.3 时间自动机

我们使用时间自动机 (TA) 作为形式化验证模型, 其验证系统模型采用一组并发的时间自动网络 (NTA) 来表示. 时间自动机的定义如下[14].

定义 1. 时间自动机是一个元组 $(L, l_0, X, \Sigma, E, I)$, 其中 L 是一个有限的位置 (状态) 集; $l_0 \in L$ 是初始位置; X 是有限时钟集; Σ 是动作的有限集合, 例如同步; $E \subseteq L \times \Sigma \times G(X) \times 2^X \times L$ 是一个有限的边集, 边上有动作、守卫以及待重置的时钟集, 其中 $G(X)$ 是 X 上的守卫集; $I:L \rightarrow G(X)$ 为每个位置分配一个不变式.

我们以图 2(a) 中的灯光单元为例来解释 TA. 在 LightUnit 中有 5 个位置, 即 LUOff、LUOn 和 LUInit 以及另外两个位置 delayOff 和 delayOn 用以表示响应时间. 在这两个位置中, 有时钟变量 t, 以及不变式 t<=2, 用于限制时钟 t 的值, 时钟变量可以随时间增加. 从 LUOff 到 delayOff 的迁移是通过接收同步信号 LUOnPulse 触发.



(a) 灯光单元　(b) 自动存储　(c) 灯光单元控制输出

图 2　智能家居系统时间自动机模型

在 NTA 中 TA 之间通过共享变量和广播信道进行通信. 共享变量在系统声明时称, 可用于 NTA 中的任何 TA. 信道提供了一种确保不同 TA 中转换同步的方法. 一旦同步信号发送 (!) 完成, 同步信号接收 (?) 也立即响应. 例如, 图 2 给出了灯光控制器例子中的 3 个 TA, 灯光单元与自动存储共享环境变量亮度 (brightness), 灯光单元与控制输出通过同步信号 LUOnPulse 和 LUOffPulse 进行通信. 控制输出发送 (!)LUOnPulse 同步信号, 而灯光单元接收 (?)LUOnPulse 同步信号.

3 方法框架与设备知识库

3.1 方法框架

为了降低验证状态空间, 我们考虑首先对复杂嵌入式系统需求进行分解, 然后分别对它们进行组合验证. 一般来说, 一个嵌入式系统通常包含软件和多个设备, 如传感器和执行器等. 软件从传感器获得信息, 通过执行器控制外部操作对象, 其需求具有较强的设备依赖性. 这种依赖性使得嵌入式系统的需求相互交织. 这使得我们选择基于情景的问题投影[21,33]作为嵌入式系统需求的分解方式, 它可以将设备和需求一起投影在同一个情景维度下, 使得子需求也保持与设备的关系. 我们将投影后的问题图和情景图组合作为输入. 其次, 在组合验证中, 由于需求之间可能存在设备的共享, 为了保障该设备行为的约束, 我们提出对该设备进行预建模, 在验证的时候对其验证系统进行动态组装. 最后, 在验证系统组装中, 由于设备和数据的共享, 造成了需求之间的数据和控制依赖, 这些依赖使得被依赖需求必须与依赖需求一起运行, 才能组装成可以独立运行的验证子系统. 因此, 我们设计了 4 步的复杂嵌入

式系统需求一致性的组合验证方法框架, 如图 3 所示. 首先根据问题图和设备知识库识别系统需求之间的依赖关系, 其次根据依赖关系生成验证子系统架构, 明确每个验证子系统模型所需的构件, 第 3 步根据情景图生成每个构件的时间自动机表示, 与领域设备知识库一起生成可执行验证子系统模型集合. 最后对可以执行验证子系统进行一致性验证, 对验证结果进行综合并进行不一致的定位.

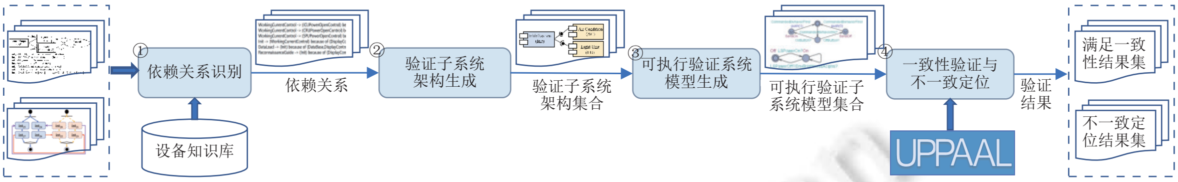


图 3　方法框架图

第 1 步: 需求依赖关系识别. 识别子需求之间由于数据和设备共享造成的依赖关系, 包括数据依赖和控制依赖. 其中数据依赖是由于数据共享造成的, 主要刻画了有的需求“消费”数据而必须依赖有些需求“生产”数据的情况. 控制依赖由设备共享造成的, 刻画了不同需求中设备状态变化导致的“后面”状态的需求依赖“前面”状态需求的情况. 识别这些依赖关系需要知道不同需求对词法领域的操作以及设备的状态.

第 2 步: 验证子系统架构生成. 目的是要明确验证子系统的组成构件. 基本原则是每个子需求都应该对应一个验证子系统, 但必须加入被依赖项的需求对应的验证模型. 在每个需求的问题图和情景图中都包含了对软件与设备的交互, 我们将其转换为构件和连接件, 以形成验证子系统的架构. 其中构件包括由于依赖关系相互依赖的软件构件, 以及软件构件所需的设备构件, 连接件包括各种构件之间的交互信息.

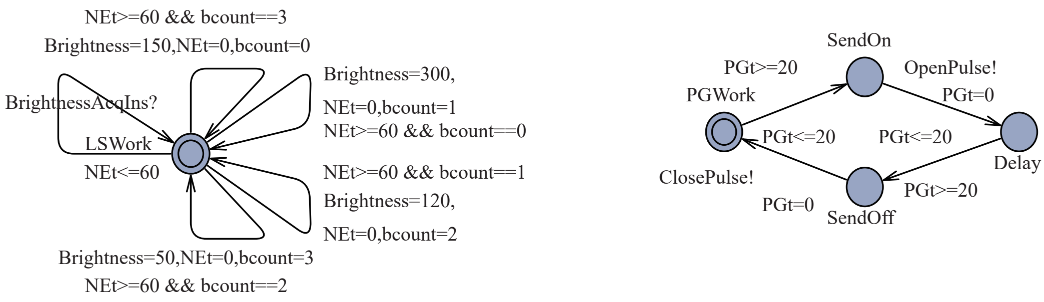
第 3 步: 可执行验证系统模型生成. 在得到的验证子系统架构的基础上, 生成多组可执行的验证子系统模型. 每一组可执行的验证系统模型都要包括每个构件的 TA 模型, 它们之间的连接, 以及对信道、变量以及模型的声明. 可以根据情景图自动生成软件构件的 TA. 可以接从领域设备知识库中选择所需的设备 TA. 对于验证子系统架构中的连接件, 则根据 TA 模型的通信机制生成相应的同步信号和共享变量.

第 4 步: 一致性验证与不一致定位. 我们将生成的多组可执行验证子系统模型通过 UPPAAL 平台进行一致性验证. 本文中将一致性定义为同一设备不能出现调度冲突. 具体的性质可以依据生成的可执行验证系统模型中的设备及设备库中的可能状态进行自动生成. 最后综合所有的可执行验证子系统模型验证结果, 只有所有验证结果都是“一致”的情况下, 最终的结果才为“一致”, 否则为“不一致”. 对所有不满足一致性的需求进行错误定位, 方便用户进行更改.

3.2　设备知识库

设备一般包括传感器和执行器. 传感器用于定期监测外部环境变量或检测人类行为. 为了构建监视属性的传感器 TA, 考虑工作模式和可能的模式转换. 我们将每个工作模式映射到 TA 中的一个位置节点. 通常, 传感器有两种工作模式, 关闭和打开. 工作模式的转换可以是时间驱动的, 也可以是事件驱动的. 同时传感器具有更新它们所监视的环境变量的自迁移. 例如, 在图 4(a) 中, 亮度传感器通过触发同步信号“BrightnessAcqIns”获取环境亮度. 对于用于检测人类行为的传感器 TA, 建立模型中的位置节点以及相应的迁移与前述过程非常相似. 唯一的不同在于需要建模涉及相关的人类指令.

执行器通常由控制信号触发, 并以特定的工作模式运行. 这些模式是明确定义的, 例如开或关. 在 TA 建模中, 执行器的工作模式可以直接映射到位置节点, 由映射的迁移触发状态转换. 例如, 图 4(b) 中脉冲生成器有两种工作模式, 发送打开脉冲 (SendOn) 和发送关闭脉冲 (SendOff), 这对应到 TA 模型中的两个不同位置节点 SendOn 和 SendOff. 执行器通常具有响应时间, 我们可以在执行器接收到同步信号后、达到预期状态前添加待定位置节点, 如图 4(b) 中, 位置节点 PGWork 和 Delay 具有不变式“PGt<=20”, 用来表达接收开关脉冲信号响应时间 20 s 后才进入对应的发送开关脉冲状态位置节点.



(a) 传感器: 亮度传感器 (b) 执行器: 脉冲生成器

图 4 智能家居系统设备 TA 举例

4 方法细节

4.1 依赖关系识别

需求依赖关系分为两类: 数据依赖和控制依赖.

(1) 数据依赖识别

数据依赖指两个或多个需求共享同一个词法领域, 一个需求“生产”数据, 另一个需求“消费”数据. 数据依赖发生在那些存在对同一数据读写操作的需求之间. 对于使用问题框架描述的需求模型, 若需求 R_1 的交互 int_i 表达了从词法领域 ds 取出数据 data 的行为, 而需求 R_2 的交互 int_j 表达了将数据 data 保存到词法领域 ds 的行为. 那么需求 R_1 由于数据 data 而数据依赖于 R_2, 记为“R_1->R_2, because of {data}”. 例如, 图 5 展示了命令保存与灯光单元控制输出需求间的数据依赖关系. 命令保存需求需要软件接收脉冲生成器发出的开关灯信号, 再将信号转换为灯光开关指令 (OpenInstruction, CloseInstruction) 保存到灯光命令词法领域中. 灯光单元控制输出需求要求软件从灯光命令领域中取出保存的灯光开关指令, 将其转换为灯光单元能够识别的信号格式后再发给灯光单元, 从而实现开灯或关灯效果. 图 5 案例存在数据依赖“LU control output -> Commanded save, because of {OpenInstruction, CloseInstruction}”, 表示灯光单元控制输出因为打开指令 (OpenInstruction)、关闭指令 (CloseInstruction) 而数据依赖于命令保存.

