软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2023,34(8):3467−3484 [doi: 10.13328/j.cnki.jos.006867]

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

基于不可满足核的近似逼近可达性分析*

于忠祺, 张小禹, 李建文

(华东师范大学 软件工程学院, 上海 200063)

通信作者: 李建文, E-mail: jwli@sei.ecnu.edu.cn

摘　要: 近年来, 形式化验证技术受到了越来越多的关注, 它在保障安全关键领域系统的安全性和正确性方面发挥着重要的作用. 模型检测作为形式化验证中自动化程度较高的分支, 具有十分广阔的发展前景. 研究并提出了一种新的模型检测技术, 可以有效地对迁移系统进行模型检测, 包括不安全性检测和证明安全性. 与现有的模型检测算法不同, 提出的基于不可满足核(unsatisfied core, UC)的近似逼近可达性分析(UC-based approximate incremental reachability, UAIR), 主要利用不可满足核来求解一系列的候选安全不变式, 直至生成最终的不变式, 以此来实现安全性证明和不安全性检测(漏洞查找). 在基于 SAT 求解器的符号模型检测中, 使用由可满足性求解器得到的 UC 构造候选安全不变式, 如果迁移系统本身是安全的, 得到的初始不变式只是安全不变式的一个近似. 然后, 在检查安全性的同时, 逐步改进候选安全不变式, 直到找到一个真正的不变式, 证明系统是安全的; 如果系统是不安全的, 该方法最终可以找到一个反例证明系统是不安全的. 作为一种全新的方法, 利用不可满足核进行安全性模型检测, 取得了相当好的效果. 众所周知, 模型检测领域没有绝对最好的方法, 尽管该方法在基准的可解数量上无法超越当前的成熟方法, 例如 IC3, CAR 等, 但是该方法可以解出 3 个其他方法都无法解出的案例, 相信本方法可以作为模型检测工具集很有价值的补充.

关键词: 符号模型检测; 形式化验证; 不可满足核; SAT 求解器; 不变式
中图法分类号: TP301

中文引用格式: 于忠祺, 张小禹, 李建文. 基于不可满足核的近似逼近可达性分析. 软件学报, 2023, 34(8): 3467−3484. http://www.jos.org.cn/1000-9825/6867.htm

英文引用格式: Yu ZQ, Zhang XY, Li JW. UC-based Approximate Incremental Reachability. Ruan Jian Xue Bao/Journal of Software, 2023, 34(8): 3467−3484 (in Chinese). http://www.jos.org.cn/1000-9825/6867.htm

UC-based Approximate Incremental Reachability

YU Zhong-Qi, ZHANG Xiao-Yu, LI Jian-Wen

(Software Engineering Institute, East China Normal University, Shanghai 200063, China)

Abstract: In recent years, formal verification technology has received more and more attention, and it plays an important role in ensuring the safety and correctness of systems in critical areas of safety. As a branch of formal verification with a high degree of automation, model checking has a very broad development prospect. This work studies and proposes a new model checking technique, which can effectively check transition systems, including bug-finding and safety proof. Different from existing model checking algorithms, the proposed method, unsatisfied core (UC)-based approximate incremental reachability (UAIR), mainly utilizes the unsatisfied core to solve a series of candidate safety invariants until the final invariant is generated, so as to realize safety proof and bug-finding. In symbolic model checking based on the SAT solver, the UC obtained by the satisfiability solver is used to construct candidate safety invariant, and if the transition system itself is safe, the initial invariant obtained is only an approximation of the safety invariant. Then, while checking the safety, the

* 基金项目: 国家自然科学基金(62002118, U21B2015); 上海市浦江人才计划(19511103602); 上海市可信工业互联网软件协同创新中心(2021-2025)
本文由“约束求解与定理证明”专题特约编辑蔡少伟研究员、陈振邦教授、王戟研究员、詹博华副研究员、赵永望教授推荐.
收稿时间: 2022-09-04; 修改时间: 2022-10-13; 采用时间: 2022-12-14; jos 在线出版时间: 2022-12-30

candidate safety invariant is gradually improved until a real invariant is found that proves the system is safe; if the system is unsafe, the proposed method can finally find a counterexample to prove the system is unsafe. As a brand new method, UCs are exploited for safety model checking and sound results are achieved. As it is all known, there is no absolute best method in the field of model checking, although the proposed method cannot surpass the current mature methods such as IC3, CAR, etc., the method in this paper can solve 3 cases that other mature methods are unable to solve, it is believed that this method can be a valuable addition to the model checking toolset.

Key words: symbolic model checking; formal verification; unsatisfied core; SAT solver; invariant

许多安全关键领域如航空航天、轨道交通、芯片设计等, 由于涉及人类生命、重大财产安全, 对系统的安全性十分重视. 以芯片为例, 无论是自动驾驶汽车、智能手机, 还是微小的心脏起搏器, 都有芯片的身影, 如果芯片出现问题, 可能会造成严重的后果, 给当事人或相关企业带来极大的生命或财产损失. 为解决此类问题, 从芯片设计到生产为成品进入市场的过程中, 有诸多流程以保证安全芯片的质量, 如仿真(simulation)、验证(verification)、可测性设计(design for test, DFT)等[1]. 形式化验证是一种将数学方法应用在计算机安全领域的科学方法. 简单来讲, 形式化验证利用数学方法将目标系统进行形式化建模, 在此基础上, 用数学推理的方法(定理证明)或者自动化穷举所有行为的方式(模型检测)对目标模型检测其是否符合某种设计者编写的性质. 无论输入是什么, 形式化验证提供了一种确保硬件或软件系统满足某些关键性质的方法. 在某些情形下, 过去20年里开发的自动化方法可以使用最小的用户交互去验证大型复杂系统的性质, 并检测这些系统中的错误. 这些自动化程度非常高的方法可以作为设计验证过程中仿真和测试的辅助手段, 使设计的系统更健壮、更可靠.

使用合适的工具对目标迁移系统进行建模, 并用逻辑(例如线性时态逻辑, linear temporal LogicF, LTL[2])指定系统性质之后, 就能够以自动化方式来形式化地验证这些性质. 作为一种完全自动化的验证技术, 模型检测技术由 Clarke 和 Emerson 首次提出[3]. 该种技术不仅可以证明系统是满足性质的, 还能在系统不满足性质时提供反例, 即一条从初始状态到违反性质状态的路径. 反例可以用于追踪错误的系统行为, 这对诊断系统问题非常有价值. 一般地, 给定一个系统模型 M 和一个性质 P, 模型检测回答了 P 是否满足模型 M 的问题. 模型检测技术的基本原理是: 利用自动化搜索技术(算法)对待检查模型的整个状态空间进行遍历, 查看待检查模型是否符合给定的性质. 模型检测器作为模型检测算法的实现实体, 其作用如图1所示. 它将要验证的性质 P 和待检测的模型 M 作为输入, 若性质成立, 则找到一个不变式证明性质 P 在模型 M 上始终成立; 否则, 提供一个反例作为系统违反性质的证据. 这里, 反例通常是一个输入序列, 通过该序列可以还原系统到达违反性质的路径. 特别的, 若 P 为所谓的安全性质(safe property), 模型检测找到的反例长度就一定是有限的. 本文主要关注的是安全性质的模型检测技术.

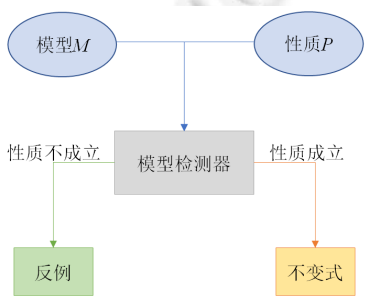


图 1 模型检测框架

作为一种有效的形式化验证技术, 模型检测在硬件设计界中受到越来越多的关注[1,4], 已应用于软硬件开发生命周期的大多数阶段以确保正确性, 如在电子设计自动化(electronic design automation, EDA)领域应用广泛. 除此之外, 模型检测可以用于验证软件需求[5−8]、软件设计模型[9−11], 甚至用于测试和调试[12,13]. 最后, 模型检测还可以用于验证广泛的应用程序, 如 Web、设备驱动程序[14−16]、GUI[17,18]、分布式程序[19−21]、嵌入式

系统[22−24]、数据库[25]和恶意软件[26]. 这些场景表明, 模型检测在软件工程中扮演了重要的角色[27].

虽然模型检测已广泛应用于软件和硬件验证, 但为了能帮助解决更多的工业实例, 继续改进性能仍然是迫切需要. 众所周知的是: 在模型检测领域, 没有一种最佳技术是优于所有其他技术的; 而且不同的算法在不同的基准上表现不同的性能[28]. 尽管在近 30 年前发明了有界模型检测(bounded model checking, BMC)[29], 但其依然被认为是检测性质违反或 bug 的最有效的技术. 同时, 插值模型检测(IMC)[30]和 IC3[31], 或性质有向可达性(PDR)[32], 在证明正确性方面具有更强的能力. 因此, 学术界和工业界通常会维护一个模型检测技术的组合来解决不同的问题.

