软件学报 ISSN 1000-9825, CODEN RUXUEW
2026,37(9):3508−3520 [doi: 10.13328/j.cnki.jos.007601] [CSTR: 32375.14.jos.007601]

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

# 混成精化逻辑*

孙 欢, 王竟亦, 王文海

(浙江大学 控制科学与工程学院, 浙江 杭州 310027)

通信作者: 王竟亦, E-mail: wangjyee@zju.edu.cn; 王文海, E-mail: zdzzlab@zju.edu.cn

**摘 要**: 混成通信顺序进程 (hybrid communicating sequential processes, HCSP) 是一种广泛应用于混成系统建模的形式化语言. 它结合了由逻辑驱动的状态跳转 (典型于数字计算) 和由微分方程驱动的连续演化 (用于刻画物理过程), 从而统一刻画了离散与连续行为. 这种双重特性使其特别适用于建模通常具有安全关键性的信息物理系统 (cyber-physical system, CPS). 然而, 由于混成系统实现复杂、同步行为错综交织, 其验证在实际中面临显著挑战. 为此, 提出一种用于验证从抽象模型到具体实现之间精化关系的逻辑体系——混成精化逻辑 (hybrid refinement logic, HRL). HRL 通过按照系统的结构进行分解, 并基于精化构造分层的证明过程, 从而提升验证的可复用性和模块化程度. 此外, HRL 还支持并行同步进程与顺序进程之间的精化验证, 进一步降低了证明复杂性, 能有效减轻验证负担.

**关键词**: 混成系统; 混成通信顺序进程; 精化关系; 形式化验证; 定理证明

**中图法分类号**: TP301

中文引用格式: 孙欢, 王竟亦, 王文海. 混成精化逻辑. 软件学报, 2026, 37(9): 3508–3520. http://www.jos.org.cn/1000-9825/7601.htm

英文引用格式: Sun H, Wang JY, Wang WH. Hybrid Refinement Logic. Ruan Jian Xue Bao/Journal of Software, 2026, 37(9): 3508–3520 (in Chinese). http://www.jos.org.cn/1000-9825/7601.htm

## Hybrid Refinement Logic

SUN Huan, WANG Jing-Yi, WANG Wen-Hai

(College of Control Science and Engineering, Zhejiang University, Hangzhou 310027, China)

**Abstract**: Hybrid communicating sequential process (HCSP) is a formal modeling language widely used for hybrid systems. It integrates logic-driven state transitions, typical of digital computation, with continuous evolution governed by differential equations that capture physical dynamics. This dual nature makes HCSP particularly suitable for modeling safety-critical cyber-physical system (CPS). However, practical verification of hybrid systems remains challenging due to their complex implementations and intricate synchronization behaviors. This study introduces hybrid refinement logic (HRL), a formal logic framework for verifying the refinement relation between abstract models and concrete implementations. HRL promotes modular and reusable verification by structuring proofs hierarchically according to the structure of the system. To further reduce verification complexity, HRL also supports refinement reasoning between parallel synchronous processes and sequential processes, significantly alleviating the proof burden.

**Key words**: hybrid system (HS); hybrid communicating sequential process (HCSP); refinement relationship; formal verification; theorem proving

混成系统 (hybrid system, HS) 将离散的计算行为与连续的物理动态相结合, 被广泛用于建模现代信息物理系统 (cyber physical system, CPS)[1], 例如自动驾驶汽车、医疗设备以及工业控制系统等. 这些系统通常部署在对安全性要求极高的环境中, 系统的正确性对于避免潜在的灾难性事故至关重要.

形式化验证通过数学逻辑对系统行为进行建模和推理, 能够从理论上保证系统满足预期的性质, 如安全性、

* 基金项目: 中央高校基本科研业务费专项资金 (2025ZFJH02); 浙江省重点研发计划 (2025C01083)
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: 2025-09-03; 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: 2026-05-14

活性、鲁棒性等, 避免因传统测试未覆盖的边界情况而导致系统故障. 因此, 在确保混成系统按预期运行方面, 形式化验证发挥着不可替代的核心作用. 在混成系统的形式化建模与验证方面, 已有大量研究工作, 主要包括模型检测方法和定理证明[2]. 模型检测方法将混成系统建模为混成自动机[3–7], 并通过计算系统可达状态集来验证系统行为. 模型检测方法具有良好的自动化验证能力, 但是由于混成系统状态空间通常是无限的, 验证过程中容易遭遇状态爆炸问题, 限制了其适用范围. 相较之下, 定理证明方法通过专用的混成建模语言和推理规则对系统进行建模与验证, 尽管自动化程度较低, 通常需要手工编写证明脚本, 但其不受状态爆炸影响, 适用于更复杂系统, 具有更好的可拓展性. 当前具有代表性的定理证明方法主要包括微分动态逻辑 (differential dynamic logic, dL)[8–14]、扩展持续时间逻辑 (extended duration calculus)[15]以及混成霍尔逻辑 (hybrid Hoare logic, HHL)[16–23].

然而, 随着系统规模和复杂度的持续增长, 仅验证系统是否满足某一性质已难以满足模块化系统开发和长期维护的需求. 精化 (refinement)[24]是形式化验证中的一种强大技术, 它确保一个更具体或底层的系统能够保留其抽象设计中的核心性质. 通过将系统划分为多个层级, 并逐层验证它们之间的精化关系, 开发者可以更清晰地组织复杂系统的正确性证明, 从而提高验证的可复用性与可维护性. 精化技术已在操作系统内核[25–27]、文件系统[28,29]以及虚拟机监控器[30]等大规模软件系统的形式化验证中取得了显著成果. 然而, 在混成系统领域, 尽管已有针对离散系统的精化研究广泛展开, 针对具有通信、并发与中断特性的混成系统的精化方法仍相对匮乏. 这一缺陷严重限制了在混成系统环境中对关键的安全性、鲁棒性等属性进行形式化保障的能力.

针对上述问题, 本文提出了一种支持进程间同步通信、并发组合与中断行为的混成精化逻辑 (hybrid refinement logic, HRL), 以增强混成系统精化验证能力. 和现有关于混成系统的精化验证工作相比, 例如微分精化逻辑[31,32], 本文提出的混成精化逻辑支持进程间同步通信、并发组合以及中断等特性, 这些特性都是建模混成系统或信息物理系统的关键组成部分. 本文的贡献主要如下.

(1) 提出针对混成通信顺序进程 (hybrid communicating sequential processes, HCSP) 的混成精化关系和用于验证混成精化关系的混成精化逻辑, 包括针对并发程序的并发混成精化逻辑和针对同步行为的同步混成精化逻辑.

(2) 证明了一个月球着陆控制系统和抽象规约之间的精化关系, 从而证明实际系统运行过程中满足安全不变式.

(3) 提出混成精化逻辑框架, 包括 HCSP 语义、并发与同步精化关系定义以及证明规则, 均已在交互式定理证明器 Isabelle/HOL 中实现. 基于此框架, 进一步在 Isabelle/HOL 中实现了月球着陆器案例的证明. 整个项目累计包含 ≈ 3 000 行定义和证明脚本 确保了本文理论基础的可靠性. 本文所有 Isabelle/HOL 证明脚本均已公开, 代码库地址: https://github.com/SunHuan321/Hybrid_Refinement_Logic.

本文第 1 节介绍关于混成系统验证的相关工作. 第 2 节介绍本文所需的基础知识, 包括 HCSP (本文用于建模混成系统的建模语言) 的语法和语义. 第 3 节介绍本文提出的混成精化逻辑, 包括针对并发程序的并发混成精化逻辑和针对同步行为的同步混成精化逻辑. 第 4 节通过实际案例验证了所提逻辑的有效性. 第 5 节总结本文并展望未来.

## 1 混成系统定理证明和精化逻辑相关工作

定理证明是一种基于逻辑推理的形式化验证方法, 通过构造严格的数学证明, 以确保系统满足给定的性质. 与模型检测不同, 定理证明技术能有效应对无限状态空间、实数变量以及微分方程等复杂结构, 因而特别适用于混成系统与信息物理系统这类包含连续与离散交互行为的场景.

定理证明技术已广泛应用于混成系统的形式化验证. Hooman 等人[33]将霍尔逻辑扩展到时间通信顺序进程, 提出了可用于推理时间和并发行为的构造. 在此基础上, 混成霍尔逻辑 (HHL)[16–22]被发展用于验证混成通信顺序进程 (HCSP)[16–18]. HCSP 是一种融合了通信与并行组合的形式化语言——这两者都是建模混成系统或信息物理系统时不可或缺的基本构件. HHL 随后在交互式定理证明器 Isabelle/HOL[34]中实现, 使得对混成程序的机械化推理成为可能. 进一步地, HHL 还通过符号执行技术提供了自动化或半自动化的证明工具, 例如 HHLPy[35]、HHLPar[36]等, 显著提升了验证效率与工程实用性.

除 HHL 外, 另一个具有代表性的定理证明方法是微分动态逻辑 (dL)[8–14], 它在动态逻辑[8,9]的基础上, 引入了对混成程序的模态扩展. 为应对混成系统中日益复杂的行为, dL 被不断扩展, 衍生出多个适用于不同场景的逻辑变体[31,32,37–39]. 这些逻辑体系由 KeYmaera[13]及其后继者 KeYmaera X[40]提供工具支持, 结合自动化与交互式推理功能, 在复杂系统的验证中展现出良好的实用性.