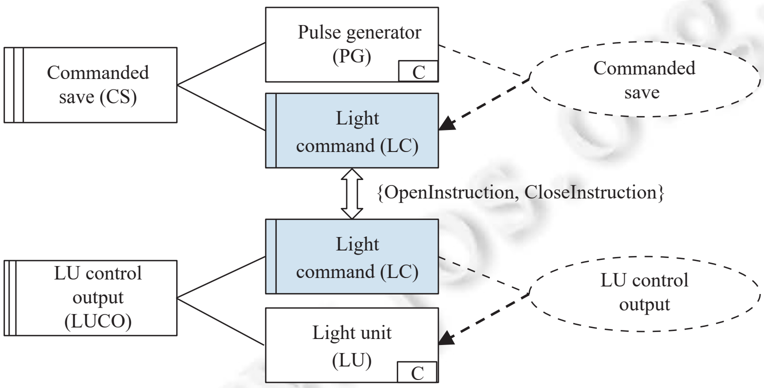


图 5 数据依赖例子

(2) 控制依赖识别

控制依赖是指两个或多个需求共享同一个设备, 由于设备状态之间的关系导致的需求之间的先后关系. 这种先后关系可以通过触发设备发生状态变迁的交互和该需求的前置条件进行识别. 其中需求的前置条件是指需求中涉及的设备在进入该需求之前的状态. 该条件需要在问题图中进行补充. 根据需求间交互和前置条件之间的关系,

控制依赖可以分为 3 种情况.

情况 1. 一个需求使得另外一个需求的前置条件发生: 若需求 R_1 发生的前置条件为设备 dev 处于状态 s, 而综合设备知识库中该设备 TA 可知, 能够令该设备进入状态 s 的交互只有需求 R_2 的交互, 那么需求 R_1 因为设备 dev 的状态 s 而控制依赖于 R_2, 记为 $<R_1$, dev.s, $R_2>$.

情况 2. 需求前置条件之间存在关系: 若需求 R_1 的前置条件为设备 dev 处于状态 s_m, 需求 R_2 发生的前置条件为设备 dev 处于状态 s_n, 并且根据设备知识库中该设备 TA 可知, 从初始状态出发到 s_m 的所有路径中都包含位置 s_n, 即若要到达 s_m 必须先经过 s_n, 那么需求 R_1 因为设备 dev 的状态 s_m、s_n 而控制依赖于 R_2, 记为 $<R_1$, {dev.s_m, dev.s_n}, $R_2>$.

情况 3. 需求引发的设备状态变迁之间存在关系: 若需求 R_1 的期望交互 int_i 驱使设备 dev 进入状态 s_m, 需求 R_2 的期望交互 int_j 驱使设备 dev 进入状态 s_n, 并且根据设备知识库中该设备 TA 可知, 从初始状态出发到 s_m 的所有路径中都包含 s_n, 即若要到达 s_m 必须先经过 s_n, 那么需求 R_1 因为设备 dev 的状态 s_m、s_n 而控制依赖于 R_2, 记为 $<R_1$, {dev.s_m, dev.s_n}, $R_2>$.

例如, 图 6 展示了初始化与灯光单元控制输出两个需求间的控制依赖关系. 初始化需求驱使灯光单元从灯光单元初始化 (LUInit) 状态进入灯光关闭 (LUOff) 模式. 控制输出需求可以驱使灯光单元设备进入灯光关闭或灯光打开 (LUOn) 状态, 并且在设备知识库灯光单元自动机模型中, 从初始节点到灯光打开状态的所有路径中都包含灯光关闭状态, 符合情形 3, 因此存在控制依赖<LU control output, {LU.LUOff, LU.LUOn}, Initialization>.

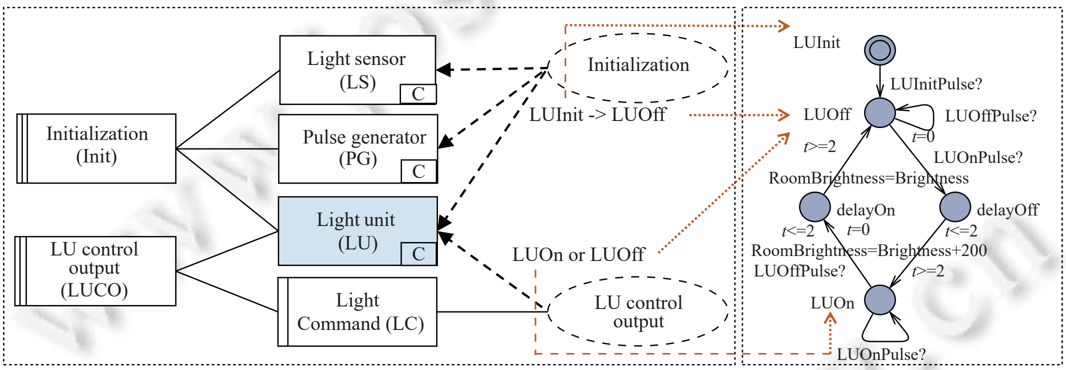


图 6　控制依赖例子

4.2 验证子系统架构生成

验证子系统架构生成的目的就是基于识别出的依赖关系将待验证需求分解为多个相关需求组, 并据此定义验证子系统, 明确每个验证子系统所包含的构件. 通常, 验证子系统的数量与子需求的个数相同. 为了确保每个子需求都满足待验证的属性, 我们应为每个子需求分配一个验证子系统. 然而, 由于子需求之间存在各种依赖关系, 每个验证子系统应包含其对应子需求所依赖的子需求, 以确保其能真正实现. 这也可能导致部分子系统是重复的, 因此, 实际子系统的数量需要剔除重复的子系统. 在验证子系统的构成方面, 根据嵌入式系统的架构, 验证子系统通常包含软件组件、设备组件以及组件之间的连接器. 为了表达验证子系统架构, 我们使用 UML 组件图, 将输入问题图的各个领域转换为组件图中的不同组件, 并将行为交互转换为组件图中的连接器.

具体验证子系统架构生成过程为: 首先每个子需求对应一个验证子系统. 除此之外, 该验证子系统还包括该子需求所依赖的其他子需求. 当然这个过程中可能存在子需求不依赖于其他任何子需求. 最后验证子系统还需要加入与该子系统中所有需求相关的设备, 即与需求对应的问题图中的机器领域存在行为交互关系的因果领域或词法领域. 接下来, 我们可以根据所有上述相关要素来比较两个子系统是否重复. 如果发现有重复的子系统, 我们将只保留其中一个.

接下来对各个验证子系统生成其架构图, 即将对应的问题图转换成 UML 构件图的表达方式. 问题图中机器领域是指待构建的软件系统, 因此将问题图中的机器领域转换成软件构件. 问题图中的问题领域指与软件系统存在交互的外部实体. 根据问题框架方法, 问题领域可以分为因果领域、词法领域和自主领域, 因果领域为受控领域, 词法领域是指数据存储域, 我们统一将因果领域和词法领域转换成设备构件, 表示该领域具有明确的因果关系. 对于自主领域, 指不可控、不可预测的外部环境, 由于嵌入式系统中, 软件要通过设备与外部环境进行交互, 我们在领域设备知识库的构建过程中, 已经将外部环境的各种属性建模在设备模型中, 因此不再考虑自主领域. 对于机器领域与问题领域之间的行为交互, 我们将其转换为软件构件与设备构件之间的连接件. 由此得到问题图转换为构件图的对应关系, 如图 7 所示.

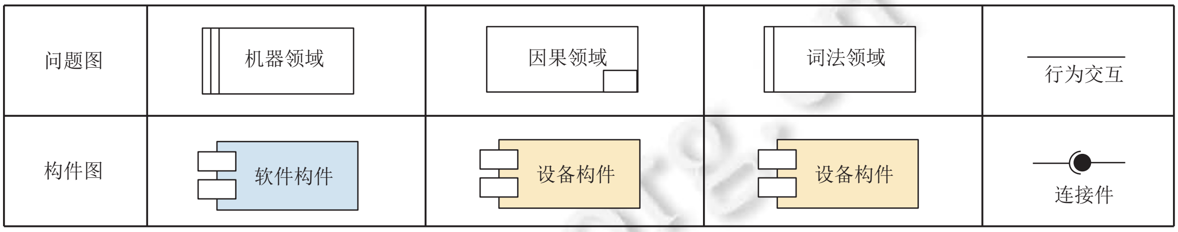


图 7　不同问题图结构对应的构件图结构

下面以智能家居系统为例介绍验证子系统架构生成过程. 如图 8 所示, 在智能家居系统例子中有 6 个子需求, 则得到 6 个验证子系统分别为初始化、自动存储、命令存储、温度控制、灯光单元控制输出和空调控制输出子系统. 以灯光单元控制输出子系统为例, 根据其控制依赖<LU Control Output, {LU.LUOff, LU.LUOn}, Initialization> 和数据依赖<LU Control Output, {OpenInstruction, CloseInstruction}, Commanded Save> , <LU Control Output, {OpenInstruction, CloseInstruction}, Automatic Save>. 可知其验证子系统包含初始化、命令存储、自动存储和灯光单元控制输出需求, 对应为问题图中机器领域, 因此转换成构件图中的软件构件. 与这些需求对应的机器领域存在行为交互的因果领域和词法领域为空调、灯光单元、灯光传感器、脉冲生成器和灯光命令, 因此转换为构件图中的设备构件, 并将行为交互转换成连接件.