在本文中, 我们提出一种全新的安全性质模型检测算法 UAIR (UC-based approximate incremental reachability), 它使用可满足性判定得到的不可满足核来构造候选安全不变式, 并在检查不安全性的同时对候选安全不变式进行改进, 直到找到一个真正的不变式证明系统是安全的, 或者找到一个反例来证明系统不安全. 与之前的模型检测算法 IC3/PDR 或是 CAR[33,34]相比, UAIR 的主要特征是不需要维护从起点或终点可达的(向上近似的)状态序列, 而是通过计算一组候选安全不变式(本质上也是真实状态空间的超集), 并将它们组合在一起最终收敛得到一个真实的系统不变式. 我们基于 2015 年[35]和 2017 年[36]硬件模型检测大赛的 749 个工业实例进行了综合实验评估. 限定实验时间 1 h, 实验结果显示: 本方法可以求解 179 个案例, 其中包括 3 个其他几个方法在相同时间内无法求解的案例.

本文第 1 节讨论模型检测技术的相关工作. 第 2 节列出模型检测及其前置知识. 第 3 节介绍 UAIR 的设计并对 UAIR 进行高层次的分析展示. 第 4 节给出实验过程及实验结果. 最后, 第 5 节对本文进行讨论和总结.

1 模型检测相关工作

给定一个迁移系统(例如芯片的设计), 模型检测在状态空间中自动探索以验证给定性质是否在系统上成立. 现有的模型检测算法大多数都基于展开公式, 或利用上近似和下近似并且显式地维护一个或多个状态序列. 目前, 流行的硬件模型检测技术包括有界模型检测(BMC)、插值模型检测(IMC)、IC3/PDR 以及互补近似可达性分析(CAR).

BMC 将搜索归约为一个 SAT 调用序列[37], 每个 SAT 调用对应于某个步骤中的检查. 若其中一个 SAT 调用可满足, 则证明了模型违背给定的性质. BMC 只能用于寻找反例不能用于证明系统的正确性, IMC 在此基础上通过引入 Craig 插值作为一种抽象技术, 从而维护了一个从初始状态可达的向上近似的状态序列, 并通过检查该状态序列是否收敛实现了证明正确性的目标. IC3/PDR[31,32]是近年来主流、完备的模型检测技术,它也像 IMC 一样维护了向上近似的状态序列来证明正确性. 然而与 IMC 相比, IC3/PDR 最大的优势就是不需要对系统进行展开, 从而避免了给底层求解器造成的负担, 也可以获得更好的验证性能.

现今已有相当多的基于 IC3/PDR 的扩展优化工作. 比如, AVY[38]将 IC3/PDR 与 BMC 结合, 代替求插值的方法从而得到一个新的完备算法; QUIP[39]探索更加灵活且高效的约束学习技巧来提高证明的效率; simpleic3[28]给出了一组 IC3/PDR 内部不同参数的最优组合, 使得 IC3/PDR 的性能表现(实验性)最优; Seufert 等人[40]研究了将正向与反向 IC3/PDR 结合的方式, 可以比单一方向的效果要更佳; Dureja 等人[41]提出, 可以通过学习包含内部组合变量(之前都只包含内部状态变量)来提升 IC3/PDR 证明的效率; Seufert 等人[42]还提出, 可以通过限制部分输入变量的方式快速找到反例; 否则, 在此基础上逐步放松对输入变量的约束, 从而得到一个完备的算法.

这 3 种模型检测方法(BMC, IMC, IC3/PDR)已经被证明是高度可扩展的, 并且是今天现代符号模型检测器算法组合的一部分, 例如 ABC[43]和 NuSMV/NuXmv[44].

互补近似可达性(complement approximate reachability, CAR)是一种相对较新的基于 SAT 的安全性模型检测方法, CAR 本质上是一种受 IC3/PDR 启发的可达性分析算法. 与 IC3/PDR 相比, CAR 摒弃了状态序列的单调性质, 从而得到一种更加灵活高效的寻找反例的完备算法. 实验结果表明: CAR 在寻找反例上有比 IC3/PDR 更优异的性能, 在证明正确性上也可以作为 IC3/PDR 的一个补充.

本文所介绍的 UAIR 是一种全新、完备的模型检测算法, 与之前的 IMC, IC3/PDR 和 CAR 相比, UAIR 不需要维护一个向上近似的状态序列来帮助做证明, 而是采用近似逼近的方式, 通过求解一系列的候选安全不变式直至生成最终的不变式, 实现对系统正确性的证明或是反例查找.

2 基础知识

本文所提方法主要基于布尔迁移系统、SAT 求解器和符号模型检测方法, 下面就相关概念和基本知识予以介绍.

2.1 布尔迁移系统

基于 SAT 的模型检测技术将布尔迁移系统视为模型. 布尔迁移系统 Sys 定义为三元组(V,I,T), 其中, V 是布尔变量的集合, 系统中的每个状态 s 属于 2^V, 即给 V 中变量真值赋值的集合; I 是初始状态的集合; 如果把 V 的拷贝标记为 V'以表示后继变量的集合, T 表示系统中基于 $V\cup V'$的迁移关系, 那么我们把状态 s_2 称为状态 s_1 的后继, 当且仅当 $s_1\cup s_2'\vDash T$, 标记为$(s_1,s_2)\in T$.

有限序列 $s_0,s_1,\ldots,s_k$ 被称为一条长度为 k 的路径, 当每个(s_i,s_{i+1})对 $0\leqslant i\leqslant k-1$ 都属于 T. 我们称状态 t 是从状态 p 出发 k 步可达的, 当存在一条有限长度为 k 的路径使得 $s_0=p$ 以及 $s_k=t$ 成立. 所有从初始状态 I 出发可以到达的状态都称为 Sys 的可达状态. 给定一个安全性质 P(通常表示为布尔公式)和一个布尔迁移系统 $Sys=(V,I,T)$, 若每个 Sys 的可达状态 s 满足 P, 我们称该系统对 P 是安全的[45], 也就是说 $s\vDash P$; 否则, 这个系统是不安全的. 我们把违反性质 P 的状态称为坏状态, 并且用$\neg P$(等价 bad)代表包含所有坏状态的集合. 一条从 I 中初始状态出发到某个$\neg P$ 中坏状态的路径称为反例(counterexample).

令 $X\subseteq 2^V$ 表示 Sys 中状态的集合. 我们定义关系 $R(X)$表示 X 中状态的后继的集合, 即 $R(X)=s'|(s,s')\in T$ 对 $s\in X$. 对 $i>1$, 我们定义 $R^i(X)=R(R^{i-1}(X))$. 类似地, 定义 $R^{-1}(X)$作为 X 中状态的前驱集合, $i>1$ 时则对应 $R^{-i}(X)$.

举例来说, 电路就是一种典型的迁移系统, 可以建模为多种格式. 如图 2 所示的迁移模型是某个电路建模后的展示, 建模格式为与非图(and-inverter graph, aig). 图 2 底部三角形表示变量输入, 椭圆形表示常量输入, 矩形表示锁存器存储周期信息; 实线为当前值, 虚线为当前值取反; 中间层圆形为与运算符; 顶部为输出, 三角形为输出目标即要验证的目标, 矩形为锁存输出, 表示对应序号锁存器下个周期的值.

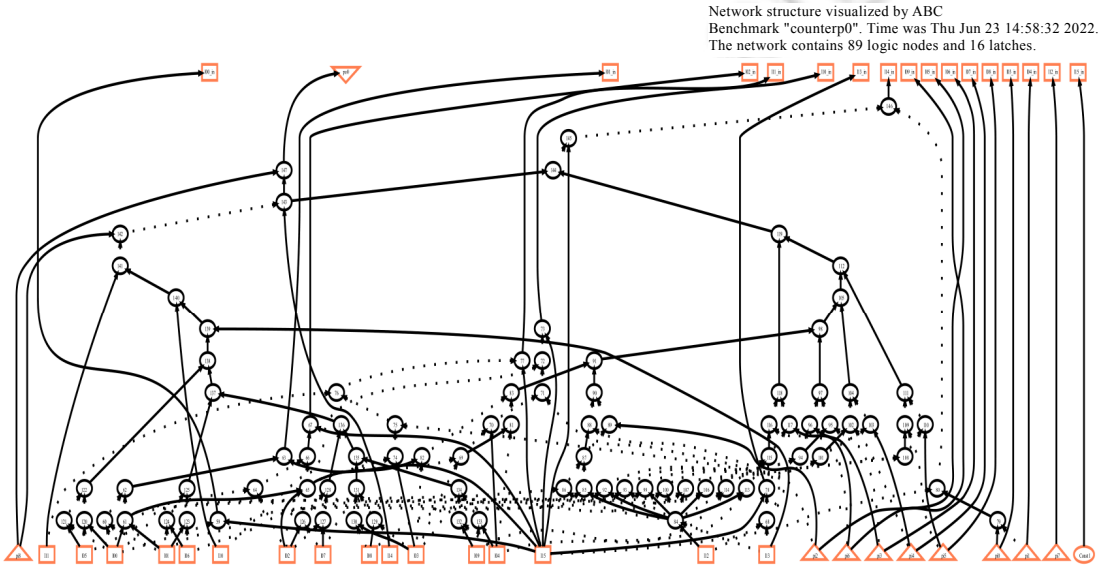


图 2 aiger 建模迁移系统示例

2.2 标记符号

V 中的每个变量 a 称为原子. 我们把布尔变量 a 或其否定$\neg a$ 称为文字(literal) l. 令 L 表示 literal 的集合. cube 表示$\wedge l$ 形式的布尔公式, 其中, $l\in L$, 即若干个 literal 的合取式, 例如 $l_1\wedge l_2\wedge\ldots\wedge l_k$, $k\geqslant 1$. 对应地, clause 表示$\vee l$ 形式的布尔公式, 其中, $l\in L$, 即若干个 literal 的析取式, 例如 $l_1\vee l_2\vee\ldots\vee l_k$, $k\geqslant 1$. 很明显, cube 的否定是 clause; 反之亦然. 且并不难看出: *Sys* 中的任意状态是 cube, 即 literal 的合取式. 为了便于描述, 在本文后面的章节, 我们也会混用术语状态(state)和 cube.