在混成系统的精化验证方面, Abrial 等人[41]提出了 Hybrid Event-B, 将连续变量及其在时间区间上的连续演化引入 Event-B 框架. Butler 等人[42]进一步定义了一个针对 Hybrid Event-B 的受限精化概念, 采用“两方面推进”的策略: 一方面通过忽略微分方程的具体形式来减少系统抽象演化中的不确定性, 另一方面则向模型中加入额外的离散动作以提升表达能力. Loos 等人[31]提出了一种基于微分动态逻辑的微分精化逻辑 (differential refinement logic, dRL), 该逻辑通过对 KAT 表达式之间的不等式进行推理, 而这些不等式被解释为混成程序之间的精化关系. dRL 不仅支持对微分方程进行抽象, 更重要的是, 这些抽象和精化属性并非作为假设直接使用, 而是可通过微分不变式、微分剪断和微分精化在逻辑系统内部进行形式化证明.

综上所述, 现有混成系统验证方法在理论构建与工具实现方面已取得长足进展, 但在支持具有同步通信和并发结构的系统精化验证方面仍存在一定不足, 亟需新的逻辑框架予以补充与扩展.

## 2 基础知识

本文所提混成精化逻辑主要针对混成通信顺序进程, 下面就相关混成通信顺序进程和其语义予以介绍.

### 2.1 混成通信顺序进程语言

混成通信顺序进程 HCSP[16–18]是在顺序通信进程 (communicating sequential process, CSP) 的基础上扩展而来的一种形式化语言, 专为建模混成系统设计. HCSP 引入常微分方程 (ordinary differential equation, ODE) 来建模系统的物理演化过程与中断行为. 该语言中的进程间交互完全通过通信实现, 并行进程之间不允许共享变量. 本文使用的 HCSP 的语法定义如下:

$$C := skip \mid x := e \mid x := * \mid ch?x \mid ch!e \mid Wait\ d \mid C_1 \sqcup C_2 \mid C_1; C_2 \mid C^* \mid assume\ B \mid \left\langle \overrightarrow{\dot{x}} = \overrightarrow{e} \,\&\, B \right\rangle \mid \left\langle \overrightarrow{\dot{x}} = \overrightarrow{e} \,\&\, B \right\rangle \trianglerighteq_{i \in I} (io_i \to C_i),$$

$$PC := C \mid PC_1 \|_{cs} PC_2,$$

其中, $C$ 和 $C_i$ 表示串行进程; $PC$ 和 $PC_i$ 表示并行进程; $x$ 是进程中的实数变量; $\dot{x}$ 表示变量 $x$ 关于时间的导数, $\overrightarrow{x}$ ($\overrightarrow{e}$) 是变量 (表达式) 向量; $ch$ 为信道名; $io_i$ 表示一个通信事件, 通信事件可以是输入事件 $ch?x$ 或输出事件 $ch!e$; $I$ 是一个非空的标识符集合; $B$ 是布尔表达式; $cs$ 是一组信道名的集合.

空指令 $skip$ 不执行任何操作; 赋值指令 $x := e$ 将表达式 $e$ 的值赋予变量 $x$; 非确定性赋值 $x := *$ 将某个不确定的实数值赋给变量 $x$; 假设指令 $assume\ B$ 的行为如下: 当布尔表达式 $B$ 为真时, 它的执行效果等同于 $skip$ (即不改变状态); 当布尔表达式 $B$ 为假时, 程序不存在后续执行, 即该分支被阻塞, 不存在可达的状态. $C_1 \sqcup C_2$ 表示非确定性选择执行; $C_1; C_2$ 表示顺序执行; $C^*$ 表示非确定性循环, 即执行零次或任意多次 $C$. 常见的程序命令, 如 if-else 分支、while 循环以及一定范围限制下的非确定性赋值等, 可以通过上述语句的组合实现:

$$\textbf{if}\ (B)\{C_1\}\ \textbf{else}\ \{C_2\} \equiv (assume\ B; C_1) \sqcup (assume\ \neg B; C_2),$$

$$\textbf{while}\ (B)\{C\} \equiv (assume\ B; C)^*; assume\ \neg B,$$

$$x := randInt(a,b) \equiv x := *; assume\ a \leqslant x \leqslant b.$$

输入操作 $ch?x$ 表示从信道 $ch$ 接收一个值并赋给变量 $x$; 输出操作 $ch!e$ 表示将表达式 $e$ 的值通过信道 $ch$ 发送出去. 输入或输出操作都可能被阻塞, 直到其通信对象准备就绪. 物理演化过程 $\left\langle \overrightarrow{\dot{x}} = \overrightarrow{e} \,\&\, B \right\rangle$ 表示变量向量 $\overrightarrow{x}$ 根据微分方程 $\overrightarrow{\dot{x}} = \overrightarrow{e}$ 进行演化. 其中 $B$ 为物理演化过程的限制条件, 只要限制条件 $B$ 的取值为真, 物理演化过程一直进行, 一旦限制条件 $B$ 取值为假, 演化过程立即终止. 通信中断进程 $\left\langle \overrightarrow{\dot{x}} = \overrightarrow{e} \,\&\, B \right\rangle \trianglerighteq_{i \in I} (io_i \to C_i)$ 表示系统按照微分方程 $\overrightarrow{\dot{x}} = \overrightarrow{e}$ 执行物理演化过程, 除非在演化过程中被某个通信事件 $io_i$ 中断, 中断后程序继续执行剩余的进程 $C_i$; 如

果没有通信抢占发生, 则物理演化过程会持续直到限制条件 $B$ 取值为假为止. 当通信中断进程只有一个可能的中断事件时, 为简化符号, 本文省略标识符集合 $I$, 将只有一个中断事件的程序简写为 $\left\langle \overrightarrow{x} = \overrightarrow{e} \,\&\, B \right\rangle \trianglerighteq (io \to C)$. 并行组合 $PC_1 \|_{cs} PC_2$ 表示两个进程 $PC_1$ 和 $PC_2$ 并行执行, 在不需要通信的情况下两个进程独立执行, 互不影响. 当它们在共享信道集 $cs$ 上有同步通信事件时, 必须立即进行同步通信. 需要注意的是, 并行进程间的状态空间相互独立, 即不存在共享变量, 进程间的交互完全通过在共享信道上的通信事件完成.

下面这个 HCSP 程序建模了一个典型的周期性控制算法: 左边进程建模了被控对象的行为, 其中 $F(u,s,\dot{s})=0$ 描述了被控对象的物理过程, 其中 $u$ 为控制输入, $s$ 为被控对象的状态. 信道 $p2c$ 用于将被控对象的状态发送给控制器, 而信道 $c2p$ 用于从控制器接收控制信号 $u$. 被控对象按照物理过程的微分方程执行物理演化过程, 并尝试发送被控状态. 当被控状态被控制器接收时, 物理演化过程被打断, 被控对象接收控制器发送的控制信号, 重新执行物理演化过程. 右边部分建模了控制器的行为, 控制器每间隔 $d$ s 接收被控对象的状态, 根据被控对象状态输出相应的控制信号:

$$(\langle F(u,s,\dot{s})=0 \,\&\, True\rangle \trianglerighteq (p2c!s \to c2p?u))^* \|_{\{p2c,c2p\}} (Wait\ d; p2c?x; c2p!f(x))^*.$$

### 2.2 混成通信顺序进程语义

本文使用事件和轨迹描述 HCSP 的行为. 每个事件表示 HCSP 运行过程中的一个可观察步骤. 事件分为两类: 通讯事件和连续事件

- 通讯事件的形式为 $\langle ch|\triangleright v\rangle$, 其中 $|\triangleright$ 可以是 ?、!或 ∘, 分别表示输入、输出或同步, $v$ 是通信过程中传输的实际数值.
- 物理演化事件的形式为 $\langle d,p,rdy\rangle$. 其中 $d$ 是一个正实数, 表示该连续事件的持续时间; $p$ 是一个从时间区间 $[0,d]$ 映射到状态的物理演化函数, 描述了物理变量随时间演化的轨迹, 其中状态是变量到实数的映射; $rdy$ 是在这段时间内等待通信的信道集.

一个轨迹是 HCSP 运行过程中产生的事件序列. 它可以是一个空轨迹 $\epsilon$, 也可以是两个轨迹 $tr_1$ 和 $tr_2$ 的串联, 记作 $tr_1@tr_2$, 并以递归方式定义. 例如, 进程 $ch!3$ 可能产生的轨迹有 $tr_1=\langle ch!3\rangle$ (发送的消息立刻被接收), 或 $tr_1'=\langle 1,Id_{s1},\{ch\}\rangle@\langle ch!3\rangle$ (发送行为被阻塞 1 s 后与其他进程同步, $Id_s$ 表示把区间内所有时间点映射到状态 $s$). 类似的进程 $ch?3$ 可能有 $tr_2=\langle ch?3\rangle$ 或 $tr_2'=\langle 1,Id_{s2},\{ch\}\rangle@\langle ch?3\rangle$.