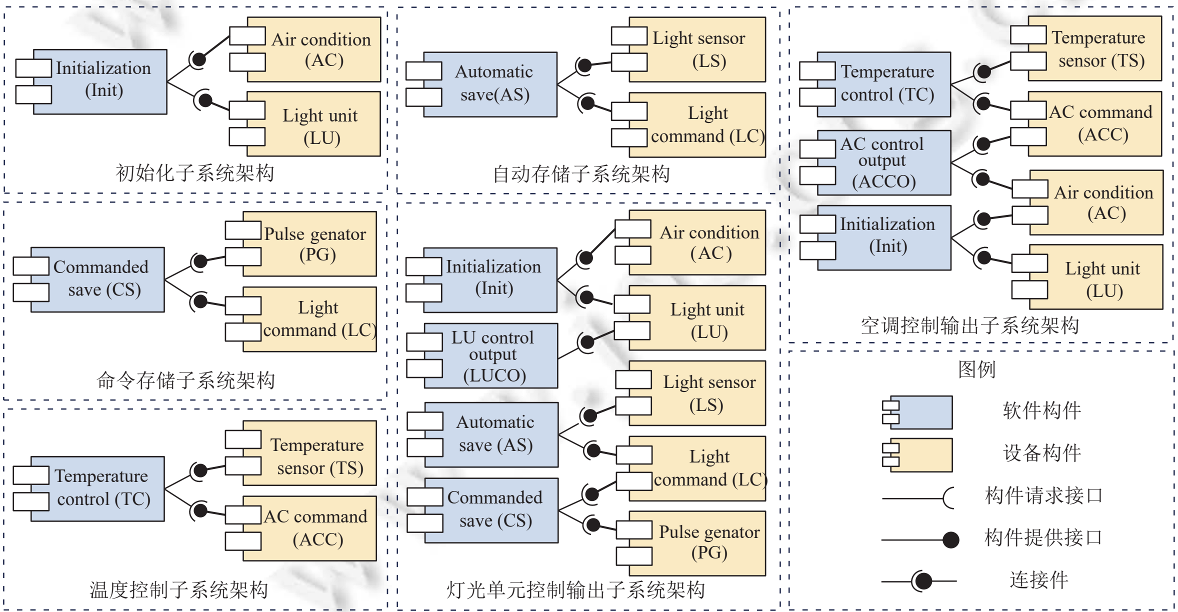


图 8　智能家居系统验证子系统架构图

4.3 可执行验证系统模型生成

这一步就是要根据生成的验证子系统架构生成可以在 UPPAAL 平台执行验证的 NTA. 每个 NTA 都包括生成验证子系统架构中的软件构件、设备构件以及连接件转换生成对应的 TA. 其具体过程如下.

首先, 我们将验证子系统架构中的设备构件转换为 TA 模型. 这些设备构件主要来源于问题图的因果领域和词法领域. 对于由因果领域转换而来的设备构件, 它们属于受控设备, 可以直接从领域设备知识库中获取其 TA 模型. 然而, 来自词法领域的设备构件则代表数据存储域. 在 TA 模型中, 数据存储类的物理设备主要通过更新变量来实现数据的存储. 在验证模型的过程中, 我们并不过分关注数据本身, 而是更关注整个系统的运行. 因此, 对于数据存储域类的构件, 我们仅在与其通过连接件连接的软件构件上做出相应处理, 而不单独建立 TA 模型.

接下来为软件构件生成对应的 TA 模型. 这一过程将情景图中的软件行为生成对应的 TA 模型. 需要分别考虑情景图中包含的顺序结构、选择结构、分支结构、并行结构、循环结构和时间约束. 每种情景图结构对应转换成 TA 模型结构如图 9 所示. 对于顺序结构, 表示的是两个行为交互之间的顺序关系, 因此对于每个行为交互节点, 在 TA 中建立一个位置, 并建立从前一个位置到该位置的迁移, 迁移上的约束条件根据交互对象类型以及行为交互的类型建立对应的约束条件. 对于选择结构, 首先标记选择结构开始位置, 如 start, 然后建立两个位置节点, 并建立从 start 到这里两个位置的迁移, 迁移上的约束条件根据选择结构中的约束条件, 一个为满足该条件的约束, 另一个即为不满足该条件的约束, 然后根据其后续结构建立对应模型, 最后汇合两条分支到一个位置节点, 如 end. 对于分支结构, 首先也是标记其分支开始位置, 如 start, 其可能具有多条分支, 对于每条分支都建立一个位置节点, 建立从 strat 到每个分支位置节点的迁移, 迁移上的约束条件根据交互对象类型以及行为交互的类型进行建立, 最后所有分支到一个位置节点.

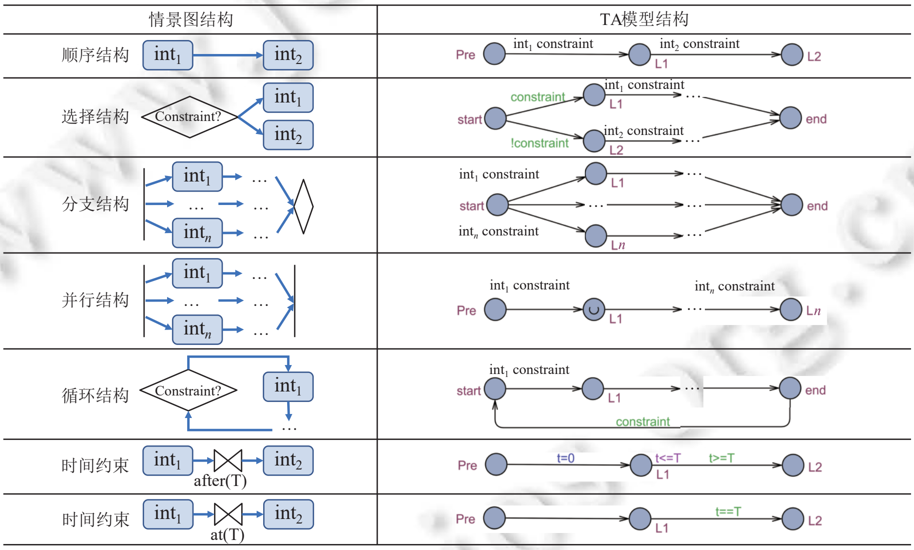


图 9　不同情景图结构对应的 TA 模型结构

对于并行结构, 表达的是所有分支上并发执行. 我们使用紧急位置节点 (位置节点图标中带有 U 的位置节点, 表示在该位置上没有时间流逝) 表示进入紧急位置节点的迁移以及离开紧急位置节点的迁移是同时发生的. 因此对于并行结构, 我们在 TA 中, 建立每个行为交互的紧急位置节点, 并建立从前一位置节点到该位置的迁移, 迁移约束条件根据交互对象类型以及行为交互类型建立. 对于循环结构, 建立 TA 模型, 首先标记循环开始位置节点, 如 start, 然后根据循环内的行为交互建立对应的位置节点及到该位置的迁移约束条件, 最后建立从循环内最后一个位置节点到开始位置节点的迁移, 迁移上的约束条件根据循环结构内的约束条件生成. 对于时间约束, 情景图中有两类时间约束, 一类表达延迟, 即一个行为交互发生后间隔多久时间再发生下一个行为交互, 另一类表达某个行

为交互在指定的时刻发生. 对于延迟的时间约束, 我们对前一个行为交互对应的位置节点加入一个不变式 $t<=T$ 表示在 T 时间内不能离开该位置节点, 并在进入该位置节点的迁移上初始化时钟变量 t, 再离开该位置节点的迁移上加入守卫条件 $t>=T$. 对于指定时刻发生的时间约束, 我们在进入指定时刻发生的行为交互对应的位置节点的迁移上加入守卫条件 $t==T$, 表示指定时刻才能发生对应的行为交互转换的约束条件.

特别地, 表 1 中给出了交互对象类型以及行为交互现象类型生成迁移上约束条件的对应关系. 对于与因果领域的行为交互, 如果现象类型是值或状态, 在迁移的更新条件中添加 int_x(Phe)=1, 并声明变量 int int_x(Phe) 意思是将对应的行为交互 int_x 的现象 Phe 声明为一个 int 变量, 并将在迁移上将其值更新为 1, 通过将其值设置为 1 来标识这个值/状态代表的数据被获取, 用来代表数据获取过程. 对于与因果领域的行为交互, 如果现象类型是事件, 在迁移上添加同步信号, 如果是发送信号, 则添加 int_x(Phe)!, 如果是接收信号, 则添加 int_x(Phe)?, 并声明信道 chan int_x(Phe). 对于与词法领域的行为交互, 无论哪种现象类型, 在迁移的更新条件中添加 int_x(Phe)=1, 并声明变量 int int_x(Phe) 意思是将对应的行为交互 int_x 的现象 Phe 声明为一个 int 变量, 并将在迁移上将其值更新为 1, 通过将其值设置为 1 来标识这个值/状态代表的数据被存储或者获取, 用来代表数据存储或者数据获取过程.

表 1　不同交互对象类型和行为交互现象类型对应的迁移约束条件

参与对象	现象类型	迁移约束条件
因果领域	值/状态	在迁移中添加更新条件, “int_x(Phe)=1”, 声明变量“int int_x(Phe)”
	事件	在迁移中添加同步信号, 如果是发送同步信号, 添加“int_x(Phe)!”, 如果是接收同步信号, 添加“int_x(Phe)?”, 声明信道“chan int_x(Phe)”
词法领域	值/状态/事件	在迁移中添加更新条件, “int_x(Phe)=1”, 声明变量“int int_x(Phe)”

对于构件图中的连接件, 在根据软件构件生成 TA 过程中, 将连接件代表的构件之间的行为交互转换成 TA 中的通信机制. 在 TA 中, 通过共享变量实现数据的通信, 通过同步信号实现控制器与设备之间的调度. 对于软件构件和设备构件之间的连接件, 根据其代表的行为交互类型使用不同的通信机制. 特别的, 对于数据存储域一类的设备构件, 并没有设备 TA 供软件控制器使用同步信号进行通信, 因此软件构件 TA 与数据存储域之间只通过共享变量来实现通信. 具体 TA 模型间的通信机制如表 2 所示.

表 2　TA 模型通信机制

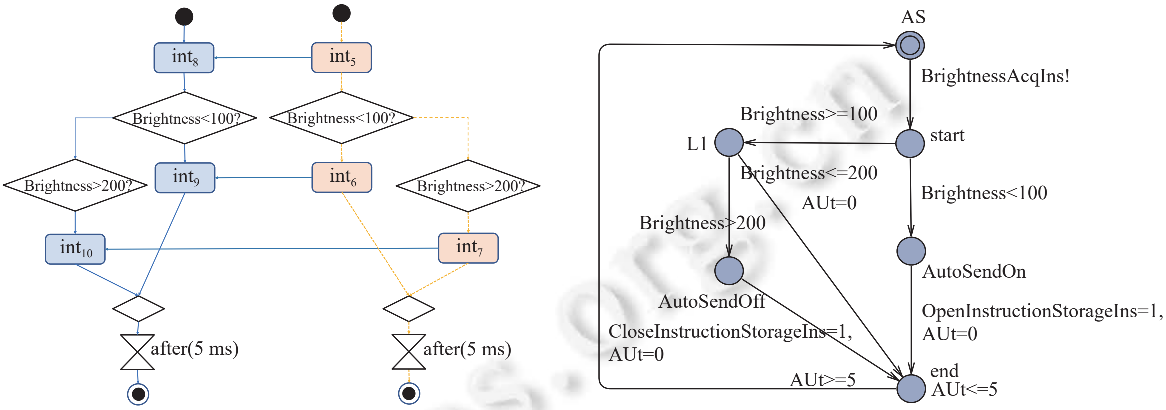
参与对象I	参与对象II	现象类型	TA通信机制
机器领域	因果领域	值/状态	共享变量
		事件	同步信号
机器领域	词法领域	值/状态/事件	共享变量

最后, 对获取的设备 TA 和生成的软件控制器 TA 进行模型声明, 变量和信道声明. 模型声明在 UPPAAL 工具的模型声明界面进行添加, 具体为使用 system 关键字后跟每个模型名称, 每个模型名称用“,”分隔, 最后以“;”结尾. 对于信道变量声明使用关键字 chan, 整型变量声明使用关键字 int.