令 C 是 cube 的集合(与此相对应的是 clause), 我们定义布尔公式 $f(C)=\vee_{c\in C}c$(对应地, $f(C)=\wedge_{c\in C}c$). 为了简单些, 当它出现在布尔公式中时, 我们用 C 表示 $f(C)$. 例如, 公式$\phi\wedge C$ 和$\phi\vee C$ 是$\phi\wedge f(C)$和$\phi\vee f(C)$的简写.

一个 cube(或 clause) c 视为 literal 的集合、一个布尔公式还是一个状态集合, 取决于它使用的上下文. 如果 c 出现在一个布尔公式中, 例如 $c\Rightarrow\phi$, 则它被视为一个布尔公式. 如果我们称集合 c_1 是 c_2 的子集, 那么我们把 c_1 和 c_2 视为 literal 的集合. 如果我们说一个状态 st 属于 c, 那么我们把 c 视为一个状态(state).

我们使用符号 $s(x)/s'(x')$表示状态 s 的当前/下个版本. 类似地, 使用符号$\phi(x)/\phi'(x')$表示布尔公式ϕ的当前/下个版本. 对于迁移公式 T, 我们用标记 $T(x,x')$突出它同时包含当前和下个变量. 考虑一个布尔公式ϕ, 它的字母表是 $V\cup V'$并且是合取范式(CNF). 若ϕ是可满足的, 即有一个完整赋值 $A\in 2^{V\cup V'}$使得 $A\vDash\phi$. 另外, 存在一个部分解 $A^p\subseteq A$, 使得对于每个全解 $A'\supseteq A^p$ 有 $A'\vDash\phi$始终成立. 在后续章节, 我们使用标记 $pa(\phi)$表示ϕ的一个部分解, 并且用 $pa(\phi)|_x$表示通过只投影变量到 V 得到的 $pa(\phi)$的子集. 另一方面, 若ϕ是不可满足的, 存在一个最小不可满足核(MUC) $C\subseteq\phi$(这里ϕ被视为 clause 的集合)使得 C 是不可满足的, 并且每个 $C'\subset C$ 都是可满足的. 在后续章节, 我们使用标记 $muc(\phi)$表示这样的一个ϕ的 MUC, 并且使用 $muc(\phi)|_{c'}$代表通过把 clause 只投影到 c'得到的 $muc(\phi)$的子集. 由于 c'是一个 cube, 因此 $muc(\phi)|_{c'}$也是一个 cube.

2.3 布尔可满足性求解

SAT 问题指的是: 给定一个布尔公式, 公式中的变量是否存在一组赋值, 使得公式值为真. SAT 问题也是第一个被证明的 NP 完全问题, 许多问题可以归约为 SAT 问题, 因此, 这个问题也是自动化验证的基础过程.

现代符号模型检测方法常用到 SAT 求解器, SAT 求解器是一种可以用来判定布尔可满足性问题的程序. 给定一个布尔公式, 送给 SAT 求解器求解, 若公式是可满足的, 求解器可以返回一组赋值, 这组赋值能够使得布尔公式的值为真. 若公式是不可满足的, 即不存在这样一组赋值, 使得公式的值为真, 求解器会返回公式中的部分内容, 称为不可满足核, UC 表示了该公式不可满足的原因.

为了提高求解效率, 在求解布尔可满足性问题的时候, 通常把合取范式作为输入, 若输入不是合取范式的形式, 可以通过 Tseitin 转换得到合取范式的形式. 把若干个 clause 的合取式作为条件, 交给 SAT 求解器求解, 即可判定该公式是否是可满足的. 实践中, 往往把模型及要验证的性质以布尔公式的形式建模为若干个 clause 合取的形式, 这样就可以通过 SAT 求解器求解当前状态可以转移到的状态以及进行可达性分析.

布尔公式可满足性求解示例如图 3 所示. 图中左侧为一个合取范式的公式, 每个原子的值可以为 0 或 1. 对于这样的一个公式, 要判定其是否是可满足的, 目前有多种算法用于可满足性判定, 如 DPLL[46]、冲突驱动的子句学习(conflict driven clause learning, CDCL)[47]. 通过这些算法判定, 能够得到一组值使得该公式为真, 原子赋值如图 3 右侧所示, 即该公式是可满足的.

目前有多种开源 SAT 求解器可供使用, 如 MiniSAT[48], Glucose 等[49]. 一款 SAT 求解器通常支持以下 API 调用.

- *IsSAT*(ϕ), 判定输入公式ϕ的可满足性;
- *AddClause*(*clause*), 向 SAT 求解器中添加子句;
- *SATAssume*(*assumption*,ϕ), 判定基于额外假设(给一些原子变量预设的值)*assumption* 下的公式ϕ的可满足性;
- *GetModel*(·), 当 *IsSAT*(ϕ)或 *SATAssume*(*assumption*,ϕ)为真时, 通过该方法可以获得该调用的一个模型,

模型中包含该次调用的待判定公式中每个原子变量的值(true 或 false). 并且, $GetModel|_V$ 返回该模型在 V 上的投影结果;

- *GetUNSATAssumptions*(·), 当 *IsSAT*(ϕ)或 *SATAssume*(*assumption*,ϕ)为假时, 通过该方法可以获得一该调用的假设文字的不可满足核(unsatisfied core, UC). 该不可满足核可以为假设文字的一个子集, 该子集足够使得公式不可满足.

利用布尔可满足性的判定, 可以进行可达性分析, 以此进行模型检测.

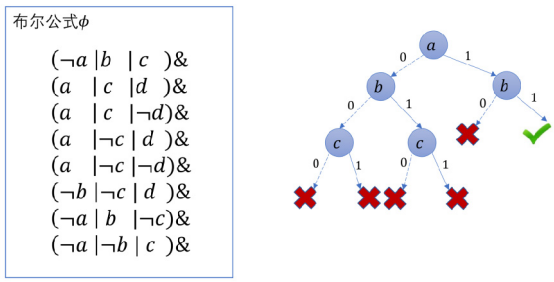


图 3 SAT 求解示例

2.4 符号模型检测

利用自动机理论, 我们可以进行模型检测. 自动机的状态图可能非常大. 在最坏的情况下, 系统自动机的状态数是系统中寄存器(或其他状态保持元件)可能配置的数量, 它是指数的. 对于大多数性质语言来说, 自动机中与性质对应的状态数量也与性质 literal 的大小呈指数相关. 因此, 上述蛮力方法对于大型硬件设计是没有希望的. 从理论上讲, 模型状态空间的大小随着模型的规模呈指数增长. 我们将模型检测领域的这个瓶颈称为"状态空间爆炸"[50]. 正是因为这个瓶颈的存在, 模型检测的实际应用受到极大限制. 过去模型也一般是显式表示, 即拿到一个模型, 该模型直接包含了所有状态及状态间的迁移关系. 这种表示方式使得模型检测算法无法处理非常大规模的问题.

为了解决这个问题, 学者们已经开发了许多启发式方法来避免显式地构建状态图. 因此, 现在模型一般用隐式表示, 即模型中不包含所有状态, 而是把状态间的迁移关系编码为布尔公式, 每次通过计算得到状态. 目前流行的模型检测算法都是符号模型检测算法, 即将目标迁移系统建模为符号表示, 例如硬件迁移系统. 每个变量用一个值为 0 或 1 的数字表示, 即 false 和 true. 在符号模型检测中, 安全性检测是通过符号可达性分析来实现的.

在显式模型检测中, 模型中任意两个状态的迁移关系被显式存储, 占用空间大, 难以适用到大规模的问题中. 而符号模型检测[51]把模型符号化, 并且把状态间的迁移关系通过隐式的方式存储. 给定一个状态 s, 可以通过计算的方式, 通过求解器和迁移关系 T 得到后继状态. 现代大部分模型检测方法都使用符号模型检测的方式取代了过去的显式模型检测, 使得模型检测适用的问题规模得到大幅度提高.