一个并行 HCSP 进程 $PC_1 \|_{cs} PC_2$ 产生的轨迹 $tr$ 是由进程 $PC_1$ 产生的轨迹 $tr_1$ 和进程 $PC_2$ 产生的轨迹 $tr_2$ 同步后得到, 记作 $tr_1 \|_{cs} tr_2 \Downarrow tr$. 例如在上述例子中 $tr_1 \|_{cs} tr_2 \Downarrow \langle ch\circ 3\rangle$, 而 $tr_1' \|_{cs} tr_2'$ 无法同步 ($rdy$ 不匹配, 如果两个进程能进行同步, 则需立刻同步). HCSP 的完整语义和轨迹的同步规则可以参考本文的 Isabelle/HOL 实现. HCSP 进程 $PC$ 的语义通过一组转移规则进行定义, 记作 $(PC,s)\to(s',tr)$, 表示进程 $PC$ 从初始状态 $s$ 运行到终止状态 $s'$ 停止, 运行过程中产生的轨迹为 $tr$.

## 3 混成精化逻辑

本节介绍本文提出的混成精化逻辑关系与推理规则. 为适应不同类型的精化需求, HRL 提供了两套逻辑分支, 分别适用于并发进程之间的精化 (并发混成精化逻辑) 和并发进程向串行进程的精化 (同步混成精化逻辑), 如图 1 和图 2 所示. 并发混成精化逻辑用于组合验证多个并发 HCSP 进程之间的精化关系. 若目标是验证一个并发 HCSP 实现是否为抽象层面多个子进程的组合精化, 则可分别对各子进程建立精化关系, 并使用组合规则推导整体精化结论. 同步混成精化逻辑则面向另一类常见情形: 若一个并发 HCSP 进程通过通信进行紧密同步, 其行为可在忽略通信事件的情况下抽象为一个串行进程. 此时, HRL 支持将并发结构精化为更简单的串行结构, 从而简化验证任务. 如图 2 所示, 左边的进程和右边的进程先进行一次同步, 之后左边的进程被阻塞 (等待右边进程与其进行同步通信), 而右边的进程进行一系列的计算后再与左边的进程进行同步. 这样的程序行为能被抽象为一个串行 HCSP, 即首先执行第 1 次同步, 然后进行计算, 最后进行第 2 次同步.

与 dRL 相比, 本文提出的并发混成精化逻辑具有如下两个关键区别.

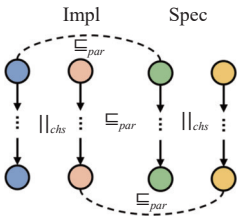


图 1 并发混成精化关系

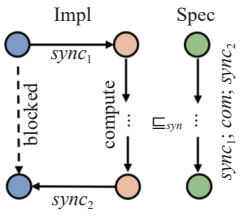


图 2 同步混成精化关系

(1) 状态不需完全一致: dRL 通常假设实现与抽象之间状态在每一步都严格一致, 这在处理包含中间变量或额外控制逻辑的实际系统时限制较大. 相对地, 本文提出的并发混成精化逻辑允许两者之间通过状态模拟关系 (如映射函数或不变式) 进行灵活关联.

(2) 组合性支持: 本文提供了模块化的组合推理规则, 若每个子进程都精化其抽象对应进程, 则整个并发系统也可推出整体精化. 这一组合性原则对于大规模系统的结构化验证尤为关键. 相比之下, dRL 的建模语言不包含并发和通信操作, 不支持针对并发进程的组合推理.

与 dRL 相比, 本文提出的同步混成精化逻辑支持验证并发 HCSP 进程和串行 HCSP 进程的精化关系. 例如第 2 节的控制程序例子, 每隔 $d$ s 被控对象发送受控状态, 接收控制器根据受控状态计算的控制信号, 然后重新执行物理过程. 如果忽略通信行为, 这一控制程序可以等价为一个更为简单的串行混成通信进程:

$$(t := 0; \langle F(u, s, \dot{s}) = 0, \dot{t} = 1 \& t \leqslant d \rangle; u := f(s))^*,$$

即每隔 $d$ s 根据受控状态更新控制信号, 然后重新执行物理过程. 通常验证并发的 HCSP 进程需要大量的手工证明, 而验证 HCSP 的串行进程已经有自动化程度较高的工具支持[35], 因此证明一个并发同步混成通讯顺序进程是一个串行混成顺序进程的精化, 再证明串行混成顺序进程的精化满足安全性质能大大减轻证明负担. dRL 的建模语言不支持同步通信、并发组合以及中断等特性, 因此无法证明并发进程和串行进程的精化关系.

### 3.1 并发混成精化逻辑

并发混成精化逻辑用于推理两个并发 (或串行) HCSP 进程之间的精化关系. 本文定义的并发混成精化关系形式为 $(PC_c, s_c) \sqsubseteq_{par} \alpha (PC_a, s_a)$, 其中 $PC_c$ 和 $s_c$ 为实现层面的 HCSP 进程和初始状态, $PC_a$ 和 $s_a$ 为抽象层面的 HCSP 进程和初始状态, $\alpha$ 为实现层面和抽象层面的状态模拟关系. 直观而言, 如果 $(PC_c, s_c) \sqsubseteq_{par} \alpha (PC_a, s_a)$ 成立, 则每一次实现层面进程的执行, 都可以在抽象进程中找到一个与之对应的执行, 执行过程中实现层面和抽象层面的状态始终保持状态模拟关系 $\alpha$, 且通信事件类型相同, 传输的值满足状态模拟关系, 从而保证实现行为不会偏离抽象规范所允许的范围. 并发混成精化关系定义如下:

**定义 1 (并发混成精化关系).**

$$(PC_c, s_c) \sqsubseteq_{par} \alpha(PC_a, s_a) \triangleq (s_c, s_a) \in \alpha \wedge (\forall s_c'\ tr_c.(PC_c, s_c) \to (s_c', tr_c) \to \exists s_a'\ tr_a.(PC_a, s_a)$$
$$\to (s_a', tr_a) \wedge tr_c \overset{\alpha}{\simeq} tr_a \wedge (s_c', s_a') \in \alpha).$$

并发混成精化关系要求实现层面的 HCSP 进程是抽象层面 HCSP 进程的精化, 当且仅当初始两者状态满足状态模拟关系 $\alpha$, 且实现进程的每一次执行结果都可以被抽象进程模拟, 使得两者的终止状态满足状态模拟关系 $\alpha$, 且实现层面进程产生的轨迹和抽象层面进程产生的轨迹也满足状态模拟关系, 记作 $tr_c \overset{\alpha}{\simeq} tr_a.$, 具体定义如下:

$$tr_c \overset{\alpha}{\simeq} tr_a \triangleq length(tr_c) = length(tr_a) \wedge (\forall i < length(tr_c).\ \textbf{if}\ tr_c[i] = \langle d, p, rdy \rangle$$
$$\textbf{then}\ tr_a[i] = \langle d, p', rdy \rangle \wedge (\forall t \in [0, d].(p(t), p'(t) \in \alpha)\ \textbf{else}\ tr_a[i] \overset{\alpha}{\simeq} tr_c[i]).$$

实现层面进程产生的轨迹和抽象层面进程产生的轨迹满足状态模拟关系, 当且仅当轨迹的长度相同, 且对于实现层面轨迹中的任意事件, 如果该事件是物理演化事件, 那么抽象层面对应的事件也为物理演化事件, 且两者的持续时间和等待通信的信道集合相同, 在时间区间 $[0, d]$ 内, 实现和抽象层面的物理演化函数满足状态模拟关系; 如果该事件是通信事件, 则两者类型相同, 且传输的值满足状态模拟关系.

如果规定实现层面的状态与抽象层面的状态在每一步中都完全相同时, 状态模拟关系为 $Id \equiv \{(s_c, s_a) | s_c = s_a\}$, 则精化关系要求实现层面和抽象层面的初始状态相同, 且实现层面的每一种可能的执行结果都被抽象层面所包含, 此时得到的精化关系和 dRL 类似, 如定理 1 所示.

**定理 1 (Id 状态模拟关系下的精化关系)**.

$$(PC_c, s_c) \sqsubseteq_{par} Id(PC_a, s_a) \Rightarrow s_c = s_a \wedge (\forall s'\ tr.(PC_c, s_c) \rightarrow (s', tr) \Rightarrow (PC_a, s_a) \rightarrow (s', tr)).$$

不变式是形式化验证中的一种基本而核心的思想, 它描述的是在系统运行过程中始终为真的性质. 这类性质通常用于刻画系统的安全边界, 例如: 变量始终处于合法范围 (如温度不超过上限); 某些约束始终成立 (如资源不被同时访问) 等. 在混成系统中, 不变式的作用尤为重要, 因为系统的状态会在离散跳转与连续演化之间交替变化. 相比纯离散系统, 混成系统还需保证不变式在微分方程描述的物理演化过程中的每一个时刻都成立, 这就需要对系统的动力学行为进行更强的约束, 本文对 HCSP 的不变式定义如下.