下面以智能家居系统中的灯光单元控制输出子系统架构为例介绍生成可执行验证子系统模型的过程. 该子系统中的设备构件包括空调、灯光单元、亮度传感器、脉冲生成器以及灯光命令. 在领域设备知识库中获取设备构件的 TA 模型, 其中灯光命令为词法领域转换来的设备构件, 在领域设备知识库中没有其 TA 模型. 灯光单元 TA 如图 2(a) 所示, 亮度传感器以及脉冲生成器 TA 如图 4 所示, 空调与灯光单元控制输出无关, 只是在初始化需求中对空调进行初始化, 因此其 TA 不再展示. 其软件构件包括初始化、自动存储、命令存储以及灯光单元控制输出, 以自动存储软件构件为例, 其输入的情景图如图 10(a) 所示. 该情景图中存在顺序结构、选择结构以及时间约束. 将其转换成 TA 模型的过程如下.

首先在 TA 中建立初始位置, 命名为 AS, 然后根据情景图结构生成 TA. 对于情景图首先是行为交互 int_8 内容为 int_8:<AS, LS, BrightnessAcqIns>[event], 其交互对象 LS 属于因果领域, 现象类型为事件, 根据表 1 和表 2 的规

则, 在 TA 模型中添加一个位置节点, 建立从初始位置到该位置的迁移, 并在迁移上添加同步信号 BrightnessAcqIns!. 接着情景图中是一个选择结构, 根据情景图选择结构对应的 TA 模型结构, 先标记当前位置节点为选择结构开始位置命名为 start, 建立满足约束条件的一个分支, 添加一个位置节点命名为 AutoSendOn, 建立从 start 到 AutoSendOn 的迁移, 并将选择结构中的约束条件 Brightness<100 作为迁移的守卫条件. 同时建立另外一个为不满足约束条件的分支, 添加一个位置节点命名为 L1, 建立从 start 到 L1 的迁移, 将选择结构中约束条件取反即 Brightness>=100 作为迁移上的守卫条件.



(a) 自动存储的情景图 (b) 自动存储的TA模型

图 10 自动存储的情景图及生成的 TA 模型

对于满足约束条件的分支, 情景图中下一个行为交互 int$_9$ 内容为 int$_9$:<AS, LC, OpenInstructionStorageIns>[event], 交互对象 LC 是词法领域, 现象类型为事件, 根据表 1 和表 2 规则, 添加一个位置节点, 由于该行为交互下一个便是分支结构结束, 因此将该位置节点命名为 end, 并建立从 AutoSendOn 到 end 的迁移, 在迁移上添加更新条件 OpenInstructionStorageIns=1. 对于不满足约束条件的分支, 情景图中下一个又是一个选择结构, 因此再分别建立满足该约束条件和不满足该约束条件的分支, 对于满足约束条件的分支, 添加一个位置节点, 命名为 AutoSendOff, 建立从 L1 到 AutoSendOff 的迁移, 迁移上的添加守卫条件 Brightness>200, 对于不满足约束条件的分支, 由于后续没有其他行为交互, 因此直接建立从 L1 到 end 的迁移, 并添加守卫条件 Brightness<=200. 接着对于情景图中最后一个行为交互 int$_{10}$ 内容为 int$_{10}$:<AS, LC, CloseInstructionStorageIns>[event], 交互对象为词法领域, 现象类型为事件, 因此建立从 AutoSendOff 到 end 的迁移, 并添加更新条件 CloseInstructionStorageIns=1. 最后情景图中还有一个时间约束, 表示延时 5ms, 因此在 end 位置上添加不变式 Aut<=5, 并在进入 end 位置的所有迁移上添加更新条件 Aut=0, 以及在离开 end 位置的迁移上添加守卫条件 Aut>=5. 由于 end 位置为选择结构的所有分支的汇合位置节点, 为了使该控制器 TA 可以重复调度执行, 因此建立从 end 位置到初始位置 AS 的迁移. 生成的自动存储 TA 模型如图 10(b) 所示.

最后, 对从领域设备知识库中获取的设备构件的 TA 与根据软件构件和其对应情景图生成的软件构件 TA 进行模型声明, 以及变量和信道声明. 在 UPPAAL 工具的模型声明界面添加模型声明 system Initialization, AutomaticSave, CommandedSave, LUControlOutput, LightUnit, LightSensor, PulseGenerator, AirCondition. 变量声明便是对生成 TA 过程中添加的时钟、整型变量、信道进行声明.

4.4 一致性验证与不一致定位

根据文献 [34], 需求一致性被定义为在转换系统中存在满足所有时间约束的任意长的时间路径. 时间路径是一个状态序列, 以及系统从一个状态到下一个状态的时间. 在我们的方法中, 我们使用时间自动机来表示状态、状态迁移和时间, 用时间计算树逻辑 (TCTL) 来表达要验证的性质. 我们认为需求的一致性在嵌入式系统中表现为

设备的调度冲突, 这是由于设备的物理特性决定的. 它指的是在任意时刻, 都不能同时向设备发送不同的信号, 使其处于不同的状态. “不能同时向设备发送不同的信号”这条性质在表达时, 由于信号是来自于其他不同模型的, 比如模型 a 和模型 b, 假设模型 a 在状态 m 和模型 b 在状态 n 时, 分别向该设备发送不同的信号. 我们将这句话表达为 “A[] not 模型 a. 状态 m and 模型 b. 状态 n”, 其含义是在所有路径所有状态中模型 a 的状态 m 和模型 b 的状态 n 不能同时达到. 如果验证结果满足则表明这些状态不可能同时发生, 则需求无调度冲突. 如果验证结果不满足, 说明有可能在某个时刻, 两个状态会同时出现, 导致向同一设备发送不同信号, 从而产生设备调度冲突. 在生成具体性质时, 需要查找可能向同一设备发送不同信号的模型, 并利用这些信号触发之前的状态进行生成, 可能会生成多个性质. 由于不同验证子系统中涉及的设备和调度设备模型不同, 可运行的性质可能也会有所不同.

将针对每个验证子系统生成的一致性性质输入 UPPAAL 平台, 对该子系统模型进行验证, 即可得到该部分的一致性验证结果. 若每个验证子系统验证结果都是“一致”, 最终的结果才为“一致”, 否则为“不一致”. 对于不满足一致性性质的可执行验证子系统模型, 我们希望定位出在该可执行验证系统模型中具体不满足的原因. 我们假设领域设备库中的设备模型都是正确的, 那么出错的原因只能是在生成的软件构件 TA 模型. 对于不满足一致性的可执行验证系统模型, 根据其依赖关系, 从依赖的第 1 个需求对应的可执行验证系统模型开始进行一致性验证, 随后依次添加依赖关系上的下一个需求对应的可执行验证系统模型, 据此定位出不满足一致性的具体某个需求对应的软件构件 TA 模型.

5 案例与评估

5.1 案例研究

本节使用航空领域机载侦查控制系统进行案例研究. 该系统由显控设备 (display control device, DCD)、综合控制单元 (integrate control unit, ICU)、采集接收单元 (collecte receive unit, CRU)、信号处理单元 (signal process unit, SPU) 等组成. 系统能够实现加电控制、初始化、数据加载、信号处理 (信号分选、信号识别)、频谱监测、温度报告、超温保护等功能, 共 19 个需求. 主要功能包括: 操作员通过显控设备发送指令给综合控制单元以及各分设备, 包括加电、断电或者其他功能执行命令; 显示各分设备实时状态到显控设备上; 采集接收单元依据系统指令实时采集信号, 信号处理单元对采集的信号进行周期处理并传给综合控制单元.

步骤 1: 依赖关系识别. 首先识别数据依赖关系. 分析机载侦查控制系统 19 个子需求, 通过输入输出关系推导出子需求间存在的数据依赖关系. 例如, 温度报告 (TemperatureReport) 需求从综合控制单元传感器 (ICUSensor)、干扰处理单元传感器 (JPUSensor) 等设备采集温度信息 (ICUTemperature,…, JPUTemperature), 并将数据保存到温度数据 (TemperatureData) 词法领域中. 超温保护需求 (OverTempProtect) 从温度数据词法领域中取出各设备的温度信息, 根据具体的温度数值判断是否需要对设备进行开启或关闭操作. 因此, 温度报告需求是温度数据的生产者, 超温保护需求为温度数据的消费者, 存在数据依赖<OverTempProtect, {ICUTemperature,…, JPUTemperature}, TemperatureReport>. 类似地, 对所有子需求进行数据依赖识别, 可获得 11 组数据依赖关系.

接着, 识别控制依赖关系. 例如, 信号电流控制需求 (WorkingCurrentControl) 能够驱使综合控制单元 (ICU) 进入开启 (ICUOn) 或关闭 (ICUOff) 状态. 初始化需求能够驱使综合控制单元由开启状态进入初始化 (ICUInit) 状态, 再由初始化状态进入到工作 (ICUWork) 状态. 根据领域设备知识库综合控制单元的设备 TA 可知, 由初始状态 (ICUOff) 到达初始化状态的所有路径中均包含开启状态, 同时由 TA 初始状态到达工作状态的所有路径也都含有开启状态, 符合控制依赖识别方法中的情形 3, 因此, 识别出控制依赖<Init, {ICU.ICUOn, ICU.ICUInit, ICU.ICUWork}, WorkingCurrentControl>, 如图 11 所示. 类似地对所有需求进行控制依赖识别, 最终可获得 6 组控制依赖关系.

步骤 2: 验证子系统架构生成. 对于机载侦查控制系统共有 19 个需求, 以每个需求为关注点生成 19 个验证子系统架构. 这里以超温保护子系统架构为例, 说明得到其架构的过程. 根据超温保护需求与其他需求之间的数据依赖关系 (图 11) 可知, 超温保护需求数据依赖于温度报告需求, 这些需求生成超温保护子系统中的软件构件, 并将

这些需求涉及的相关设备转换成设备构件. 最后得到的超温保护子系统架构包括 2 个软件构件和 8 个设备构件, 如图 12 所示. 类似地, 可以得到其他的 18 个验证子系统架构, 每个验证子系统包含的软件构件和设备构件数量如表 3 所示, 其中需求组中的 1、2 等为需求标号.