如图 4, 我们可以通过 T 计算 s 的后继状态, 也可以通过 T 判定可达性.

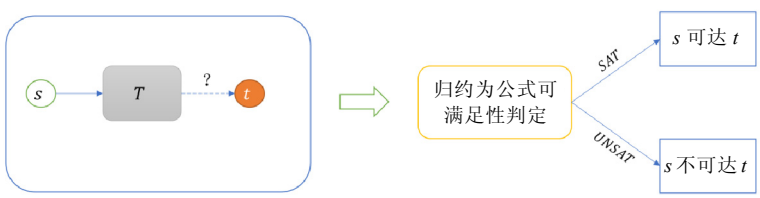


图 4 隐式模型中判定可达性示意图

把模型符号化, 还支持对模型进行抽象和简化, 更多的优化技术使得模型检测应用的规模能够不断提高.

3 UAIR 框架

在本节, 我们会展示 UAIR 的方法概览, 并介绍 UAIR 设计框架和相关定义、定理及证明.

3.1 方法概览

在基于 SAT 的模型检测方法中, 经常会得到 SAT 求解器返回的不可满足核, 而不可满足核蕴含的信息往往被忽略或利用一部分作为优化. 我们发现: 可以直接利用 SAT 求解器返回的不可满足核来逐步地构造不变式, 并基于这样的不变式进行安全性证明和不安全性检测. 基于以上的思路, 下面通过例子对本文提出的可达性分析方法进行说明. 总的过程如图 5–图 8 所示, 图中可达状态空间仅表示当前已探测可达状态空间.

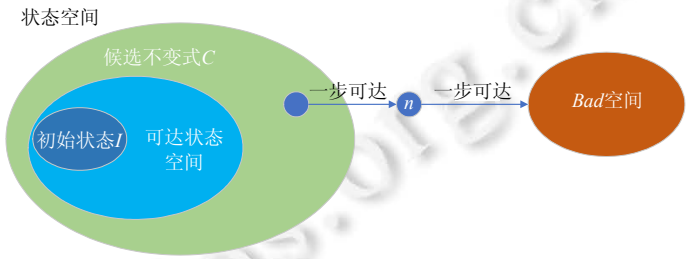


图 5 UAIR 方法概览 1

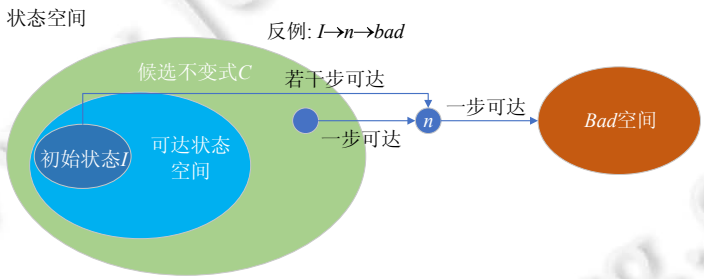


图 6 UAIR 方法概览 2

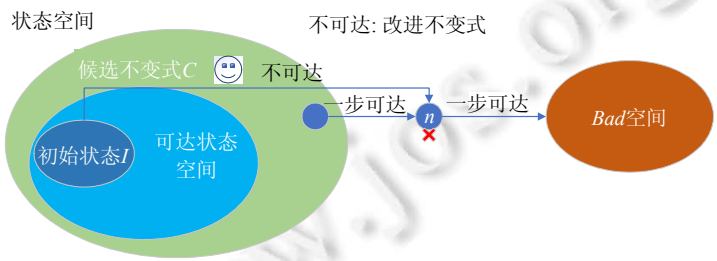


图 7 UAIR 方法概览 3

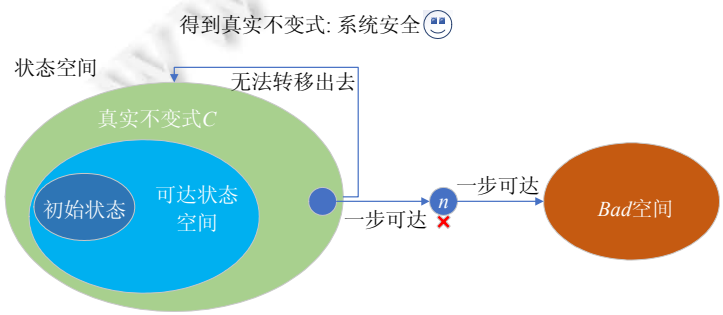


图 8 UAIR 方法概览 4

初始目标为检测从初始状态 I 出发是否可以转移到 *bad* 状态空间. 图 5 中, 我们以 *bad* 为目标, 基于不可满足核构造的候选安全不变式包含一步可达不满足 C 的状态 n, 且 n 可以一步转移到 *bad* 状态. 接下来, 以 n 为目标, 以相同方式检测从 I 出发是否可以转移到 n, 这是一个递归过程.

如图 6 所示, 若从 I 出发经过若干步可以转移到 n, 则证明从 I 出发存在一条路径可以转移到违反性质的状态, 则检测出系统不安全且找到一个反例. 该反例表示从初始状态 I 出发, 可经过若干步转移到状态 n, 再经过一步转移到违反安全性质 P 的 *bad* 状态空间.

如图 7 所示, 若从初始状态 I 出发, 经过若干步无法转移到 n, 且能够证明 n 是从 I 出发不可达的状态(该过程会构造一个针对 n 的安全不变式, 而对于整个 *bad* 空间来说则是候选安全不变式), 则不再把 n 作为待检测状态(通过候选安全不变式来实现). 继续到图 8 所示状态, 由于从候选安全不变式 C 出发, 可转移状态除了 n 之外为空, 且 n 已经被排除在外, 因此归纳得到 C 为安全不变式, 证明系统是安全的, 即从初始状态 I 出发不可达 *bad* 空间.

图 9 是一个抽象实例, 其中, s_0, s_1, s_2, s_3, s_4 为从初始状态 s_0 出发实际可达的状态, 而 s_5, s_6, s_7, s_8 是可以转移到违反安全性质 P(即 *bad*)的状态.

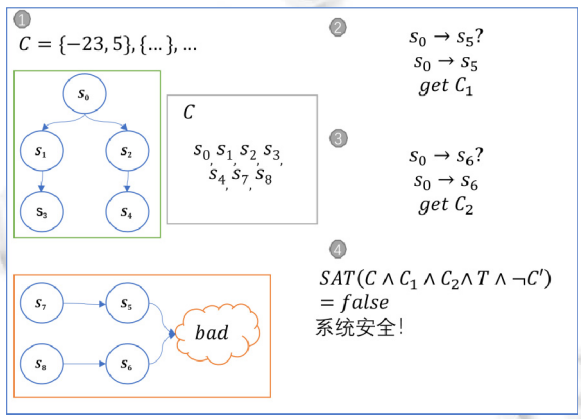


图 9 UAIR 方法示例

在我们提出的可达性分析中, 假设已经拿到了一个目标为 *bad* 的候选安全不变式 C, C 包含多个 UC, 如{–23,5}等, 其含义为: 如果一个状态中序号为 23 和 5 的变量值分别为 false 和 true, 则这个状态不可能在一步之内转移到 *bad*. 在此案例中, 候选安全不变式 C 当前已经把状态 s_0, s_1, s_2, s_3, s_4, s_7, s_8 包含在内.

在改进 C 的过程中, 会发现 C 中存在状态 s_7 能够转移到¬C 的 s_5, 因此, 检测变为判定从 s_0 到 s_5 的可达性. 经过检测发现: 从 s_0 到 s_5 不可达, 得到一个针对 s_5 不可达的安全不变式 C_1. 因此, C_1 可以在后续检测中作为一个约束来减少不必要的探索.

继续改进 C, 发现 C 中存在状态 s_8 能够转移到¬C 的 s_6, 因此, 检测变为判定从 s_0 到 s_6 的可达性. 经过检测发现: 从 s_0 到 s_6 不可达, 得到一个针对 s_6 不可达的安全不变式 C_2. 因此, C_2 也可以在后续检测中作为一个约束来减少不必要的探索. 此时, C 中包含的所有状态在 C_1 和 C_2 的约束下, 这些状态都无法转移到¬C, 因此得到结论, C 也是一个安全不变式. 因为从 I 出发可达的所有状态都无法转移到 *bad*, 该系统是安全的.

3.2 理论框架

总的来说, 与其他现有模型检测算法不同, UAIR 利用不可满足核来对模型进行检测, 包括安全性证明和不安全性检测. 我们使用从 SAT 求解器的可满足性判定中获得的不可满足核构造候选安全不变式, 随后, 在检测过程中, 以递归的方式改进候选安全不变式, 直到获得安全不变式, 证明系统是安全的, 或找到一个反例证明系统不安全. 我们相信, 这个方法可以作为现代模型检测工具集中一个有价值的补充.