**定义 2 (并发混成不变式)**.

$$Inv(PC, s, B) \triangleq B(s) \wedge \forall s'\ tr.(PC, s) \rightarrow (s', tr) \Rightarrow B(s') \wedge (\forall i < length(tr).\ \textbf{if}\ tr[i] = \langle d, p, rdy \rangle\ \textbf{then}\ (\forall t \in [0, d].B(p(t)))).$$

一个并发 HCSP 进程 $PC$ 在初始状态 $s$ 下满足不变式 $B$, 记作 $Inv(PC, s, B)$, 当且仅当初始状态满足不变式, 且任一从初始状态出发执行产生的最终状态以及物理演化过程中每一时刻的状态都满足不变式 $B$. 本文证明, 并发混成精化关系保留不变式关系, 即如果抽象模型满足不变式, 且实现是抽象的精化, 则实现也满足相应的不变式:

**定理 2 (并发精化关系保留不变式关系)**.

$$(PC_c, s_c) \sqsubseteq_{\text{par}} \alpha(PC_a, s_a) \wedge (\forall s_c' s_a'.B_a(s_a') \wedge (s_c', s_a') \in \alpha \Rightarrow B_c(s_c')) \Rightarrow Inv(PC_a, s_a, B_a) \Rightarrow Inv(PC_c, s_c, B_c).$$

$$(PC_c, s_c) \sqsubseteq_{par} Id(PC_a, s_a) \Rightarrow Inv(PC_a, s_a, B) \Rightarrow Inv(PC_c, s_c, B).$$

通常情况下, 抽象模型比实现模型在结构和行为上都更加简洁. 抽象系统通常刻画的是系统在高层次下的本质性质, 省略了大量实现细节, 例如中间变量、控制流程、通信机制和冗余状态转换等. 这种简化不仅使得抽象模型的状态空间更小、演化路径更少, 也使得系统行为更加规则化. 因此, 在抽象系统上进行不变式证明往往更为直接和高效.

图 3 所示为并发混成精化关系的证明规则. *Weaken* 规则证明如果精化关系在状态模拟关系 $\alpha$ 下成立, 那么在更宽松的状态模拟条件 $\beta$ 下也成立; *Refl* 规则证明如果状态模拟关系 $\alpha$ 满足自反性, 那么进程自身是对自身的一个精化; *Comp* 规则允许层层精化验证, 从最底层的实现出发不断提高抽象程度, *Comp* 规则中的 $\alpha \circ \beta$ 指两个状态模拟关系的组合 $\alpha \circ \beta = \{(s_c, s_a) | \exists s_m.(s_c, s_m) \in \alpha \wedge (s_m, s_a) \in \beta\}$.

*Nondt* 规则体现了精化关系减少非确定性的思想. 一个赋值语句 $x := e$ (确定性行为) 是 $x := *$ (非确定性行为, 其中*表示任意实数值) 的精化. 这意味着实现 $x := e$ 比抽象的规约 $x := *$ 更加受限和精确, 因此满足精化关系. 通过将具体赋值替换为非确定性赋值, *Nondt* 规则使得验证过程中能够忽略变量的具体取值, 仅保留与证明相关的性质, 从而避免在验证中处理无关的细节, 在混成系统的实际验证中起到重要作用[31,32]. 一个常见的应用场景是, 当赋值总是满足某种约束的条件时, 可以证明一个确定性赋值是带守卫条件的非确定性赋值的精化, 从而在验证中只关注赋值的约束条件本身, 而无需跟踪具体的赋值结果:

$$\frac{B(s(x := e(s)))}{(x := e, s) \sqsubseteq_{par} Id(x ::= *; assume\ B, s)}.$$

*Choicel* 和 *Choicer* 规则用于证明非确定性选择程序的精化关系; *Asm* 规则证明如果实现层的假设 $B_c$ 推出抽象层的假设 $B_a$, 那么该假设语句可以作为精化成立. 也就是说, 实现比抽象更保守或要求更强的条件, 是精化成立的充分条件; *Seq* 规则用于证明顺序执行进程的精化关系, 要证明 $(C_{c1}; C_{c2}, s_c) \sqsubseteq_{par} \alpha(C_{a1}; C_{a2}, s_a)$, 首先需要证明 $(C_{c1}, s_c) \sqsubseteq_{par} \alpha(C_{a1}, s_a)$, 然后证明在所有 $C_{c1}$ 从 $s_c$ 出发的可达状态 $s_c'$ 下 (记作 $s_c' \in R(C_{c1}, s_c)$) 进一步证明 $(C_{c2}, s_c') \sqsubseteq_{par} \alpha(C_{a2}, s_a')$. *Seq* 规则的一个优点是, 它允许在证明 $(C_{c2}, s_c') \sqsubseteq_{par} \alpha(C_{a2}, s_a')$ 时使用 $C_{c1}$ 的行为信息, 即可以利用 $C_{c1}$ 执行后的状态集或不变式. 如果将 *Seq* 规则变为一个全局版本: 直接要求 $(C_{c2}, s_c') \sqsubseteq_{par} \alpha(C_{a2}, s_a')$ 在任意状态下都成立, 那么该规则

逻辑上也是正确的, 但这会丢失所有关于 $C_{c1}$ 的推理信息 (例如它的后置状态、轨迹特性等), 因此该版本在实际证明中很少适用. *Unloop* 规则允许将 $C_c^*$ (即重复执行 $C_c$ 任意次数) 运行期间保持成立的不变式信息传递到精化证明中, 即 $(C_c, s_c) \sqsubseteq_{par} \alpha\,(C_a, s_a)$ 的推理中. 这对于从循环结构中提取已知性质, 并用于后续精化推理非常关键.

$$\frac{\alpha \subseteq \beta \quad (PC_c, s_c) \sqsubseteq_{par} \alpha\ (PC_a, s_a)}{(PC_c, s_c) \sqsubseteq_{par} \beta\ (PC_a, s_a)}[Weaken] \quad \frac{refl(\alpha)}{(PC, s) \sqsubseteq_{par} \alpha\ (PC, s)}[Refl] \quad \frac{(PC_c, s_c) \sqsubseteq_{par} \alpha\ (PC_m, s_m) \quad (PC_m, s_m) \sqsubseteq_{par} \beta\ (PC_a, s_a)}{(PC_c, s_c) \sqsubseteq_{par} \alpha \circ \beta\ (PC_a, s_a)}[Comp]$$

$$\frac{(s_c, s_a) \in \alpha \quad (s_c(x := e(s_c),\ s_a(y := v)) \in \alpha}{(x := e, s_c) \sqsubseteq_{par} \alpha\ (y := *, s_a)}[Nondt] \quad \frac{(C_{c1}, s_c) \sqsubseteq_{par} \alpha\ (C_a, s_a) \quad (C_{c2}, s_c) \sqsubseteq_{par} \alpha\ (C_a, s_a)}{(C_{c1} \sqcup C_{c2}, s_c) \sqsubseteq_{par} \alpha\ (C_a, s_a)}[Choicel]$$

$$\frac{((C_c, s_c) \sqsubseteq_{par} \alpha\ (C_{a1}, s_a)) \vee ((C_c, s_c) \sqsubseteq_{par} \alpha\ (C_{a2}, s_a))}{(C_c, s_c) \sqsubseteq_{par} \alpha\ (C_{a1} \sqcup C_{a2}, s_a)}[Choicer] \quad \frac{(C_{c1}, s_c) \sqsubseteq_{par} \alpha\ (C_{a1}, s_a) \quad \forall\, s_c'\, s_a'.\ s_c' \in R(C_{c1}, s_c) \wedge (s_c', s_a') \in \alpha \Rightarrow (C_{c2}, s_c') \sqsubseteq_{par} \alpha\ (C_{a2}, s_a')}{(C_{c1}; C_{c2}, s_c) \sqsubseteq_{par} \alpha\ (C_{a1}; C_{a2}, s_a)}[Seq]$$

$$\frac{(s_c, s_a) \in \alpha \quad B_c(s_c) \Rightarrow B_a(s_a)}{(assume\ B_c, s_c) \sqsubseteq_{par} \alpha\ (assume\ B_a, s_a)}[Asm] \quad \frac{\forall\, s_c'\, s_a'.\ s_c' \in R(C_c^*, s_c) \wedge (s_c', s_a') \in \alpha \Rightarrow (C_c, s_c') \sqsubseteq_{par} \alpha\ (C_a, s_a')}{(C_c^*, s_c) \sqsubseteq_{par} \alpha\ (C_a^*, s_a)}[Unloop]$$

$$\frac{Inv(\langle \vec{x} = \vec{e}\ \&\ B_1 \rangle, s,\ B_2)}{\left(\langle \vec{x} = \vec{e}\ \&\ B_1 \rangle,\ s\right) \sqsubseteq_{par} Id\ \left(\langle \vec{x} = \vec{e}\ \&\ B_1 \wedge B_2 \rangle,\ s\right)}[DC] \quad \frac{(PC_{c1}, s_{c1}) \sqsubseteq_{par} \alpha\ (PC_{a1}, s_{a1}) \quad (PC_{c2}, s_{c2}) \sqsubseteq_{par} \beta\ (PC_{a2}, s_{a2})}{(PC_{c1} \|_{cs} PC_{c2}, s_{c1} \uplus s_{c2}) \sqsubseteq_{par} \alpha \uplus \beta\ (PC_{a1} \|_{cs} PC_{a2},\ s_{a1} \uplus s_{a2})}[Par]$$

图 3 并发混成精化规则

*DC* 规则提供了精化处理微分方程的方法. 该规则表明, 如果一个微分方程在其物理演化过程中始终满足不变式 $B_2$, 那么该微分方程是一个限制条件加入不变式 $B_2$ 的微分方程的精化. 该规则与微分动态逻辑中的微分剪断类似: 微分剪断技术通过引入额外的不变条件来限制系统的演化范围, 从而简化证明. 类似地, *DC* 规则在精化逻辑中的作用是将隐含的不变量显式化, 使得验证能够在一个更小的搜索空间内进行, 避免在后续证明中重复推导该不变式, 从而有效降低了证明的复杂度. *DC* 规则的一个常见的使用场景是提取一个时间驱动的具体实现所满足的性质[31,32]. 例如, 对于一个无人驾驶系统, 实现往往通过控制车辆的加、减速时间使车辆的速度保持在一定的范围. *DC* 规则可以将隐含的速度的不变量显示化, 使得抽象不需要关注实现细节 (即加、减速时间), 而只需关注速度所需要满足的性质.