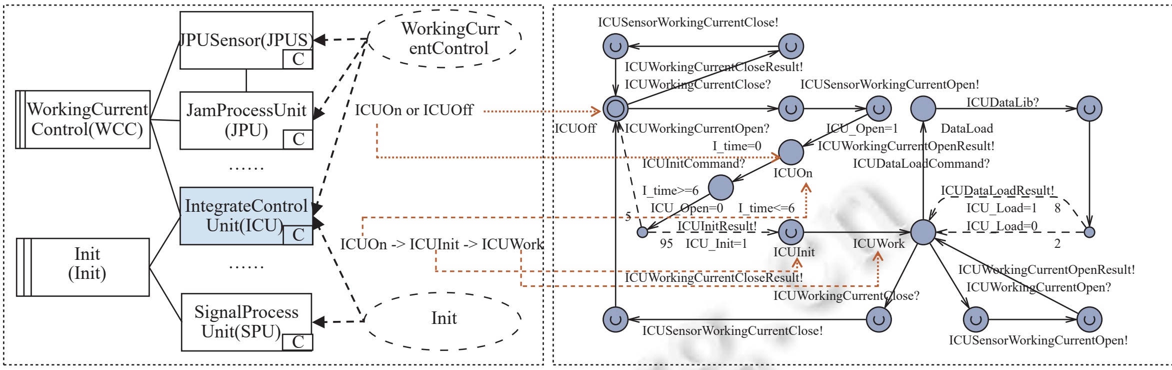


图 11 初始化与信号电流控制需求间的控制依赖示意图

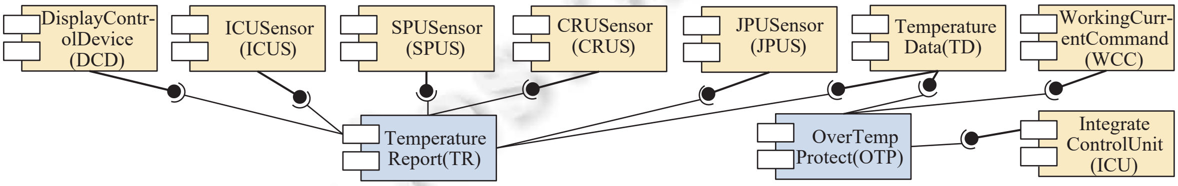


图 12 超温保护验证子系统架构

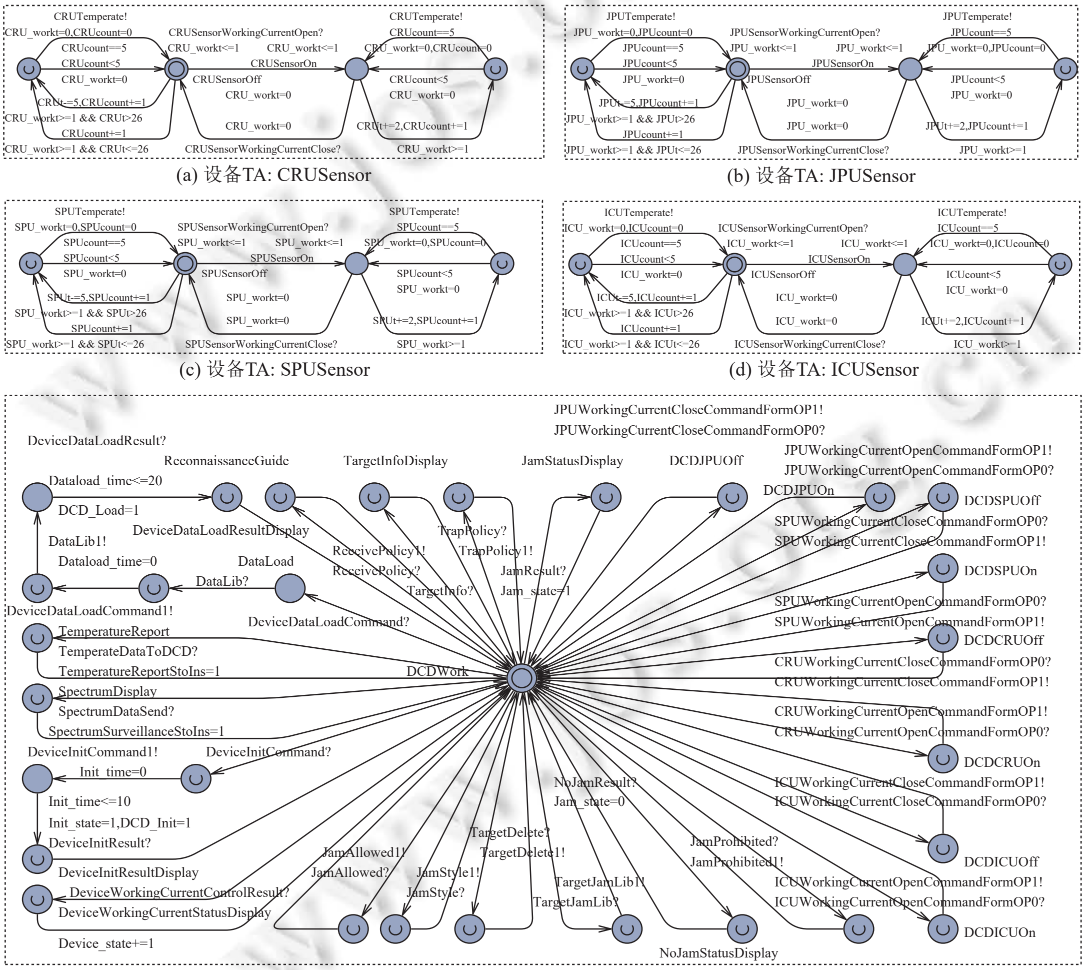
表 3 机载侦查系统验证子系统架构组成

验证子系统	对应需求组	架构组成	验证子系统	对应需求组	架构组成
综合控制单元加电控制 (ICUPowerOpenControl)	{1}	软件构件: 1个 设备构件: 2个 连接件: 2个	数据加载 (DataLoad)	{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}	软件构件: 11个 设备构件: 19个 连接件: 44个
综合控制单元断电控制 (ICUPowerCloseControl)	{2}	软件构件: 1个 设备构件: 2个 连接件: 2个	信号分选 (SignalSort)	{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12}	软件构件: 11个 设备构件: 20个 连接件: 43个
信号处理单元加电控制 (SPUPowerOpenControl)	{3}	软件构件: 1个 设备构件: 2个 连接件: 2个	频谱监测 (SpectrumSurveillance)	{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 13}	软件构件: 11个 设备构件: 19个 连接件: 42个
信号处理单元断电控制 (SPUPowerCloseControl)	{4}	软件构件: 1个 设备构件: 2个 连接件: 2个	目标上报 (TargetReport)	{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 14, 15}	软件构件: 1个 设备构件: 2个 连接件: 2个
采集接收单元加电控制 (CRUPowerOpenControl)	{5}	软件构件: 1个 设备构件: 2个 连接件: 2个	信号识别 (SignalIdentify)	{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15}	软件构件: 13个 设备构件: 22个 连接件: 51个
采集接收单元断电控制 (CRUPowerCloseControl)	{6}	软件构件: 1个 设备构件: 2个 连接件: 2个	侦查引导 (ReconnaissanceGuide)	{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 16}	软件构件: 11个 设备构件: 18个 连接件: 42个
干扰处理单元加电控制 (JPUPowerOpenControl)	{7}	软件构件: 1个 设备构件: 2个 连接件: 2个	干扰处理 (JamProcess)	{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 17}	软件构件: 11个 设备构件: 20个 连接件: 43个

表 3　机载侦查系统验证子系统架构组成 (续)

验证子系统	对应需求组	架构组成	验证子系统	对应需求组	架构组成
干扰处理单元断电控制 (JPUPowerCloseControl)	{8}	软件构件: 1个 设备构件: 2个 连接件: 2个	温度上报 (TemperatureReport)	{18}	软件构件: 1个 设备构件: 6个 连接件: 6个
信号电流控制 (WorkingCurrentControl)	{1, 2, 3, 4, 5, 6, 7, 8, 9}	软件构件: 9个 设备构件: 18个 连接件: 34个	超温保护 (OverTempProtect)	{18, 19}	软件构件: 2个 设备构件: 8个 连接件: 9个
初始化 (Initialization)	{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}	软件构件: 10个, 设备构件: 18个, 连接件: 39个			

步骤 3: 可执行验证子系统模型生成. 根据 19 个验证子系统架构, 从领域设备知识库中获取设备 TA. 根据软件构件以及输入的情景图生成软件构件 TA, 将设备 TA 与软件构件 TA 组合成 NTA 生成可执行验证系统模型集合, 共包括设备 TA 模型 12 个, 软件构件 TA 模型 19 个. 这里以超温保护可执行验证子系统模型为例进行说明. 根据图 12, 首先从领域设备知识库中获取设备构件对应的设备 TA. 设备构件中有 6 个属于因果领域可以直接获取对应 TA 模型如图 13(a)–(f) 所示, 两个属于词法领域没有 TA 模型. 接下来根据超温保护子系统架构中的两个软件构件对应的情景图生成软件控制器 TA, 如图 13(g) 和 (h) 所示.



(a) 设备TA: CRUSensor

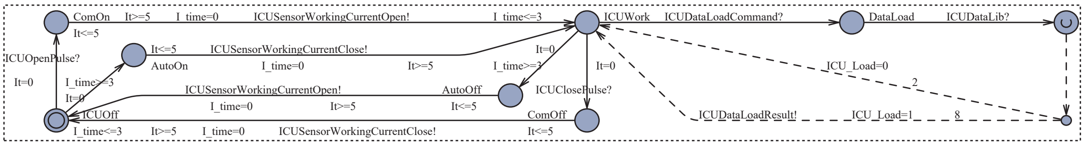
(b) 设备TA: JPUSensor

(c) 设备TA: SPUSensor

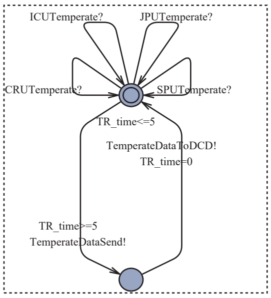
(d) 设备TA: ICUSensor

(e) 设备TA: DisplayControlDevice

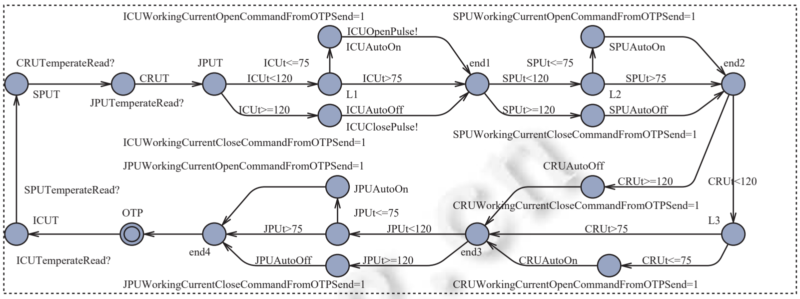
图 13　验证子系统超温保护 NTA



(f) 设备TA: IntegrateControlUnit



(g) 软件构件TA: TemperatureReport



(h) 软件构件TA: OverTempProtect

图 13 验证子系统超温保护 NTA (续)

接下来对超温保护可执行验证子系统模型进行模型声明、变量和信道声明以及变量初始化. 使用 system 关键字进行模型声明, 使用 chan 关键字声明信道, 使用 int 对整型变量进行声明. 最终包含 7 个模型声明, 26 个信道声明, 5 个时钟变量, 8 个整型变量. 具体声明情况如表 4 所示.