从初始状态 I 开始, UAIR 在检测的同时构造候选安全不变式. 即: 在检测的过程中, 利用不可满足核把所

有无法一步转移到¬P(即 bad)的状态集合添加进不变式; 同时, 遍历当前状态可以转移到的在不变式之外的状态. 继续此过程, 直到找到反例, 或完成候选安全不变式构造. 候选安全不变式保证范围内的状态都是无法一步转移到¬P 的状态, 但可能存在状态能够转移到候选安全不变式之外. 通过候选安全不变式求解是否存在状态 s 和状态 n, 其中: s 属于候选安全不变式, n 不属于候选安全不变式, s 可一步转移到 n. 若 n 可以一步转移到¬P, 屏蔽状态 n, 以相同方式判断 I 是否能够转移到 n. 如果 I 可以转移到 n, 则找到一条路径可以从 I 到¬P; 否则, 继续寻找新的 n. 直到没有状态能够转移到不变式之外, 此时, 该候选安全不变式为安全不变式, 可以证明系统的安全性.

在本文中, 我们引入了候选安全不变式的概念, 即该不变式可以通过后续的改进过程成为一个真实的不变式.

定义 1(安全不变式及候选安全不变式). 给定一个布尔迁移系统 $Sys=(V,I,T)$和安全性质 P, 我们称 Sys 中的状态空间集合 C 是一个安全不变式, 当它同时满足以下 3 个条件时:

(1) $I\Rightarrow C$, 即初始状态包含在 C 里;

(2) $C\wedge T\Rightarrow P'$, 即 C 中所有状态的后继都满足 P;

(3) $C\wedge T\Rightarrow C'$, 即 C 中所有状态的后继也都在 C 中.

特别的, 我们称 C 为一个候选安全不变式, 当它仅满足条件(1)和条件(2)时.

若一个 Sys 中的状态集合 C 在给定 P 的情况下满足定义 1, 我们用术语“C 是关于(Sys,P)的安全不变式(或候选安全不变式)”来表示.

定理 1. 给定一个布尔迁移系统 $Sys=(V,I,T)$和安全性质 P, Sys 满足 P 当且仅当存在 Sys 中的一个状态集合 C 为(Sys,P)的安全不变式.

证明:

- (⇐) 我们证明: (a) C 中包含从初始状态 I 出发所有的可达状态; (b) C 中的每个状态 s 都满足 P, 即 $s\Rightarrow P$. 对于条件(a), 根据定义 1 中的条件(1)和条件(3)可以推导出结论; 对于条件(b), 根据定义 1 中的条件(1)和条件(2)可以得出结论. 由于条件(a)和条件(b)均成立, 所以 Sys 满足 P;
- (⇒) 若 Sys 满足 P, 则存在一个状态集合 C, 使得: (a) C 中包含从初始状态 I 出发所有的可达状态; (b) C 中的每个状态 s 都满足 P. 将满足条件(a)、条件(b)的 C 代入定义 1, 可以发现, C 是一个安全不变式.

  证毕.

显然, 如果我们能找到一个安全不变式, 那么就可以证明系统 Sys 是满足 P 的. 但一般来说, 要同时满足安全不变式的3个条件可能较难达到, 所以我们可以先构造候选安全不变式, 再得到最后的安全不变式. 具体地, 根据以上候选安全不变式的定义, 我们可以得到定理 2.

定理 2. 给定一个布尔迁移系统 $Sys=(V,I,T)$和安全性质 P, 若 C_1 为(Sys,P)的候选不变式, 而 C_2 为(Sys,P_1)的候选不变式, 这里 $C_1\Rightarrow P_1$ 成立, 那么 $C_1\cap C_2$ 也为(Sys,P)的一个候选安全不变式.

证明: 容易证明: $I\Rightarrow C_1\cap C_2$ 和$(C_1\cap C_2)\wedge T\Rightarrow P'$在 C_1 为(Sys,P)的候选不变式, C_2 为(Sys,P_1)的候选不变式的前提下都成立. 所以, $C_1\cap C_2$ 也为(Sys,P)的候选不变式.

定理 2 表明: 我们可以构造一个严格单调的候选安全不变式序列, 使得它最终会逼近安全不变式. 这也构成了本文的核心定理.

定理 3(核心定理). 给定一个布尔迁移系统 $Sys=(V,I,T)$和安全性质 P, Sys 满足 P 当且仅当存在一个有限的状态序列 $C_1,C_1\cap C_2,C_1\cap C_2\cap C_3,\ldots,C=C_1\cap C_2\cap\ldots\cap C_n$($n\geqslant 1$)使得: (a) C 为安全不变式; (b) C_1 为(Sys,P)的候选不变式; 且(c) $C_1\cap C_2\cap\ldots\cap C_{k+1}$ 为(Sys,$C_1\cap C_2\cap\ldots\cap C_k$)的候选不变式($n-1\geqslant k\geqslant 1$).

证明:

- (⇐) 由于 C 是安全不变式, 所以根据定义 1 直接可以得出 Sys 是满足 P 的;
- (⇒) 当 Sys 满足 P 时, 我们提出以下构造算法来构造该候选安全不变式的序列:

(1) 首先构造 C_1, 使得它满足 $I \Rightarrow C_1$ 和 $C_1 \wedge T \Rightarrow P$ 成立, 即 C_1 中的状态都是一步不可达$\neg P$. 根据定义 1, 我们知道 C_1 是一个候选安全不变式;

(2) 若存在某个状态 t 的后继在$\neg P$ 中并且它是 C_1 中某个状态 s 的后继, 那么我们构造 C_2, 使得它满足 $I \Rightarrow C_2$ 和 $C_2 \wedge T \Rightarrow \neg t$ 成立, 即 C_2 中的状态都是一步不可达 t 的. 根据定义 1, 我们知道 C_2 为(Sys,C_1)的候选不变式. 根据定理 2, 可以证明 $C_1 \cap C_2$ 是(Sys,P)一个候选安全不变式, 并且 $C_1 \cap C_2$ 是严格小于 C_1 的, 因为 s 包含在 C_1 但不包含在 C_2 中;

(3) 重复以上的过程, 我们就可以不断地构造该候选安全不变式序列, 直至最终求得安全不变式.

通过以上构造方法, 我们保证了该候选安全不变式序列是严格单调的, 所以它最终一定会收敛到安全不变式上.

下面的算法 1 具体实现了我们构造候选安全不变式和安全不变式的过程.

算法 1. UC-based Approximately Incremental Reachability (UAIR).

Input: 模型 M 和安全性质 P;

Output: 系统不安全和反例,或者是系统安全.

```
1:  function MAIN(M,bad)
2:      inv←∅
3:      s0←I
4:      t←bad
5:      bigCList←∅   //存储已获得安全不变式
6:      blockNList←∅   //存储求解中获得的可一步转移到目标 t 的状态 n
7:      if CHECK(t,inv,0) then
8:          return UNSAFE
9:      else
10:         return SAFE
11:     end if
12: end function
```

```
1:  function BOOL CHECK(t,inv,depth)
2:      block each n in blockNList in current solver, except t
3:      add each C in bigCList to current solver
4:      C_depth←∅
5:      if TRYSOLVE(initState,t,C_depth) then   //initState: 全局变量, 来自对 M 的解析
6:          return True
7:      else
8:          while SAT(C_depth ∧ T ∧ ¬C'_depth) do
9:              n←n∈¬C_depth
10:             if SAT(n∧T∧t') then
11:                 inv2←∅
12:                 block n in current solver
13:                 blockNList←blockNList∪n
14:                 if CHECK(n,inv2,depth+1) then
15:                     return UNSAFE
16:                 else
```

```
17:              add each C not added in bigCList to current solver
18:              block each n not blocked in blockNList in current solver, except t
19:            end if
20:          else
21:            c←uc∈n    //uc 来自 SAT 求解器(如 MiniSAT)
22:            C_depth←C_depth∪c
23:          end if
24:        end while
25:        bigCList←bigCList∪C_depth
26:        inv←C_depth
27:        return SAFE
28:      end if
29:  end function

1:   function BOOL TRYSOLVE(s,t,C)
2:     if SAT(s∧T∧t′) then
3:       return True
4:     else
5:       c←uc∈s
6:       C←C∪c
7:       while SAT(s∧T∧¬C′) do
8:         m←m∈¬C
9:         if TRYSOLVE(m,t,C) then
10:          return True
11:        end if
12:      end while
13:      return False
14:    end if
15:  end function
```

算法结束

Main 作为程序的入口, 其中, *bad* 是 *P* 的取反, *I* 是初始状态. *Check* 和 *TrySolve* 是算法的两个主要过程.