以上规则针对串行进程精化关系验证, 可以将不同规则进行组合得到更加复杂的验证规则, 例如下述规则是一个实用的用于验证串行控制程序的规则:

$$\frac{\begin{array}{c}\forall ss'.B_2(s) \Rightarrow s' \in R(C_1, s) \Rightarrow B_2(s') \\ \forall s.B_2(s) \Rightarrow Inv\left(\left\langle \overrightarrow{x} = \overrightarrow{e} \& B_1 \right\rangle, s, B_2\right) B_2(s)\end{array}}{\left(\left(C_1; \left\langle \overrightarrow{x} = \overrightarrow{e} \& B_1 \right\rangle; C_2\right)^*, s\right) \sqsubseteq_{par} Id\left(\left(C_1; \left\langle \overrightarrow{x} = \overrightarrow{e} \& B_1 \wedge B_2 \right\rangle; C_2\right)^*, s\right)}.$$

规则针对的是一个典型控制程序: 循环执行初始化, 物理演化过程和控制算法. 该条规则可以用于证明控制程序在执行过程中始终满足不变式 $B_2$. 利用图 3 中的推理规则证明这条规则成立. 可以首先利用 *Unloop* 规则将控制程序循环的精化关系验证分解为单个循环节的精化关系验证:

$$\forall s.B_2(s) \Rightarrow \left(C_1; \left\langle \overrightarrow{x} = \overrightarrow{e} \& B_1 \right\rangle; C_2, s\right) \sqsubseteq_{par} Id\left(C_1; \left\langle \overrightarrow{x} = \overrightarrow{e} \& B_1 \wedge B_2 \right\rangle; C_2, s\right).$$

接着使用 *Seq* 规则分别对循环节中的 3 个程序 (即子目标) 进行精化关系验证: (1) $\forall s.(C_1, s) \sqsubseteq_{par} Id(C_1, s)$; (2) $\forall s'.s' \in R(C_1, s) \Rightarrow \left(\left\langle \overrightarrow{x} = \overrightarrow{e} \& B_1 \right\rangle, s'\right) \sqsubseteq_{par} Id\left(\left\langle \overrightarrow{x} = \overrightarrow{e} \& B_1 \wedge B_2 \right\rangle, s'\right)$; (3) $\forall s'.(C_2, s') \sqsubseteq_{par} Id(C_2, s')$. 其中子目标 (1) 和子目标 (3) 可以利用 *Refl* 规则证明, 子目标 (2) 可以使用 *DC* 规则证明.

最后介绍本文针对并行进程的组合规则. *Par* 规则是并发组合的精化规则, 该规则表明如果两个子进程分别满足精化关系: $(PC_{c1}, s_{c1}) \sqsubseteq_{par} \alpha\,(PC_{a1}, s_{a1})$ $(PC_{c2}, s_{c2}) \sqsubseteq_{par} \beta(PC_{a2}, s_{a2})$, 那么它们的并发组合也满足精化关系, 对应的状态模拟关系为 $\alpha \uplus \beta$. 本文定义 $s_1 \uplus s_2$ 为两个状态 $s_1$ 和 $s_2$ 的合并 (注意 HCSP 不允许共享资源, 所以状态 $s_1$ 和 $s_2$ 是

严格独立的), $\alpha \uplus \beta$ 为两个状态模拟关系的合并, 具体定义如下:

$$\alpha \uplus \beta \equiv \{(s_1 \uplus s_2, s_1' \uplus s_2') \mid (s_1, s_1') \in \alpha \wedge (s_2, s_2') \in \beta\}.$$

### 3.2 同步混成精化逻辑

本文定义的同步混成精化关系形式为 $(PC_c, s_c) \sqsubseteq_{syn} \alpha (C_a, s_a)$, 其中 $PC_c$ 和 $s_c$ 为实现层面的并行 HCSP 进程和初始状态, $C_a$ 和 $s_a$ 为抽象层面的串行 HCSP 进程和初始状态, $\alpha$ 为实现层面和抽象层面的状态模拟关系. 直观而言, 如果 $(PC_c, s_c) \sqsubseteq_{syn} \alpha (C_a, s_a)$, 则每一次实现层面进程的执行, 都可以在抽象进程中找到一个与之对应的执行, 执行过程中实现层面和抽象层面的状态保持状态模拟关系 $\alpha$, 从而保证在忽略通信事件的前提下, 实现行为不会偏离抽象规范所允许的范围. 同步混成精化关系定义如下.

**定义 3 (同步混成精化关系).**

$$\begin{aligned}(PC_c, s_c) \sqsubseteq_{syn} \alpha (C_a, s_a) \triangleq (s_c, s_a) \in \alpha \wedge \big(\forall s_c'\ tr_c.(PC_c, s_c) \to (s_c', tr_c) \Rightarrow \exists s_a' tr_a.(C_a, s_a)\\ \to (s_a', tr_a) \wedge filter\_io(tr_c) \overset{\alpha}{\sim} filter\_io(tr_a) \wedge (s_c', s_a') \in \alpha\big).\end{aligned}$$

实现层面的 HCSP 进程是抽象层面 HCSP 进程的精化当且仅当初始状态满足状态模拟关系 $\alpha$, 且对于实现进程的每一执行结果都能被设计进程模拟, 使得终止状态满足状态模拟关系 $\alpha$, 且实现进程产生的轨迹和抽象进程产生的轨迹满足状态模拟关系. 同步混成精化逻辑和并发混成精化逻辑的区别有两点: (1) 同步混成精化逻辑要求抽象层面的 HCSP 进程为串行进程. (2) 同步混成精化逻辑只关心系统状态而不关心通信事件, 因此同步混成精化逻辑只关心去除通信事件后的轨迹 $filter\_io(tr)$. 此处 $filter\_io(tr)$ 只保留轨迹 $tr$ 中所有的非通信事件, 并且保持这些非通信事件的先后顺序和原轨迹相同, 形式化的定义可以参考本文的代码仓库.

与并发混成精化关系类似, 同步混成精化关系也保持不变式关系如下.

**定理 3 (同步精化关系保留不变式关系).**

$$(PC_c, s_c) \sqsubseteq_{syn} \alpha (C_a, s_a) \wedge (\forall s_c'\ s_a'.B_a(s_a') \wedge (s_c', s_a') \in \alpha \Rightarrow B_c(s_c')) \Rightarrow Inv(C_a, s_a, B_a) \Rightarrow Inv(PC_c, s_c, B_c).$$

本文希望证明类似第 2.1 节的并发控制程序与串行控制程序之间的精化关系. 因此, 本文定义控制程序状态模拟关系 $\alpha_{\text{ctr}}$ 为: $\alpha_{\text{ctr}} \equiv \{(s_p \uplus s_c, s_p)\}$, 其中 $s_p$ 为被控对象所在进程状态, $s_c$ 为控制器所在进程状态, 本文希望实现层面被控对象所在进程状态与抽象层面控制器进程状态相同, 图 4 是本文提出的同步精化关系验证规则.

$$\frac{(PC_{c1}, s_{c1}) \sqsubseteq_{par} Id\ (PC_{c1}', s_{c1})\ \ (PC_{c2}, s_{c2}) \sqsubseteq_{par} Id\ (PC_{c2}', s_{c2})\quad (C_a',\ s_a) \sqsubseteq_{par} Id\ (C_a,\ s_a)\ \ (PC_{c1}' \|_{cs} PC_{c2}', s_{c1} \uplus s_{c2}) \sqsubseteq_{syn} \alpha\ (C_a',\ s_a)}{(PC_{c1} \|_{cs} PC_{c2}, s_{c1} \uplus s_{c2}) \sqsubseteq_{syn} \alpha\ (C_a,\ s_a)}\ [Cons]$$

$$\frac{(C_1 \|_{cs} C_2, s_p \uplus s_c(v := e(s_p))) \sqsubseteq_{syn} \alpha_{ctr}\ (C_a, s_p)}{(ch!\, e; C_1 \|_{cs} ch?\, v; C_2, s_p \uplus s_c) \sqsubseteq_{syn} \alpha_{ctr}\ (C_a, s_p)}\ [Comm1] \quad \frac{(C_1 \|_{cs} C_2, s_p(v := e(s_c)) \uplus s_c)) \sqsubseteq_{syn} \alpha_{ctr}\ (C_a, s_p(v := e(s_c)))}{(ch?\, v; C_1 \|_{cs} ch!\, e; C_2, s_p \uplus s_c) \sqsubseteq_{syn} \alpha_{ctr}\ (v := e(s_c); C_a, s_p)}\ [Comm2]$$

$$\frac{\forall s_p'.\ (ch!\, e_1; C_1 \|_{cs} ch?\, v; C_2, s_p' \uplus s_c) \sqsubseteq_{syn} \alpha_{ctr}\ (C_a, s_p')}{\big(\langle \vec{x} = \vec{e}\ \&\ True \rangle \trianglerighteq (ch!\, e_1 \to C_1) \|_{cs} Wait\ d; ch?\, v; C_2, s_p \uplus s_c\big) \sqsubseteq_{syn} \alpha_{ctr}\ \big(\langle \vec{x} = \vec{e}, t = 1\ \&\ t \le d \rangle; C_a, s_p\big)}\ [Int]$$

$$\frac{(s_c, s_a) \in \alpha\quad sync\ (C_1, cs, C_2)\quad \forall s_c\ s_a.\ (s_c, s_a) \in \alpha \Rightarrow (C_1 \|_{cs} C_2, s_c) \sqsubseteq_{syn} \alpha\ (C_a, s_a)}{(C_1^* \|_{cs} C_2^*, s_c) \sqsubseteq_{syn} \alpha\ (C_a^*, s_a)}\ [Rep\_Sync]$$