表 4 验证子系统超温保护可执行验证系统声明情况

声明	声明变量
模型声明	system OverTempProtect, TemperatureReport, JPUSensor, CRUSensor, ICUSensor, SPUSensor, DisplayControlDevice
信道声明	chan JPUTemperate, JPUSensorWorkingCurrentOpen, JPUSensorWorkingCurrentClose, CRUTemperate, CRUSensorWorkingCurrentOpen, CRUSensorWorkingCurrentClose, ICUTemperate, ICUSensorWorkingCurrentOpen, ICUSensorWorkingCurrentClose, SPUTemperate, SPUSensorWorkingCurrentOpen, SPUSensorWorkingCurrentClose, ICUTemperateRead, ICUWorkingCurrentOpenCommandFromOTPSend, TemperateDataToDCD, ICUWorkingCurrentCloseCommandFromOTPSend, CRUTemperateRead, TemperateDataSend, CRUUWorkingCurrentOpenCommandFromOTPSend, CRUWorkingCurrentCloseCommandFromOTPSend, SPUTemperateRead, SPUWorkingCurrentOpenCommandFromOTPSend, JPUTemperateRead, SPUWorkingCurrentCloseCommandFromOTPSend, JPUWorkingCurrentOpenCommandFromOTPSend, JPUWorkingCurrentCloseCommandFromOTPSend
变量声明	clock TR_time, ICU_workt, CRU_workt, SPU_workt, JPU_workt; int ICUt, CRUt, SPUt, JPUt, ICUcount, CRUcount, SPUcount, JPUcount

步骤 4: 一致性验证与不一致定位. 对于生成的一组可执行验证系统模型, 使用 UPPAAL 工具进行一致性验证. 使用 TCTL 撰写一致性验证性质. 针对嵌入式系统中可能存在的设备调度冲突, 使用的性质模板为“A[] not 模型 a. 状态 m and 模型 b. 状态 n”对每个设备生成相应的性质描述, 放入 UPPAAL 工具进行验证. 本案例有该类待验证性质 10 条.

这里以超温保护可执行验证子系统模型为例, 说明一致性验证过程. 验证系统是否存在设备调度冲突, 其中涉及一条一致性性质“A[] not OverTempProtect.ICUAutoOff and not IntegrateControlUnit.AutoOn”, 该性质验证是否存在超温保护在温度过高时关闭 ICU 和 ICU 定时自动开关之间是否存在调度冲突, 验证结果为“不满足”. 通过不一致性定位, 由于其他依赖需求没有问题, 则推出 OverTempProtect 存在不一致问题. 其他可执行验证子系统模型的验证结果综合如后文表 5 所示.

5.2 方法评估

方法评估的目的就是要回答如下两个问题: 1) 本方法是否能够减少复杂嵌入式系统需求的验证状态空间?

2) 本方法能够提高验证的效率么? 我们考虑采用对比实验, 对比我们的方法对案例的组合验证效果以及整体验证的效果, 对比两次实验来判断本方法是否能够减少验证系统的状态空间以及是否能够提高验证效率.

表 5　机载侦查子系统调度冲突一致性验证结果综合

验证子系统	验证结果
综合控制单元加电控制 (ICUPowerOpenControl)	不涉及设备, 无相关验证
综合控制单元断电控制 (ICUPowerCloseControl)	不涉及设备, 无相关验证
信号处理单元加电控制 (SPUPowerOpenControl)	不涉及设备, 无相关验证
信号处理单元断电控制 (SPUPowerCloseControl)	不涉及设备, 无相关验证
采集接收单元加电控制 (CRUPowerOpenControl)	不涉及设备, 无相关验证
采集接收单元断电控制 (CRUPowerCloseControl)	不涉及设备, 无相关验证
干扰处理单元加电控制 (JPUPowerOpenControl)	不涉及设备, 无相关验证
干扰处理单元断电控制 (JPUPowerCloseControl)	不涉及设备, 无相关验证
信号电流控制 (WorkingCurrentControl)	生成8条性质, 6条不满足, 发现设备ICU、CRU、SPU和JPU存在设备调度冲突, WorkingCurrentControl存在不一致
初始化 (Initialization)	生成1条性质, 不满足, 发现设备LandingGear和Transmitters存在设备调度冲突, WorkingCurrentControl存在不一致
数据加载 (DataLoad)	不涉及设备, 无相关验证
信号分选 (SignalSort)	不涉及设备, 无相关验证
频谱监测 (SpectrumSurveillance)	不涉及设备, 无相关验证
目标上报 (TargetReport)	不涉及设备, 无相关验证
信号识别 (SignalIdentify)	不涉及设备, 无相关验证
侦查引导 (ReconnaissanceGuide)	不涉及设备, 无相关验证
干扰处理 (JamProcess)	不涉及设备, 无相关验证
温度上报 (TemperatureReport)	不涉及设备, 无相关验证
超温保护 (OverTempProtect)	生成1条性质, 不满足, 发现设备ICU存在设备调度冲突, 通过不一致性得出OverTempProtect存在不一致

5.2.1　实验预备

在本实验中, 我们选择了 5 个不同的案例进行实验, 分别为灯光控制系统、智能家居控制子系统、机载侦查控制系统、轨道交通车载控制子系统和航天领域太阳搜索控制子系统. 其中涉及的需求数量从 2–19 个不等, 设备也涉及 5–28 个不等, 含有不同数量的数据依赖和控制依赖. 实验使用的软件环境为 UPPAAL-4.1.24, 使用硬件设备环境为 Windows 10 系统, 32 GB 运行内存, 1 TB SSD.

5.2.2　实验过程

首先为上述 5 个案例中的每个案例的设备建立设备知识库, 将其中涉及的设备进行 TA 建模. 然后采用本文组合验证方法对需求的一致性进行验证. 根据每个案例的问题图以及建立的领域设备知识库识别每个案例中的控制依赖和数据依赖关系. 接着以每个案例中的子需求为关注点, 生成验证子系统架构, 生成可执行验证系统模型. 最后对各个案例中的验证子系统进行一致性验证, 对一致性验证结果进行综合, 并对不满足一致性的情况进行错误定位.

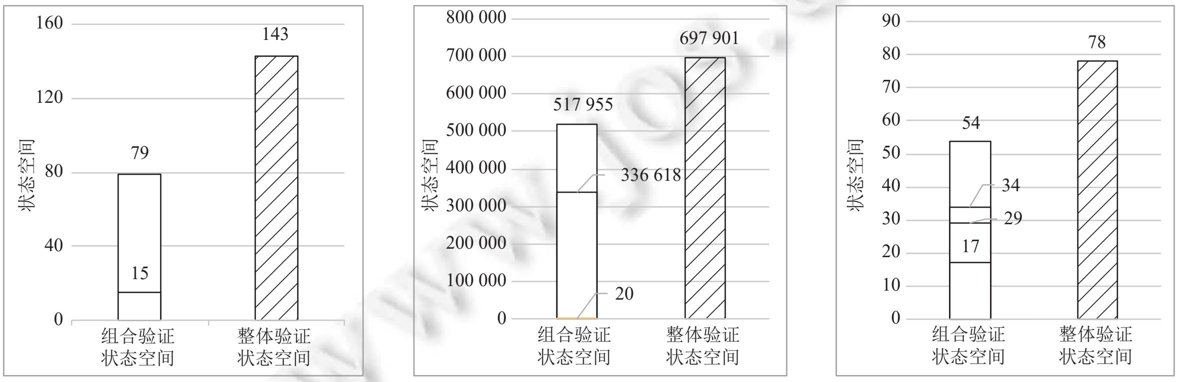
针对该问题的对比实验, 我们使用设备调度冲突一致性验证性质, 根据模板“A[] not 模型 a. 状态 m and 模型 b. 状态 n”进行性质生成, 我们记录其需求依赖关系、验证结果及验证子系统的状态空间和运行时间. 需要特别强调的是, 在验证性质中涉及到的多个设备冲突检测, 使得每个验证性质都对应着不同的状态空间. 接下来, 我们采用整体验证方法, 不考虑依赖关系, 将上述的验证子系统中的软件构件、设备构件合并在一起运行, 进行一致性验证. 在整体验证方法中使用的一致性验证性质与组合验证方法中使用的一致性性质同为一组性质. 记录其验证结果、验证系统的状态空间和运行时间. 其中也涉及多个验证性质的多组验证.

最后, 我们整理了所有的案例和验证数据, 结果如表 6 所示. 对于验证结果中涉及的状态空间, 选取整体遍历状态空间最大的一个性质的验证数据. 这是因为对于不同的验证性质, 它们可以并发执行. 特别地, 对于组合验证, 我们选取同一性质的验证结果中最大状态空间以及最大耗费时间, 这里最大耗费时间是选取对应案例实验中所有需求进行组合验证中某个需求所对应的最大验证时间, 并不是对应案例中组合验证中所有需求对应的验证时间总和, 其他具体结果如图 14 所示. 对于整体验证来说, 验证结果记录整体验证的综合结果, 若所有性质都满足一致性, 则为“满足”, 若存在不满足一致性, 则为“不满足”, 若存在性质超出内存无法验证, 则为“超出内存” (out of memory). 对于超出内存的情况其最大状态空间和最大耗费时间也无法得知, 一般认为验证状态空间过大导致无法得出验证结果, 使用“—”代表. 组合验证列中的依赖关系表明了该案例中存在的数据依赖和控制依赖数量. 组合验证列中的可执行验证子系统指的是根据需求依赖进行组合验证所划分的子系统个数, 而真正执行 NTA 则是与待验证性质相关需要运行的子系统. 子系统验证结果表明各个子系统验证结果情况. 整体模型验证列中可执行验证模型代表该案例中软件构件 TA 和设备构件 TA 数量之和, 即需求数量与设备数量之和. 另外, 考虑到操作系统等因素可能对实验结果产生影响, 我们在实验过程中对每组实验进行了 10 次验证. 我们取这 10 次验证的平均值, 以此来确定最终的探索状态数和所需时间. 其结果如表 6 和图 14 所示. 需要说明的是, 图 14 中的白色柱状展示了不同验证子系统的状态空间. 参与验证的子系统数量会根据其涉及的验证性质有所不同. 如果某个验证子系统并未涉及到特定的验证性质中的设备, 那么这个子系统就不会参与到验证过程中. 因此, 状态空间的数量并不一定会与表 6 中列出的验证子系统数量完全一致.