算法中, *TrySolve* 函数的作用是求出定理 3 中所述的单个 $C_i(i\geq 1)$. 该函数接收 3 个参数: 第 1 个参数 *s* 是出发状态, *t* 是目标状态, *C* 是作为引用返回的候选安全不变式. 首先判定 *s* 是否可以一步转移到 *t*: 若公式可满足, 则 *s* 可以一步转移到 *t*, 返回 true; 否则, 从 SAT 求解器拿到一个不可满足核 *c*(此处对应于算法中函数 *TrySolve* 的第 2–5 行), 表示从 *s* 无法转移到 *t* 的原因, 也代表多个状态. 接下来遍历状态 *s* 可以一步转移到的状态 $m(m\in\neg C)$, 把 *m* 作为新的 *s* 调用 *TrySolve* 递归求解, 判定 *m* 是否可达 *t*(此处对应于算法中函数 *TrySolve* 的第 7–12 行): 若检测过程中某个 m_i 可达 *t*, 则 *s* 可达 *t*, 返回 true; 否则, 返回 false 和 *C*.

Check 函数是检测的主函数, 负责检查构造的候选安全不变式是否已经是一个安全不变式. 它的第 1 个参数 *t* 是目标状态, *inv* 是当从 s_0 出发不可达 *t* 时返回的安全不变式, C_{depth} 是当前递归深度. 在 *Check* 过程中,若 *TrySolve* 返回的为 true, 则说明从 s_0 到 *t* 可达, 即找到了一条路径, s_0 出发可以转移到 *t*. 若 *TrySolve* 返回的为 false, 则获得一个候选安全不变式 *C*, *C* 中所有状态满足无法 1 步或 0 步转移到状态 *t*(此处对应于算法中函数 *Check* 的第 5–8 行). 但由于 *C* 是候选安全不变式, 所以要进一步判定 *C* 是否是安全不变式, 通过

$SAT(C_{depth} \wedge T \wedge \neg C'_{depth})$ 遍历可以从 C 中转移到的不满足 C 的状态, 若无此类状态或已遍历所有此类状态且未检测到 *unsafe*, 则说明 C 是安全不变式(此处对应于算法中函数 *Check* 的第 8–27 行); 否则, 检测到违反性质 P 的状态, 且该状态从 s_0 出发可达.

定理 4(算法正确性和完备性). 算法 1 返回 *SAFE* 当且仅当模型 M 满足性质 P.

证明: 在以$\neg P$ 为目标的不变式 C 的归纳过程中, 通过 *TrySolve* 过程可以得到初始候选安全不变式 C. 第 2 步, 在 C 中检查每个满足其他安全不变式的状态 s 能够转移到的状态 n, $n \in \neg C$ 且 $SAT(n \wedge T \wedge \neg P')$可满足. 判定 I 是否可达 n: 若 I 可达 n, 即找到一个反例证明系统是不安全的; 若不可达, 则重复第 2 步. 随着检测过程中得到的安全不变式的增加和待检测 n 的减少, 待检测空间严格缩小, 再根据定理 3, 我们可以得出算法最终或是检测出漏洞, 或是证明系统安全而终止.

3.3 UAIR中的优化方法

一个候选安全不变式 C, 总是从初始状态 I 和目标 t 开始构建. C 满足的性质, 即包含初始状态 I, 同时, C 所包含的状态均无法在一步之内转移到 t. 由于这是一个候选安全不变式, C 中可能存在状态能够转移到 C 之外. 因此, 接下来对能够转移出去的 C 之外的状态继续进行检测. 通过 *TrySolve* 过程得到的候选安全不变式 C, 可以用于后续的检测过程.

给定一个候选安全不变式 C, 其基于目标 t 构建, 若从 C 出发可以转移到 C 之外的状态 n, 那么就对 n 进行判定, 判定其是否可以一步转移到 t(公式 $n \wedge T \wedge t'$是否可满足). 若 n 可以一步转移到 t, 即公式 $n \wedge T \wedge t'$是可满足的, 则以之前相同的方式判定从初始状态 I 出发是否可达 n. 若 n 是可达的, 即找到了一条路径, 从 I 出发, 经过若干步可达 n, 再经过一步可以转移到 t, 证明系统是不安全的.

部分解优化(partial assignment optimization)[52], 对于 *check* 过程中的状态 n 而言, 若 n 可以一步转移到目标 t, 则会继续以 n 为目标进行检测. 此时, 若能够在对 n 进行检测前得到一个表示范围更大且与 n 等功能的 n, 则能够加快检测速度.

除部分解优化以外, 在检测过程中, 我们利用了检测过程中针对$\neg P$ 之外的目标得到的安全不变式, 称为安全不变式约束优化. 首先, 对于一个包含初始状态 I 的安全不变式 C, 无论其目标为哪个状态, 从 I 出发可达的状态均处于 C 的范围之内. 因此, 对于检测过程中得到的任意一个安全不变式 C_i, 均可用于其他安全不变式的归纳过程.

具体实施如下: 在对当前候选安全不变式 C 进行归纳的过程中, 假设有状态 m 可以转移到状态 n, 其中, $m \in C$ 且 $n \in \neg C$, 为了剪枝以加快检测速度, 若 m 无法同时满足其他所有安全不变式 C_i, 则从 I 出发, m 必定不可达,即不需要判定从 I 到 n 是否是可达的. 即: 仅当可满足性判定 $IsSAT(C \wedge C_1 \wedge C_2 \wedge \ldots \wedge T \wedge \neg C')$可满足, $m \in C$, $n \in \neg C$, 才继续检测公式 $IsSAT(I \wedge T \wedge T \wedge \ldots \wedge T \wedge n')$是否可满足.

4 实验分析

4.1 评估准备

我们在公开基准数据上进行实验, 评估数据基于来自 2015 年和 2017 年硬件模型检测比赛(HWMCC 2015 和 HWMCC 2017)[53]Aiger[54]格式的 749 个基准, 针对安全性进行检测. 749 个基准包含 2015 年和 2017 年比赛的所有案例. 2015 年和 2017 年的 HWMCC 竞赛之后的 2019 年和 2020 年竞赛都使用的是 Aiger 新格式, 使用新格式, 不同的工具对约束的支持不同, 结果会有偏差, 所以我们在这篇论文里就暂不比较新格式的 AIGER 模型, 因此使用 2015 年及 2017 年的基准.

在工具方面, 我们实现了上文所描述的方法; 同时, 以 ic3-ref, pdr, imc, forward-car 和 backward-car 作为我们实验的评估基准.

我们关注的有两个方面: 一是我们方法的求解能力和求解速度; 二是我们的方法能否求解出其他几个方法都无法求解出来的例子. 对比结果方面, 我们以其他主要方法没有分歧的结果为参考, 且认为他们的结果

都是正确的.

在实验结果方面, 我们所实现的工具(后面简称 UAIR)所找出的所有反例已经经过第三方工具验证.

部署实验的硬件集群拥有 190 个计算节点, 机器类型为 SUGON, 每个节点拥有 2 颗 64 位 Intel Xeon Gold 6130 CPU, 主频为 2.1 GHz. 每颗 CPU 为 16 核, 内存类型为 DDR4, 8×16 GB.

实验对比的其他方法运行命令及参数见表 1.

表 1 实验中评估的工具与算法

工具	算法	配置参数
ABC	PDR (ABC-pdr)	-c ‘pdr’
ABC	IMC (ABC-int)	-c ‘int’
Ic3-ref	IC3 (ic3-ref)	-b
SimpleCar	Forward CAR (-f-m-pr-d-pa) Backward CAR (-b-m-pr)	-f -muc -propagate -dead -partial -b -muc -propagate

4.2 实验结果

本节主要对本方法的运行时间、求解效率进行分析和对比.

在上述实验环境, 我们设置运行时间为 1 h, 设置运行日志级别为最少, 求解结果如图 10 所示.

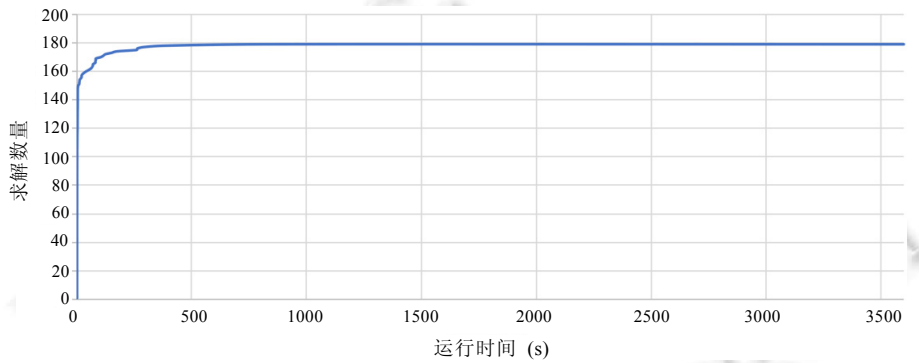


图 10 UAIR 求解结果

由图 10 可见: 在 1 h 内, 全部 749 个案例中, UAIR 可以求解出 179 个案例. 179 个案例中的大部分, UAIR 都在非常短时间内求解得到结果. 这 179 个案例的细节见表 2.

表 2 UAIR 实验结果

Method	Unsafe	Safe	Total Num	Total time (s)
UAIR	62	117	179	3 520.62

4.3 实验对比与分析

为了评估 UAIR 的有效性, 我们研究了以下问题: UAIR 与其他方法相比, 求解效率和求解性能如何.