图 4 同步混成精化规则

*Cons* 规则首先证明抽象层面的子进程 $PC_{c1}$ 和 $PC_{c2}$ 是 $PC_{c1}'$ 和 $PC_{c2}'$ 的精化, 这一步保证所有 $PC_{c1} \|_{cs} PC_{c2}$ 执行所产生的结果被 $PC_{c1}' \|_{cs} PC_{c2}'$ 所包含 (见定理 1), 再证明抽象层面进程 $C_a'$ 是 $C_a$ 的精化, 这一步保证 $C_a'$ 执行所产生的结果被 $C_a$ 包含, 最后证明精化后的实现进程是抽象进程的精化. *Cons* 规则允许利用图 3 的同步混成精化规则先对待验证的进程进行简化, 再验证简化后的进程, 从而减少验证负担. *Comm*1 和 *Comm*2 规则反映了同步消息接收的精化验证机制. 当一个进程等待发送消息而另一个进程等待接收消息, 两者会先进行消息同步, 即把发送消息的值赋予接收消息的变量, 然后继续执行剩余的进程. *Int* 规则用于处理中断通讯的精化验证机制. 当一个系统正在执行连续变化, 且只可能会被通信打断, 而另一个进程等待 $d$ s 后接收消息. 这相当于连续变化执行 $d$ s 后将发送消息的值赋予接收消息的变量, 然后继续执行剩余的进程.

在一般情形下, $(C_1\|_{cs}C_2,s_c)\sqsubseteq_{syn}\alpha(C_a,s_a)$ 并不能得出 $(C_1^*\|_{cs}C_2^*,s_c)\sqsubseteq_{syn}\alpha(C_a^*,s_a)$. 这是由于在并发系统中, 循环节的执行并不一定是同步的. 然而对于某些特殊情形, 例如周期性控制程序, 这样的精化关系是成立的. *Rep_Sync* 规则描述了这种精化关系成立的条件, 其中关键是证明循环节是同步执行的, 即 $sync(C_1,chs,C_2)$, 图 5 提供了一系列同步执行的验证规则.

$$\frac{ch\in chs}{sync(ch!\,e,chs,ch?\,v)}[Sync_{Comm}]\quad \frac{sync(C_2,chs,C_1)}{sync(C_1,chs,C_2)}[Sync_{Sym}]\quad \frac{sync(C_1,chs,C_2)\ \ sync(C_1',chs,C_2')}{sync(C_1;C_1',chs,C_2;C_2')}[Sync_{Seq}]$$

$$\frac{sync(C_1,chs,C_2)\ \ dense(C_1')\ \ dense(C_2')}{sync(C_1;C_1',chs,C_2;C_2')}[Sync_{Dense}]\quad \frac{ch\in chs\ \ \ sync(C_1,chs,C_2)}{sync(\langle\vec{x}=\vec{e}\,\&\,True\rangle\trianglerighteq(ch!\,e_1\to C_1),chs,Wait\ d;ch?\,v;C_2)}[Sync_{Int}]$$

图 5　同步执行规则

$Sync_{Comm}$ 和 $Sync_{Int}$ 规则表示接发消息和中断在相应信道是同步执行的; $Sync_{Sym}$ 表示同步执行关系满足对称性; $Sync_{Seq}$ 表示同步执行关系在顺序组合下是封闭的; $Sync_{Dense}$ 表示一对同步执行进程与一对纯计算程序的顺序组合仍是同步执行的, 其中 $dense(C)$ 表示 $C$ 是纯计算程序, 即执行不产生轨迹.

需要指出的是, 本节提出的同步混成精化规则 (图 4 和图 5) 主要针对一类在信息物理系统中常见的交互模式, 即周期性的采样-计算-驱动的控制循环. 因此, 精化规则主要覆盖了在这种模式下典型的通信和中断场景. 虽然这些规则并未涵盖 HCSP 所有可能的进程间通信组合, 但它们为验证大量实际的控制系统提供了一个高效且结构化的方法. 将此框架扩展以支持更广泛、更复杂的同步拓扑, 是未来一个重要的研究方向.

类似的, 可以利用图 4 和图 5 的同步混成精化规则导出更为复杂的验证规则. 为展示同步精化逻辑的通用性, 本节构建了一个针对一般性周期性控制程序的验证模版: 被控对象 *Plant* 执行物理演化过程, 发送受控状态并接收控制信号, 而控制程序 *Ctrl* 每隔 $d$ s 接收受控状态, 然后根据受控状态输出相应的控制信号, 被控对象和控制信号并发执行. 首先通过 HCSP 定义被控对象 *Plant* 和控制器 *Ctrl*:

$$Plant\triangleq\langle\vec{x}=\vec{e}\,\&True\rangle\trianglerighteq(ch_{p2c}!e_1\to ch_{p2c}!e_2;\ldots ch_{p2c}!e_m;ch_{c2p}?v_1;\ldots;ch_{c2p}?v_n),$$

$$Ctrl\triangleq Wait\ d;ch_{p2c}?w_1;\ldots ch_{p2c}?w_m;ch_{c2p}!f_1(w_1,\ldots w_m);\ldots;ch_{c2p}!f_n(w_1,\ldots,w_m).$$

如果忽略通信事件, 并发程序的行为相当于一个串行程序, 执行物理演化过程, 每隔 $d$ 秒根据受控状态更新控制信号. 注意被控对象的受控状态与接收变量互相独立, 即 $e_i$ 的值不受 $v_j$ 的影响. 因此, 串行的抽象程序 *Abs* 定义如下:

$$Abs\triangleq t:=0;\langle\vec{x}=\vec{e},t=1\&t\leqslant d\rangle;v_1:=f_1(e_1,\ldots,e_m);\ldots;v_n:=f_n(e_1,\ldots e_m).$$

本研究首先根据 *Comm*1、*Comm*2 和 *Int* 规则证明单个循环节的精化关系:

$$(Plant\|_{\{ch_{p2c},ch_{c2p}\}}Ctrl,s_p\uplus s_c)\sqsubseteq_{syn}\alpha_{ctr}(Abs,s_p).$$

根据图 5 的规则容易证明被控对象 *Plant* 和控制器 *Ctrl* 同步执行, 即 $sync(Plant,\{ch_{p2c},ch_{c2p}\},Ctrl)$, 由 *Rep_Sync* 规则可以得到针对一般性周期性控制程序的验证模板:

$$(Plant^*\|_{\{ch_{p2c},ch_{c2p}\}}Ctrl^*,s_p\uplus s_c)\sqsubseteq_{syn}\alpha_{ctr}(Abs^*,s_p).$$

## 4　验证示例

本节介绍一个改编自文献 [43] 的月球着陆器的示例, 展示如何使用本文提出的混成精化逻辑对一个实际的信息物理系统进行验证. 在该示例中, 着陆器在下降引导控制系统的作用下向月球表面着陆, 控制系统的目标是维持着陆器下降速度稳定.

月球着陆器的物理过程由以下微分方程定义:

$$\dot{v}=\frac{F_c}{m}-g_M,\quad \dot{m}=-\frac{F_c}{I},$$

其中, $v$ 表示下落速度, $m$ 是着陆器的质量, $F_c$ 是施加在着陆器上的推力, $g_M$ 和 $I$ 分别是月球重力加速度常数和质

量损耗速率. 推力 $F_c$ 受以下微分方程约束:

$$F_c = m\times\left(g_M - c_1(v-v_s) - c_2\left(\frac{F_c}{m} - g_M\right)\right),$$

其中, $v_s$ 是着陆器需要维持的目标速度, 在本示例中设定为−1.5 m/s. 由于原始微分方程为非多项式形式, 本文将 $\frac{F_c}{m}$ 替换为一个新变量 $w$. 经过替换后, 本文定义月球着陆控制系统的被控对象和控制程序:

$$Plant \triangleq \left\langle \dot{v} = w - 3.732, \dot{w} = \frac{w^2}{2500} \& True \right\rangle \trianglerighteq (p2c!v \to p2c!w; c2p?w),$$

$$Ctrl \triangleq wait\ T; p2c?v; p2c?w; c2p!f(v,w).$$

被控对象着陆器, 它执行物理过程并将在某个时刻被通信中断, 依次向控制器发送当前的被控变量 $v$ 和 $w$, 然后从控制器接收新的控制信号 $w$. 而控制器每隔 $T = 0.128$ s 执行一次周期性控制, 首先接收被控变量 $v$ 和 $w$, 然后根据控制算法 $f(v,w) \triangleq -(w-3.732)\times 0.01 + 3.732 - (v+1.5)\times 0.6$ 计算出新的控制信号 $w$, 并通过信道发送至被控对象着陆器.