表 6　组合验证评价结果

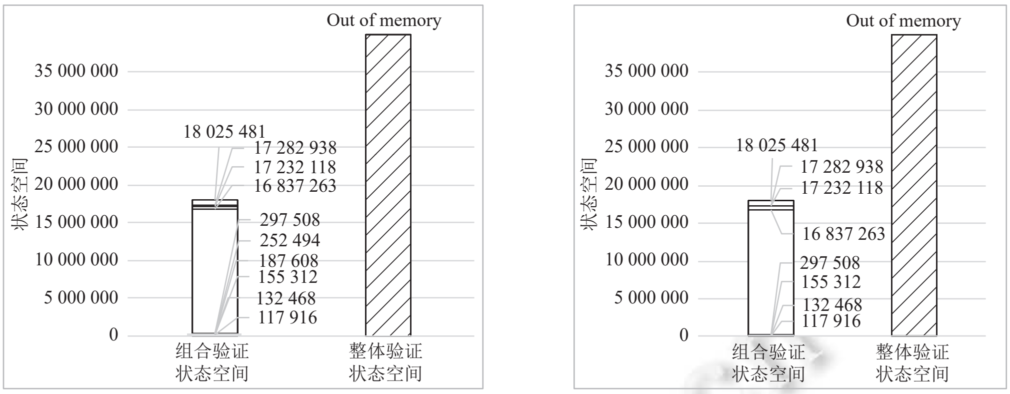
案例			本文方法					整体模型验证				状态空间约减比例 (%)	时间耗费约减比例 (%)
名称	#需求	#设备	#依赖关系	#可执行验证子系统	子系统验证结果	#探索的最大状态	最大耗费时间 (s)	#可执行验证模型	验证结果	#探索的状态	耗费时间 (s)		
灯光控制系统	6	5	数据依赖: 3 控制依赖: 2	6	5满足 1不满足	79	0	11	不满足	143	0	44.76	0
智能家居控制子系统	11	15	数据依赖: 5	11	9满足 2不满足	517955	5.875	26	不满足	697901	7.812	25.78	24.80
轨道交通控制子系统	4	7	数据依赖: 2	4	4满足	54	0	11	满足	78	0	30.77	0
机载侦查子系统	19	12	数据依赖: 11 控制依赖: 6	19	16满足 3不满足	18025481	100.094	31	超出内存	—	—	—	—
太阳搜索控制子系统	19	28	数据依赖: 16 控制依赖: 5	15	10满足 5不满足	19199651	247.236	47	超出内存	—	—	—	—

注: 符号#标注的列数据可以重点关注



(a) 灯光控制系统　(b) 智能家居控制子系统　(c) 轨道交通控制子系统

图 14　各案例组合验证与整体验证状态数对比结果



(d) 机载侦查控制子系统 (e) 太阳搜索控制子系统

图 14 各案例组合验证与整体验证状态数对比结果 (续)

5.2.3 结果分析

从表 6 中可以看出, 我们的组合验证方法表现良好, 针对每组案例都能得到验证结果. 特别是针对规模较大的机载侦查子系统和太阳搜索控制子系统, 原先的整体模型验证受限于 UPPAAL 工具的验证能力, 发生了超出内存无法验证的情况, 而我们的方法可以通过组合子系统约减状态空间来进行验证, 得到验证结果, 这说明我们的方法有着应对复杂系统的能力. 结合图 14(d) 和 (e), 可以看出这两个复杂系统在进行整体验证时探索的状态数高于 10 亿, 而通过我们的组合验证方法, 其探索的子系统的状态数最多在千万级别. 另外, 从可验证系统的角度来看, 如灯光控制系统、智能家居控制系统和轨道交通控制子系统, 本文方法已经将探索的状态空间减少了至少 25.78%. 虽然这些数据只是针对验证子系统而言, 但这两组数据充分说明本文方法能够有效地约减待验证系统的状态空间. 对于超出内存限制的整体模型验证, 如机载侦查子系统和太阳搜索控制子系统, 本文的方法仍然能够进行有效验证, 这显然在减少验证状态空间方面起到了有效作用. 然而, 这并不意味着各验证子系统探索状态的总和一定少于整体验证系统的状态, 例如在图 14(b) 的智能家居系统中, 两个子验证系统探索的状态总和已经超过了整体验证状态.

关于验证效率, 从表 6 中可以看出, 对于相对简单的案例, 如灯光控制系统和轨道交通控制子系统, 其验证状态空间值低于 100 (图 14(a) 和图 14(c)), 在验证过程中并未花费任何时间. 因此对于这类较小的案例, 组合验证方法与整体模型验证方法在时间消耗上并无差别. 然而, 对于稍微复杂一些的系统, 例如智能家居控制子系统 (如图 14(b) 所示), 其状态空间可以达到 10 万级别. 通过比较各子系统验证与整体验证所需的最大时间, 我们发现时间消耗可以降低至少 24.80%. 值得注意的是, 这个时间减少是指每个子系统与整体验证系统相比, 而并非所有子系统验证时间的总和. 若将所有子系统的验证时间加起来, 可能会超过整体系统的验证时间. 这说明对于这类规模还能做整体验证的系统, 本文方法可能并不实用. 对于更加复杂的系统, 如机载侦查子系统和太阳搜索子系统 (图 14(d) 和图 14(e)), 其在组合验证方法中验证空间大小已经达到千万级别. 整体验证方法由于超出内存无法得出验证结果, 而我们的组合验证方法可以进行验证并得出其验证耗费时间, 最大为 247 136 ms. 这充分说明我们的方法在应对复杂系统需求验证时, 能有效提升效率.

5.3 讨 论

本文提出了一种复杂嵌入式系统需求的一致性组合验证方法. 经过案例研究和方法评估, 我们发现它确实可行并且有效, 能够有效约减验证系统的状态空间, 特别是应对复杂嵌入式系统, 达成了预计目标. 但是, 我们也意识到本文方法存在如下限制.

首先, 本文方法建立在问题框架方法基础上, 其需求依赖关系的识别与问题图密切相关. 实际上需求依赖关系并不限制表示方法, 只要是子需求之间, 就有可能出现数据依赖和控制依赖. 其他的表示方法 (如顺序图), 也能挖掘数据依赖和控制依赖, 但可能比问题图要复杂, 因为问题图中对问题领域如词法领域和设备进行了明确的分类.

但这并不代表其他的方法不能用.

其次, 在本文的研究中, 我们选择了时间自动机作为形式化模型, 从验证上来看, 我们的方法受限于时间自动机的验证工具 UPPAAL 的验证能力. 从目前的验证上来看, 探索的状态空间在千万级别可以, 但是对于超过 10 亿的状态空间, 就比较危险了. 对于我们的方法来说, 其解决方案在于需要将子验证系统的规模进行控制, 使得它的验证空间不会过大, 这就对需求的分解有要求, 但如何分解出具有适当验证空间且有一定粒度的需求就成为一个新的研究问题, 但这个不在本文的研究范围内.

关于依赖关系识别的完整性问题, 本文中关注了由于不同需求共享的数据和设备造成的数据依赖和控制依赖. 至于是否还是有其他类型的依赖性, 我们目前并没有发现, 但不排除有的可能性. 在数据和控制依赖中, 我们充分考虑了其可能的影响要素, 以保证依赖关系的完整识别. 其中数据依赖我们做了名唯一假设, 而控制依赖中我们考虑的是需求的前置条件与需求对设备的控制, 这将充分识别需求由于设备的状态变迁带来的隐式关系. 我们认为对于控制依赖的识别已经足够.

另外, 我们的方法并不局限于使用时间自动机. 在实际应用中, 可以根据需要选择不同的形式化模型. 关键在于, 我们需要在第 3 步的验证系统模型中, 根据所选择的形式化模型的特性进行相应的替换和转换. 具体来说, 我们需要能够对软件构件、设备构件和连接构件进行转换. 例如, 如果选择的是状态机模型, 那么就需要能够将软件构件、设备构件和连接构件转换为相应的状态机. 如果选择的是过程代数模型, 那么就需要能够将这些构件转换为相应的过程代数表达式. 这种灵活性使得我们的方法具有很好的可扩展性, 能够适应各种不同的应用场景, 具有广泛的应用前景.

最后, 我们的方法构造的验证子系统是可执行的, 理论上来说它当然可以验证任何可以用 TCTL 描述的性质, 包括其他的安全性质. 但因本文的主题, 我们限制在了一些特定性质的表达上. 需要注意的是, 我们的验证子系统是动态组装的, 它必须依赖于设备的 TA 模型. 在本文中, 假设验证的设备 TA 模型是提前构建好的. 这个假设是可以成立的, 对于每个待验证系统, 其涉及的设备是确定的, 可以由领域专家与形式化专家提前构建在设备库中. 这并不妨碍本文方法的有效性.

6 结束语

本文提出了一种复杂嵌入式系统需求的一致性组合验证方法, 旨在有效约减验证系统的状态空间. 该方法的核心在于充分挖掘需求间的依赖, 通过这些依赖实现对复杂嵌入式系统需求的有效分组, 从而有效构建验证子系统, 并能够初步定位到不一致的需求. 在验证子系统的构建过程中, 我们制定了预期软件的 TA 模型, 并结合物理设备模型进行动态构建, 它充分考虑了嵌入式系统需求的强物理设备性. 这种组合方法有效地减少了每个验证子系统的验证状态空间, 提高验证效率, 并有助于定位不一致性. 我们进一步在航空领域的机载侦查系统中进行了实证研究, 验证了该方法的实用性和有效性, 并通过 5 个案例评估, 证实了验证状态空间的显著减少. 这种组合验证方法为复杂嵌入式系统需求的验证提供了一种实用的解决方案.