为了验证这个问题的结果, 我们将 UAIR 与其他流行的符号模型检测方法进行了对比实验, 以 UAIR 为基准, 展示本方法相对其他方法能够多解出的案例数, 运行时间同样为 1 h, 实验的结果见表 3.

表 3 UAIR 与流行符号模型检测方法性能比较

Method	Unsafe	Safe
UAIR	62	117
Backward-car	−1	−8
Forward-car	−4	−3
Ic3-ref	−1	−3
Pdr	−3	−3
Imc	−2	−6

如表 3 所示:

- 与 backward-car 相比, 有 1 个 Unsafe 和 8 个 Safe 案例是其无法求解而 UAIR 可以解决的;

- 与 forward-car 相比, UAIR 能够单独求解出 4 个 Unsafe 和 3 个 Safe 案例;
- 与 IC3 相比, 有 1 个 Unsafe 和 3 个 Safe 案例是其无法求解而 UAIR 可以解决的;
- 与 PDR 相比, UAIR 能够单独求解出 3 个 Unsafe 和 3 个 Safe 案例;
- 与 IMC 相比, 有 2 个 Unsafe 和 6 个 Safe 案例是其无法求解而 UAIR 可以解决的.

从上面结果可以看出: 和这些发展时间久、比较成熟的方法相比, UAIR 无论与哪个方法相比都有独特的能够解出的案例. 作为一个全新的方法, 这样的结果表现很不错.

图 11 同时呈现了 UAIR 和其他几个方法的实验结果.

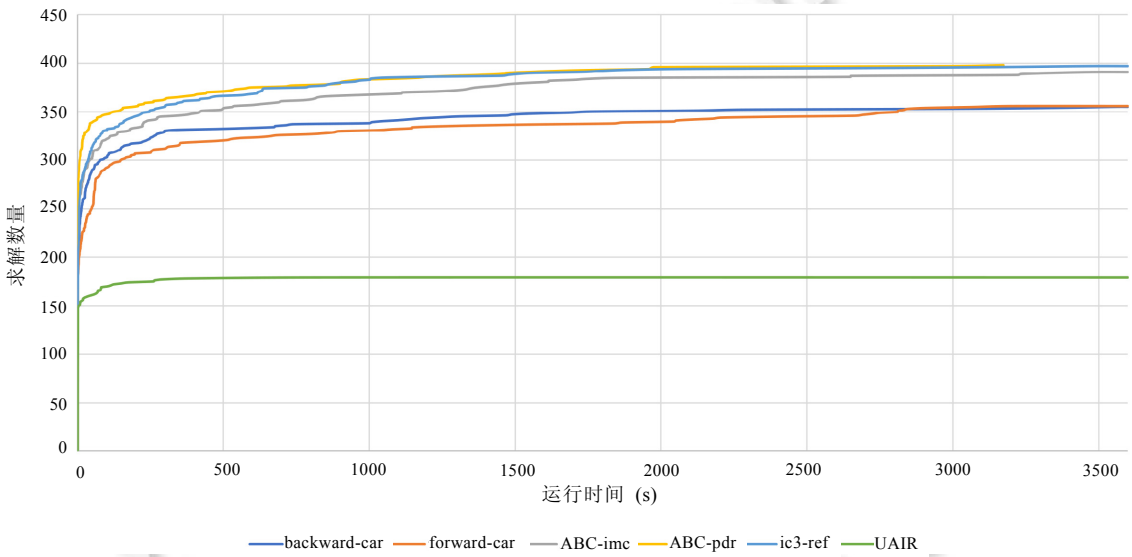


图 11 各方法运行结果时间图

如图 11 所示: 在可求解案例的速度方面, UAIR 与其他方法表现相近, 求解速度很快.

在运行 1 h 所求解的案例中, 有 3 个 UAIR 可求解案例是其他几个流行方法在相同时间内都无法求解出结果的, 我们对这几个案例进行了分析.

这 3 个案例规模都比较小, 分别包含 24, 28, 28 个锁存器, 包含的逻辑个数分别为 149, 143, 148, 内部结构似乎并不复杂. 而有些案例可能包含成百上千个锁存器, 且包含的逻辑更多, 其他方法求解能力比较强, 应该能很快求解出这几个案例, 而实际情况是无法在设定时间内解出.

通过分析日志文件发现: 这 3 个案例在构造候选安全不变式 C 的过程中, 状态空间的大量状态在检测的初期很快被 SAT 求解器返回的不可满足核所覆盖, 且在后续检测过程中, 能够转移到满足$\neg P$ 的状态路径较少, 据此进一步推测出能够转移到 *bad* 的状态总数较少, 因此能够更快收敛完成检测.

归纳来讲, 本文方法基于不可满足核来探索模型的状态空间, 而非通过存储可达状态探索模型状态空间, 每次求解不可满足时, SAT 求解器返回的不可满足核表示许多状态, 这些状态包括了当前状态, 也表示不可转移到目标状态的原因. 如果模型状态空间中的状态不可达$\neg P$ 的原因类型数量较少, 本方法求解会更快, 利用了 UC 蕴含的信息, 少数 UC 就能覆盖大量状态空间. 依赖于 UC 来实现检测过程, UC 所能覆盖的状态空间越大越好, 尽可能多地覆盖未检测空间, 符合这样特征的模型检测更快. 而 $SAT(C_{depth} \wedge T \wedge \neg C'_{depth})$ 过程是从$\neg P$ 前向搜索的过程, 可达$\neg P$ 的状态较少、从$\neg P$ 前向探索的路径较短时检测更快, 否则会有大量时间在无用、琐碎的空间探索上, 因为在前向搜索的过程中大量状态从初始状态出发不可达. 而对于模型规模较大且状态不可达$\neg P$ 原因的种类数比较多的案例, 本方法可能不适合求解. 因为这种情况下, 每次求解不可满足时, SAT 求解器返回的 UC 覆盖的状态较少, 效率会比较低; 另外, 若模型中可达$\neg P$ 的状态数较多, 或存在较多从可

达$\neg P$的状态前向搜索较长的路径, 也会有大量时间在无用、琐碎的空间探索上, 可能本方法效率会比较低.

在实验过程中, 我们也采用了多种优化方法改进 UAIR, 实现待探索状态空间的缩减以加快检测速度. 下面进行优化前后对比分析.

采用部分解优化前后对比结果如图 12 所示.

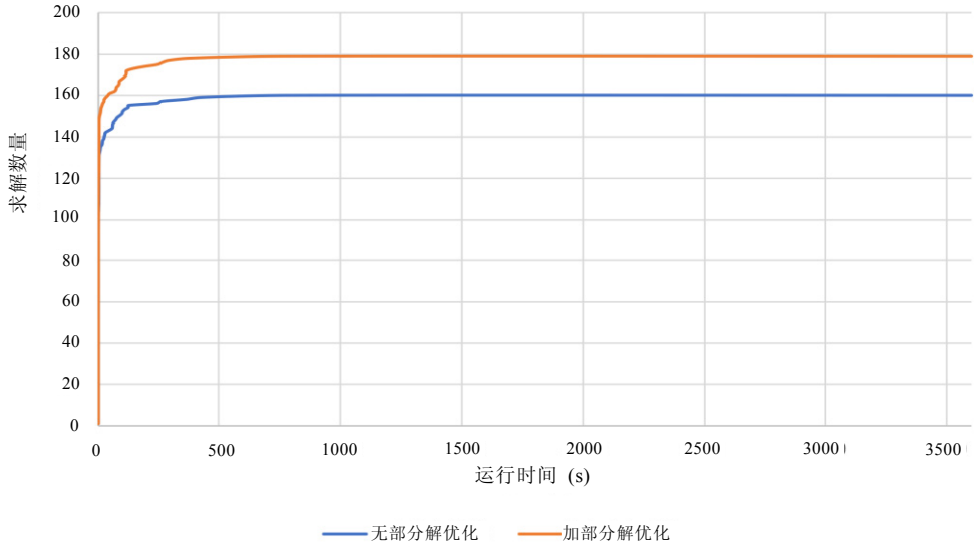


图 12 添加部分解优化前后对比

如图 12 所示: 加入部分解优化后, UAIR 在 1 h 可以求解的数量为 179, 而未加部分解优化时为 160, 求解数量提高约 12%, 并且求解速度有所提高, 证明部分解优化是一个有效的优化方法.

综上, 本方法基于不可满足核构造候选安全不变式并进行后续检测, 相比其他方法, UAIR 不需要显式存储状态, 主要思想是利用 UC 能够表示包含多个状态的状态空间来实现快速收敛, 帮助进行安全性证明和不安全性检测. 作为一个全新方法, 和其他已经发展多年甚至数十年的方法相比, 尽管总的求解数量与其他成熟方法如 IC3, CAR 等相比还存在一定距离, 但众所周知, 模型检测领域没有最好的方法, UAIR 求解效率表现优秀, 求解案例的数量让人眼前一亮, 并且能够求解出 3 个其他方法均无法求解的案例, 而基于不可满足核的可达性分析也有许多探索空间, 有很好的发展前景.