整个月球着陆器控制过程被建模为被控对象着陆器和控制器的并发组合:

$$Plant^* \|_{\{p2c,c2p\}} Ctrl^*.$$

本节首先针对该控制过程的循环节进行精化验证. 由于本示例的验证目标是验证着陆器下降速度始终保持在一个安全范围内 (−1.45 至−1.55 m/s), 因此本示例只关心被控对象的状态而不关心通信事件. 由于实现层面的 HCSP 是一个并发通信进程, 因此首先定义抽象层面的串行 HCSP 进程:

$$Abs \triangleq t := 0; \left\langle \dot{v} = w - 3.732, \dot{w} = \frac{w^2}{2500}, \dot{t} = 1 \& t \leqslant T \right\rangle; w := f(v,w).$$

接着利用已经在第 3.2 节证明的针对一般性周期性控制程序的验证模版证明其与一个串行 HCSP 进程的精化关系:

$$(Plant^* \|_{\{p2c,c2p\}} Ctrl^*, s_p \uplus s_c) \sqsubseteq_{syn} \alpha_{ctr}(Abs^*, s_p).$$

抽象层面的进程旨在能够清晰反映验证目标, 即着陆器下降速度始终保持在一个安全范围内 (−1.45 至−1.55 m/s), 因此, 系统的不变式 $Abs_{inv}$ 定义如下:

$$Abs_{inv} \triangleq t := 0; \left\langle \dot{v} = w - 3.732, \dot{w} = \frac{w^2}{2500}, \dot{t} = 1 \& t \leqslant T \wedge -1.55 \leqslant v \leqslant -1.45 \right\rangle; w := f(v,w).$$

接着, 应用第 3.1 节的串行控制程序验证规则, 可以证明上述串行 HCSP 是一个更强的 HCSP 进程的精化. 该证明基于以下假设: 被控对象初始状态 $s_p$ 满足不变式:

$$(Abs^*, s_p) \sqsubseteq_{par} Id(Abs^*_{inv}, s_p).$$

最后, 通过 *Cons* 规则把两次精化的规则结合起来, 得到实现和最高抽象层面的精化关系:

$$(Plant^* \|_{\{p2c,c2p\}} Ctrl^*, s_p \uplus s_c) \sqsubseteq_{syn} \alpha_{ctr}(Abs^*_{inv}, s_p).$$

该抽象 HCSP 进程清晰地表明, 在整个控制过程中, 被控对象着陆器下降速度 $v$ 始终保持在一个安全范围内 (−1.45 至−1.55 m/s). 根据定理 2 和定理 3, 同步混成精化关系和并发混成精化关系都保持不变式关系. 由此可证, 在实现层面的控制系统下, 被控对象着陆器下降速度 $v$ 始终保持在一个安全范围内, 从而证明了控制系统的安全性.

对于示例中的 HCSP 进程, 也可以用 HHL 等方法进行验证不变式. HHL 逻辑首先验证被控对象进程满足相应的霍尔逻辑表达式, 此表达式的后置条件需要反映两种可能的情形: (1) 物理演化过程被通信进程立即打断; (2) 物理演化过程演化一段时间后被通信进程打断, 再验证控制器进程满足相应的霍尔逻辑表达式, 其后置条件需要反映控制器经过 $T$ s 的等待后接收被控状态并输出控制信号. HHL 提供了组合规则将两个验证后的并行进程的霍尔三元组进行合并, 在这一步中, 需要证明被控对象后置条件只有物理演化过程演化 $T$ s 后的情形能与控制器的后置条件进行合并, 从而得到整个并发进程的后置条件. 而本文提出的 HRL 逻辑不需要考虑多余的情形, 直接将对

应的同步行为精化为等待 $T$ s 后更新控制器状态, 使得证明过程更为简洁可读. 此外, 相比于 HHL, 本文提出的 HRL 支持分层精化证明, 通过将系统划分为多个抽象层次, 从最顶层的规范性模型逐步过渡到最底层的实现细节, 使得系统的复杂性得到有效的控制, 降低了验证的难度, 同时增强了系统的可维护性和可拓展性, 提高了验证结果的复用性和可组合性.

## 5 总 结

本文针对混成系统的形式化验证问题, 提出了一种支持进程间通信、中断与并发组合的混成精化逻辑体系. 该逻辑以 HCSP 为建模语言基础, 通过引入状态模拟关系与轨迹一致性条件, 系统性地定义了实现进程精化抽象规范的语义判定准则. 为适应不同系统结构和验证需求, 本文进一步提出了两类逻辑分支: 并发混成精化逻辑和同步混成精化逻辑. 前者支持在实现与抽象均为并发结构的场景中, 通过模块化组合规则将子进程的局部精化推广为整体系统的精化; 后者则支持将同步通信驱动的并发进程精化为等价的串行进程, 为将复杂系统转化为有自动化验证工具支持的串行模型提供了理论基础. 此外, 混成精化逻辑刻画了净化过程中不变式的保持性, 为安全性和鲁棒性等关键性质在抽象层级间的传递性证明提供了形式化保障. 为了验证所提方法的可行性, 本文以周期控制系统为例, 构建了一个通用的验证模板, 并在此基础上对改编自月球着陆器控制系统的经典案例进行了形式化建模与精化验证. 相关定义与证明规则均已在 Isabelle/HOL 中实现, 累计构建超过 3 000 行的定义与验证脚本, 验证了该方法在定理证明器支持下的可落地性与实用性. 对于未来工作, 混成系统精化验证还有许多值得进一步探索的问题, 尤其是在自动化验证方向上. 例如: 如何设计自动或半自动的证明搜索机制, 以及精化机制是否可以用于混成系统程序的自动合成等.

## References


[1] Wing J. How can we provide people with cyber-physical systems they can bet their lives on. Computing Research News, 2008, 20(1).

[2] Bu L, Xie DB. Formal verification of hybrid system. Ruan Jian Xue Bao/Journal of Software, 2014, 25(2): 219–233 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4535.htm [doi: 10.13328/j.cnki.jos.004535]

[3] Alur R, Courcoubetis C, Henzinger TA, Ho PH. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. Hybrid Systems. Berlin: Springer, 1993. 209–229. [doi: 10.1007/3-540-57318-6_30]

[4] Henzinger TA. The theory of hybrid automata. In: Proc. of the 11th Annual IEEE Symp. on Logic in Computer Science. New Brunswick: IEEE, 1996. 278–292. [doi: 10.1109/LICS.1996.561342]

[5] Lafferriere G, Pappas GJ, Yovine S. Symbolic reachability computation for families of linear vector fields. Journal of Symbolic Computation, 2001, 32(2): 231–253. [doi: 10.1006/jsco.2001.0472]

[6] Lynch N, Segala R, Vaandrager F, Weinberg HB. Hybrid I/O automata. In: Hybrid Systems III: Verification and Control. Berlin: Springer, 1996. 496–510. [doi: 10.1007/BFb0020971]

[7] Manna Z, Pnueli A. Verifying hybrid systems. Hybrid Systems. Berlin: Springer, 1993. 4–35. [doi: 10.1007/3-540-57318-6_22]

[8] Harel D, Kozen D, Tiuryn J. Dynamic Logic. Cambridge: The MIT Press, 2000.