在未来工作中, 我们将考虑针对规模控制的需求分解, 使得分解对应的验证子系统的状态空间总是可以验证的, 解除对验证工具的限制. 另外, 一方面考虑将本方法应用于更多实际的项目去, 检测其实际可用性. 另外一方面, 拟针对更多的需求表示类型, 如 UML 顺序图等, 以及更多的性质进行验证, 对方法的应用面进行扩展, 使得方法具有实用性.

References:

[1] Yang MF, Gu B, Duan ZH, Jin Z, Zhan NJ, Dong YW, Tian C, Li G, Dong XG, Li XF. Intelligent program synthesis framework and key scientific problems for embedded software. Chinese Space Science and Technology, 2022, 42(4): 1–7 (in Chinese with English abstract). [doi: 10.16708/j.cnki.1000-758X.2022.0046]

[2] Zowghi D, Gervasi V. On the interplay between consistency, completeness, and correctness in requirements evolution. Information and Software Technology, 2003, 45(14): 993–1009. [doi: 10.1016/S0950-5849(03)00100-9]

[3] Chen XH, Zhong ZW, Jin Z, Zhang M, Li T, Chen X, Zhou TL. Automating consistency verification of safety requirements for railway

interlocking systems. In: Proc. of the 27th Int’l Requirements Engineering Conf. Jeju: IEEE, 2019. 308–318. [doi: 10.1109/RE.2019.00040]

[4] Huang YN, Zhang PJ, Hou XP, Tang T. Modeling and verification method of ZC subsystem in urban rail transit based on hybrid automata. China Railway Science, 2016, 37(2): 114–121 (in Chinese with English abstract). [doi: 10.3969/j.issn.1001-4632.2016.02.16]

[5] Fotso SJT, Frappier M, Laleau R, Mammar A. Modeling the hybrid ERTMS/ETCS level 3 standard using a formal requirements engineering approach. In: Proc. of the 6th Int’l Conf. on Abstract State Machines, Alloy, B, TLA, VDM, and Z. Southampton: Springer, 2018. 262–276. [doi: 10.1007/978-3-319-91271-4_18]

[6] Yang L, Chen YG. Modeling and verification of switch scene of zone controller based on MSC and UPPAAL. Railway Standard Design, 2018, 62(5): 171–174, 179 (in Chinese with English abstract). [doi: 10.13238/j.issn.1004-2954.201704260003]

[7] Chen XH, Liu QQ, Mallet F, Li Q, Cai SB, Jin Z. Formally verifying consistency of sequence diagrams for safety critical systems. Science of Computer Programming, 2022, 216: 102777. [doi: 10.1016/j.scico.2022.102777]

[8] 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 with English abstract). http://www.jos.org.cn/1000-9825/6861.htm [doi: 10.13328/j.cnki.jos.006861]

[9] Vuotto S, Narizzano M, Pulina L, Tacchella A. Poster: Automatic consistency checking of requirements with ReqV. In: Proc. of the 12th IEEE Conf. on Software Testing, Validation and Verification (ICST). Xi’an: IEEE, 2019. 363–366. [doi: 10.1109/ICST.2019.00043]

[10] Wang XB, Li CY, Zhao L. Requirement specification extraction and analysis based on propositional projection temporal logic. Journal of Software: Evolution and Process, 2024, 36(4): e2558. [doi: 10.1002/smr.2558]

[11] Clarke EM, Long DE, McMillan KL. Compositional model checking. In: Proc. of the 4th Annual Symp. on Logic in Computer Science. Pacific Grove: IEEE, 1989. 353−362. [doi: 10.1109/LICS.1989.39190]

[12] Vidal M, Massoni T, Ramalho F. A domain-specific language for verifying software requirement constraints. Science of Computer Programming, 2020, 197: 102509. [doi: 10.1016/j.scico.2020.102509]

[13] Chen XH, Zhang J, Jin Z, Zhang M, Li T, Chen X, Zhou TL. Empowering domain experts with formal methods for consistency verification of safety requirements. IEEE Trans. on Intelligent Transportation Systems, 2023, 24(12): 15146–15157. [doi: 10.1109/TITS.2023.3324022]

[14] Bengtsson J, Larsen K, Larsson F, Pettersson P, Yi W. UPPAAL—A tool suite for automatic verification of real-time systems. In: Proc. of the 1995 Int’l Hybrid Systems III. Berlin: Springer, 1995. 232−243. [doi: 10.1007/BFb0020949]

[15] Massoni T, Mousavi MR. Formal Methods: Foundations and Applications. In: Proc. of the 20th Brazilian Symp. Salvador: Springer, 2018. [doi: 10.1007/978-3-030-03044-5]

[16] Pereira T, Ribeiro Q, Melo M, Magro S, Alencar F, Castro J. Requirements engineering for embedded systems: A systematic literature review. In: Proc. of the 2021 Workshop on Requirements Engineering (WER). 2021. [doi: 10.29327/1298728.24-7]

[17] Yuan ZH, Chen XH, Liu J, Yu YJ, Sun HY, Zhou TL, Jin Z. 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. [doi: 10.1109/TITS.2018.2869633]

[18] Liu XS, Yuan ZH, Chen XH, Chen MS, Liu J, Zhou TL. 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]

[19] Jackson M. Software Requirements and Specifications: A Lexicon of Practice, Principles and Prejudices. New York: Addison-Wesley, 1995.

[20] Jackson M. Problem Frames: Analyzing and Structuring Software Development Problems. Boston: Addison-Wesley, 2001.

[21] Jin Z, Chen XH, Zowghi D. Performing projection in problem frames using scenarios. In: Proc. of the 16th Asia-Pacific Software Engineering Conf. Batu Ferringhi: IEEE, 2009. 249−256. [doi: 10.1109/APSEC.2009.22]

[22] Abrial JR, Butler M, Hallerstede S, Hoang TS, Mehta F, Voisin L. Rodin: An open toolset for modelling and reasoning in Event-B. Int’l Journal on Software Tools for Technology Transfer, 2010, 12(6): 447–466. [doi: 10.1007/s10009-010-0145-y]

[23] Rudolph E, Graubmann P, Grabowski J. Tutorial on message sequence charts. Computer Networks and ISDN Systems, 1996, 28(12): 1629–1641. [doi: 10.1016/0169-7552(95)00122-0]

[24] Caspi P, Pilaud D, Halbwachs N, Plaice JA. LUSTRE: A declarative language for real-time programming. In: Proc. of the 14th ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages. Munich: ACM, 1987. 178−188. [doi: 10.1145/41625.41641]

[25] Younes HLS. Verification and planning for stochastic processes with asynchronous events [Ph.D. Thesis]. Schenley Park: Carnegie Mellon University, 2004.

[26] Ramesh Y, Rao MVP. Statistical model checking for probabilistic temporal epistemic logics. In: Proc. of the 14th Int'l Conf. on Agents and Artificial Intelligence. Online: SciTePress, 2022. 53–63. [doi: 10.5220/0010847900003116]

[27] Zhu WJ, Wu HM, Deng ML. LTL model checking based on binary classification of machine learning. IEEE Access, 2019, 7: 135703–135719. [doi: 10.1109/ACCESS.2019.2942762]

[28] Jhala1 R, McMillan KL. Microarchitecture verification by compositional model checking. In: Proc. of the 13th Int'l Conf. on Computer Aided Verification. Paris: Springer, 2001. 396–410. [doi: 10.1007/3-540-44585-4_40]

[29] McMillan KL. Parameterized verification of the FLASH cache coherence protocol by compositional model checking. In: Proc. of the 11th IFIP WG 10.5 Advanced Research Working Conf. on Correct Hardware Design and Verification Methods. Livingston: Springer, 2001. 179–195. [doi: 10.1007/3-540-44798-9_17]

[30] Namjoshi KS, Trefler RJ. Parameterized compositional model checking. In: Proc. of the 22nd Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Eindhoven: Springer, 2016. 589–606. [doi: 10.1007/978-3-662-49674-9_39]

[31] Zhou S, Wang JB, Xue PP, Wang XY, Kong L. An approach to the state explosion problem: SOPC case study. Electronics, 2023, 12(24): 4987. [doi: 10.3390/electronics12244987]

[32] Shen LX, Mu DJ, Cao G, Qin MY, Zhu JC, Hu W. Accelerating hardware security verification and vulnerability detection through state space reduction. Computers & Security, 2021, 103: 102167. [doi: 10.1016/J.COSE.2020.102167]

[33] Jin Z. Environment Modeling-based Requirements Engineering for Software Intensive Systems. San Francisco: Morgan Kaufmann, 2018. [doi: 10.1016/C2014-0-00030-5]

[34] Li WB, Hayes JH, Truszczyński M. Temporal action language (TAL): A controlled language for consistency checking of natural language temporal requirements: (Preliminary results). In: Proc. of the 4th Int'l Symp. on NASA Formal Methods. Norfolk: Springer, 2012. 162–167. [doi: 10.1007/978-3-642-28891-3_16]

附中文参考文献:

[1] 杨孟飞, 顾斌, 段振华, 金芝, 詹乃军, 董云卫, 田聪, 李戈, 董晓刚, 李晓锋. 嵌入式软件智能合成框架及关键科学问题. 中国空间科学技术, 2022, 42(4): 1–7. [doi: 10.16708/j.cnki.1000-758X.2022.0046]

[4] 黄友能, 张鹏基, 侯晓鹏, 唐涛. 基于混成自动机的城市轨道交通 ZC 子系统建模与验证方法. 中国铁道科学, 2016, 37(2): 114–121. [doi: 10.3969/j.issn.1001-4632.2016.02.16]

[6] 杨璐, 陈永刚. 基于 MSC 与 UPPAAL 的区域控制器切换场景建模与验证. 铁道标准设计, 2018, 62(5): 171–174, 179. [doi: 10.13238/j.issn.1004-2954.201704260003]

[8] 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 [doi: 10.13328/j.cnki.jos.006861]

[18] 刘筱珊, 袁正恒, 陈小红, 陈铭松, 刘静, 周庭梁. 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]

杨晓(2000－), 男, 硕士生, 主要研究领域为嵌入式软件需求验证, 形式化方法.

陈小红(1982－), 女, 博士, 副教授, CCF 专业会员, 主要研究领域为需求工程, 基于知识的软件工程, 形式化方法.

王小齐(1998－), 男, 硕士生, 主要研究领域为嵌入式软件需求分析, 模型驱动开发.

金芝(1962－), 女, 博士, 教授, CCF 会士, 主要研究领域为软件需求工程, 知识工程.