[9] Harel D. First-Order Dynamic Logic. Berlin: Springer, 1979. [doi: 10.1007/3-540-09237-4]

[10] Platzer A, Clarke EM. Computing differential invariants of hybrid systems as fixedpoints. In: Proc. of the 20th Int'l Conf. on Computer Aided Verification. Princeton: Springer, 2008. 176–189. [doi: 10.1007/978-3-540-70545-1_17]

[11] Platzer A. Differential dynamic logic for hybrid systems. Journal of Automated Reasoning, 2008, 41(2): 143–189. [doi: 10.1007/s10817-008-9103-8]

[12] Platzer A. Logical Foundations of Cyber-physical Systems. Cham: Springer, 2018. [doi: 10.1007/978-3-319-63588-0]

[13] Platzer A, Quesel JD. KeYmaera: A hybrid theorem prover for hybrid systems (system description). In: Proc. of the 4th Int'l Joint Conf. on Automated Reasoning. Sydney: Springer, 2008. 171–178. [doi: 10.1007/978-3-540-71070-7_15]

[14] Platzer A. Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics. Berlin: Springer, 2010. [doi: 10.1007/978-3-642-14509-4]

[15] Zhou CC, Ravn AP, Hansen MR. An extended duration calculus for hybrid real-time systems. Hybrid Systems. Berlin: Springer, 1993. 36–59. [doi: 10.1007/3-540-57318-6_23]

[16] Zhan NJ, Wang SL, Zhao HJ. Formal modelling, analysis and verification of hybrid systems. Unifying Theories of Programming and

Formal Engineering Methods. Shanghai: Springer, 2013. 207–281. [doi: 10.1007/978-3-642-39721-9_5]

[17] Zou L, Zhan NJ, Wang SL, Fränzle M, Qin SC. Verifying simulink diagrams via a hybrid Hoare logic prover. In: Proc. of the 2013 Int'l Conf. on Embedded Software. Montreal: IEEE, 2013. 1–10. [doi: 10.1109/EMSOFT.2013.6658587]

[18] Liu J, Lv JD, Quan Z, Zhan NJ, Zhao HJ, Zhou CC, Zou L. A calculus for hybrid CSP. In: Proc. of the 8th Asian Symp. on Programming Languages and Systems. Shanghai: Springer, 2010. 1–15. [doi: 10.1007/978-3-642-17164-2_1]

[19] Guelev DP, Wang SL, Zhan NJ, Zhou CC. Super-dense computation in verification of hybrid CSP processes. In: Proc. of the 10th Int'l Symp. on Formal Aspects of Component Software. Nanchang: Springer, 2013. 13–22. [doi: 10.1007/978-3-319-07602-7_3]

[20] Wang SL, Zhan NJ, Guelev D. An assume/guarantee based compositional calculus for hybrid CSP. In: Proc. of the 9th Annual Conf. on Theory and Applications of Models of Computation. Beijing: Springer, 2012. 72–83. [doi: 10.1007/978-3-642-29952-0_13]

[21] Guo DQ, Lü JD, Wang SL, Tang T, Zhan NJ, Zhou DT, Zou L. Formal analysis and verification of Chinese train control system. Scientia Sinica Informationis, 2015, 45(3): 417–438 (in Chinese with English abstract). [doi: 10.1360/N112014-00017]

[22] Guelev DP, Wang SL, Zhan NJ. Compositional Hoare-style reasoning about hybrid CSP in the duration calculus. In: Proc. of the 3rd Int'l Symp. on Dependable Software Engineering: Theories, Tools, and Applications. Changsha: Springer, 2017. 110–127. [doi: 10.1007/978-3-319-69483-2_7]

[23] Liu T, Wang SL, Zhan NJ. Safety verification of trajectory planning for multiple robots. Ruan Jian Xue Bao/Journal of Software, 2017, 28(5): 1118–1127 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5207.htm [doi: 10.13328/j.cnki.jos.005207]

[24] Leroy X. A formally verified compiler back-end. Journal of Automated Reasoning, 2009, 43(4): 363–446. [doi: 10.1007/s10817-009-9155-4]

[25] Klein G, Elphinstone K, Heiser G, Andronick J, Cock D, Derrin P, Elkaduwe D, Engelhardt K, Kolanski R, Norrish M, Sewell T, Tuch H, Winwood S. seL4: Formal verification of an OS kernel. In: Proc. of the 22nd ACM SIGOPS Symp. on Operating Systems Principles. Big Sky: ACM, 2009. 207–220. [doi: 10.1145/1629575.1629596]

[26] Gu RH, Shao Z, Chen H, Wu XN, Kim J, Sjöberg V, Costanzo D. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In: Proc. of the 12th USENIX Symp. on Operating Systems Design and Implementation. Savannah: USENIX Association, 2016. 653–669.

[27] Xu FW, Fu M, Feng XY, Zhang XR, Zhang H, Li ZH. A practical verification framework for preemptive OS kernels. In: Proc. of the 22nd 28th Int'l Conf. on Computer Aided Verification. Toronto: Springer, 2016. 59–79. [doi: 10.1007/978-3-319-41540-6_4]

[28] Zou M, Ding HR, Du D, Fu M, Gu RH, Chen HB. Using concurrent relational logic with helpers for verifying the AtomFS file system. In: Proc. of the 27th ACM Symp. on Operating Systems Principles. Huntsville: ACM, 2019. 259–274. [doi: 10.1145/3341301.3359644]

[29] Chajed T, Tassarotti J, Kaashoek MF, Zeldovich N. Verifying concurrent, crash-safe systems with Perennial. In: Proc. of the 27th ACM Symp. on Operating Systems Principles. Huntsville: ACM, 2019. 243–258. [doi: 10.1145/3341301.3359632]

[30] Li SW, Li XP, Gu RH, Nieh J, Hui JZ. A secure and formally verified Linux KVM hypervisor. In: Proc. of the 2021 IEEE Symp. on Security and Privacy. San Francisco: IEEE, 2021. 1782–1799. [doi: 10.1109/SP40001.2021.00049]

[31] Loos SM, Platzer A. Differential refinement logic. In Proc. of the 31st Annual ACM/IEEE Symp. on Logic in Computer Science. New York: ACM, 2016. 505–514. [doi: 10.1145/2933575.2934555]

[32] Platzer A. Hybrid dynamical systems logic and its refinements. Science of Computer Programming, 2025, 239: 103179. [doi: 10.1016/j.scico.2024.103179]

[33] Hooman J. Extending Hoare logic to real-time. Formal Aspects of Computing, 1994, 6(S1): 801–825. [doi: 10.1007/bf01213604]

[34] Wang SL, Zhan NJ, Zou L. An improved HHL prover: An interactive theorem prover for hybrid systems. In: Proc. of the 17th Int'l Conf. on Formal Engineering Methods. Paris: Springer, 2015. 382–399. [doi: 10.1007/978-3-319-25423-4_25]

[35] Sheng HH, Bentkamp A, Zhan BH. HHLPy: Practical verification of hybrid systems using Hoare logic. In: Proc. of the 25th Int'l Symp. on Formal Methods. Lübeck: Springer, 2023. 160–178. [doi: 10.1007/978-3-031-27481-7_11]

[36] Jin XY, Zhan BH, Wang SL, Zhan NJ. HHLPar: Automated theorem prover for parallel hybrid communicating sequential processes. arXiv:2407.08936, 2025.

[37] Platzer A. Differential game logic. ACM Trans. on Computational Logic, 2015, 17(1): 1. [doi: 10.1145/2817824]

[38] Rajhans A, Bhave A, Ruchkin I, Krogh BH, Garlan D, Platzer A, Schmerl B. Supporting heterogeneity in cyber-physical systems architectures. IEEE Trans. on Automatic Control, 2014, 59(12): 3178–3193. [doi: 10.1109/TAC.2014.2351672]

[39] Müller A, Mitsch S, Retschitzegger W, Schwinger W, Platzer A. A component-based approach to hybrid systems safety verification. In: Proc. of the 12th Int'l Conf. on Integrated Formal Methods. Reykjavik: Springer, 2016. 441–456. [doi: 10.1007/978-3-319-33693-0_28]

[40] Fulton N, Mitsch S, Quesel JD, Völp M, Platzer A. KeYmaera X: An axiomatic tactical theorem prover for hybrid systems. In: Proc. of the 25th Int'l Conf. on Automated Deduction. Berlin: Springer, 2015. 527–538. [doi: 10.1007/978-3-319-21401-6_36]

[41] Abrial JR, Su W, Zhu HB. Formalizing hybrid systems with event-B. In: Proc. of the 3rd Int'l Conf. on Abstract State Machines, Alloy, B, VDM, and Z. Pisa: Springer, 2012. 178–193. [doi: 10.1007/978-3-642-30885-7_13]

[42] Butler M, Abrial JR, Banach R. Modelling and refining hybrid systems in event-B and Rodin. In: Petre L, Sekerinski E, eds. From Action Systems to Distributed Systems: The Refinement Approach. New York: Chapman and Hall/CRC, 2016. 29–42.

[43] Zhao HJ, Yang MF, Zhan NJ, Gu B, Zou L, Chen Y. Formal verification of a descent guidance control program of a lunar lander. In: Proc. of the 19th Int'l Symp. on Formal Methods. Singapore: Springer, 2014. 733–748. [doi: 10.1007/978-3-319-06410-9_49]

## 附中文参考文献

[2] 卜磊, 解定宝. 混成系统形式化验证. 软件学报, 2014, 25(2): 219–233. http://www.jos.org.cn/1000-9825/4535.htm [doi: 10.13328/j.cnki.jos.004535]

[21] 郭丹青, 吕继东, 王淑灵, 唐涛, 詹乃军, 周达天, 邹亮. 中国高速铁路列控系统的形式化分析与验证. 中国科学: 信息科学, 2015, 45(3): 417–438. [doi: 10.1360/N112014-00017]

[23] 刘涛, 王淑灵, 詹乃军. 多机器人路径规划的安全性验证. 软件学报, 2017, 28(5): 1118–1127. http://www.jos.org.cn/1000-9825/5207.htm [doi: 10.13328/j.cnki.jos.005207]

## 作者简介

**孙欢**, 博士生, 主要研究领域为形式化验证, 信息流安全, 定理证明.

**王竟亦**, 博士, 研究员, 博士生导师, CCF 高级会员, 主要研究领域为形式化验证与安全, 软件工程, 人工智能.

**王文海**, 博士, 教授, 博士生导师, CCF 专业会员, 主要研究领域为工业智能与优化控制, 智能感知与检测, 机器学习.