5 总 结

现代模型检测方法中, 不可满足核经常作为求解器的求解结果出现, 虽然蕴含着丰富的信息, 但是不可满足核的价值容易被忽略或仅是作为模型检测算法优化的一部分来挖掘. 在本文中, 我们提出了基于不可满足核设计的模型检测算法. 本方法利用了求解器返回的不可满足核的信息, 通过不可满足核构造候选安全不变式, 并在后续过程中不断改进候选安全不变式来实现安全性证明和漏洞查找, 设计思路直观. 作为一种全新的方法, 在实验中取得了较好的效果, 在基准集中的求解数量和求解速度都很不错, 并且相同时间内能够求解出 3 个其他对比方法都无法求解的案例, 证明了本方法的有效性, 也反映了挖掘不可满足核中蕴含的信息是模型检测可以继续探索的思路. 通过在大量基准上的测试及与其他现有工具的对比, 验证了本文提出的方法具有很好的前景和实际应用价值. 未来计划在本方法的基础上进行更多的改进, 加入更多对算法和实现的优化, 或是结合其他方法尝试以得到更好的效果, 相信在接下来的后续工作中, UAIR 可以赶上甚至超越其他方法的综合表现, 且本方法可以作为模型检测工具集很有价值的补充.

References:

[1] Golnari A, Vizel Y, Malik S. Error-tolerant processors: Formal specification and verification. In: Proc. of the 2015 IEEE/ACM Int'l Conf. on Computer-Aided Design (ICCAD). IEEE, 2015. 286−293.

[2] Rozier KY. Linear temporal logic symbolic model checking. Computer Science Review, 2011, 5(2): 163−203.

[3] Baier C, Katoen JP. Principles of Model Checking. MIT Press, 2008.

[4] Bernardini A, Ecker W, Schlichtmann U. Where formal verification can help in functional safety analysis. In: Proc. of the 2016 IEEE/ACM Int'l Conf. on Computer-aided Design (ICCAD). ACM, 2016. 1−8.

[5] Alrajeh D, Kramer J, Russo A, *et al*. Elaborating requirements using model checking and inductive learning. IEEE Trans. on Software Engineering, 2012, 39(3): 361−383.

[6] Ammann PE, Black PE, Majurski W. Using model checking to generate tests from specifications. In: Proc. of the 2nd Int'l Conf. on Formal Engineering Methods. IEEE, 1998. 46−54.

[7] Fuxman A, Pistore M, Mylopoulos J, *et al*. Model checking early requirements specifications in Tropos. In: Proc. of the 5th IEEE Int'l Symp. on Requirements Engineering. IEEE, 2001. 174−181.

[8] Heitmeyer C, Kirby J, Labaw B, *et al*. Using abstraction and model checking to detect safety violations in requirements specifications. IEEE Trans. on Software Engineering, 1998, 24(11): 927−948.

[9] Visser W, Havelund K, Brat G, *et al*. Model checking programs. Automated Software Engineering, 2003, 10(2): 203−232.

[10] Xie F, Levin V, Browne JC. Model checking for an executable subset of UML. In: Proc. of the 16th Annual Int'l Conf. on Automated Software Engineering (ASE 2001). IEEE, 2001. 333−336.

[11] Xie F, Levin V, Browne JC. Objectcheck: A model checking tool for executable object-oriented software system designs. In: Proc. of the Int'l Conf. on Fundamental Approaches to Software Engineering. Berlin, Heidelberg: Springer, 2002. 331−335.

[12] Beyer D, Keremoglu ME. CPAchecker: A tool for configurable software verification. In: Proc. of the Int'l Conf. on Computer Aided Verification. Berlin, Heidelberg: Springer, 2011. 184−190.

[13] Merz S. Model checking: A tutorial overview. Summer School on Modeling and Verification of Parallel Processes, 2000. 3−38.

[14] Kim M, Kim Y, Kim H. Unit testing of flash memory device driver through a SAT-based model checker. In: Proc. of the 23rd IEEE/ACM Int'l Conf. on Automated Software Engineering. IEEE, 2008. 198−207.

[15] Kim M, Kim Y, Kim H. A comparative study of software model checkers as unit testing tools: An industrial case study. IEEE Trans. on Software Engineering, 2010, 37(2): 146−160.

[16] Witkowski T, Blanc N, Kroening D, *et al*. Model checking concurrent Linux device drivers. In: Proc. of the 22nd IEEE/ACM Int'l Conf. on Automated Software Engineering. 2007. 501−504.

[17] Dwyer MB, Carr V, Hines L. Model checking graphical user interfaces using abstractions. ACM SIGSOFT Software Engineering Notes, 1997, 22(6): 244−261.

[18] Dwyer MB, Tkachuk O, Visser W. Analyzing interaction orderings with model checking. In: Proc. of the 19th Int'l Conf. on Automated Software Engineering. IEEE, 2004. 154−163.

[19] Artho C, Garoche PL. Accurate centralization for applying model checking on networked applications. In: Proc. of the 21st IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE 2006). IEEE, 2006. 177−188.

[20] Artho C, Leungwattanakit W, Hagiya M, *et al*. Cache-based model checking of networked applications: From linear to branching time. In: Proc. of the 2009 IEEE/ACM Int'l Conf. on Automated Software Engineering. IEEE, 2009. 447−458.

[21] Haydar M, Boroday S, Petrenko A, *et al*. Properties and scopes in Web model checking. In: Proc. of the 20th IEEE/ACM Int'l Conf. on Automated Software Engineering. 2005. 400−404.

[22] Eisler S, Scheidler C, Josko B, *et al*. Preliminary results of a case study: Model checking for advanced automotive applications. In: Proc. of the Int'l Symp. on Formal Methods. Berlin, Heidelberg: Springer, 2005. 533−536.

[23] Vörtler T, Rülke S, Hofstedt P. Bounded model checking of Contiki applications. In: Proc. of the 15th IEEE Int'l Symp. on Design and Diagnostics of Electronic Circuits & Systems (DDECS). IEEE, 2012. 258−261.

[24] Yang MF, Gu B, Duan ZH, *et al*. 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).

[25] Gligoric M, Majumdar R. Model checking database applications. In: Proc. of the Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Berlin, Heidelberg: Springer, 2013. 549−564.

[26] Song F, Touili T. Pushdown model checking for malware detection. Int'l Journal on Software Tools for Technology Transfer, 2014, 16(2): 147−173.

[27] Karna AK, Chen Y, Yu H, *et al*. The role of model checking in software engineering. Frontiers of Computer Science, 2018, 12(4): 642−668.

[28] Griggio A, Roveri M. Comparing different variants of the IC3 algorithm for hardware model checking. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, 2015, 35(6): 1026−1039.

[29] Biere A, Cimatti A, Clarke EM, *et al*. Symbolic model checking using SAT procedures instead of BDDs. In: Proc. of the 36th Annual ACM/IEEE Design Automation Conf. 1999. 317−320.

[30] McMillan KL. Interpolation and SAT-based model checking. In: Proc. of the Int'l Conf. on Computer Aided Verification. Berlin, Heidelberg: Springer, 2003. 1−13.

[31] Bradley AR. SAT-based model checking without unrolling. In: Proc. of the Int'l Workshop on Verification, Model Checking, and Abstract Interpretation. Berlin, Heidelberg: Springer, 2011. 70−87.

[32] Eén N, Mishchenko A, Brayton R. Efficient implementation of property directed reachability. In: Proc. of the 2011 Formal Methods in Computer-aided Design (FMCAD). IEEE, 2011. 125−134.

[33] Li J, Zhu S, Zhang Y, *et al*. Safety model checking with complementary approximations. In: Proc. of the 2017 IEEE/ACM Int'l Conf. on Computer-aided Design (ICCAD). IEEE, 2017. 95−100.

[34] Li J, Dureja R, Pu G, *et al*. Simplecar: An efficient bug-finding tool based on approximate reachability. In: Proc. of the Int'l Conf. on Computer Aided Verification. Cham: Springer, 2018. 37−44.

[35] HWMCC 2015. 2015. http://fmv.jku.at/hwmcc15/

[36] HWMCC 2017. 2017. http://fmv.jku.at/hwmcc17/

[37] Amla N, Du X, Kuehlmann A, *et al*. An analysis of SAT-based model checking techniques in an industrial environment. In: Proc. of the Advanced Research Working Conf. on Correct Hardware Design and Verification Methods. Berlin, Heidelberg: Springer, 2005. 254−268.

[38] Vizel Y, Gurfinkel A. Interpolating property directed reachability. In: Proc. of the Int'l Conf. on Computer Aided Verification. Cham: Springer, 2014. 260−276.

[39] Ivrii A, Gurfinkel A. Pushing to the top. In: Proc. of the 2015 Formal Methods in Computer-aided Design (FMCAD). IEEE, 2015. 65−72.

[40] Seufert T, Scholl C. Combining PDR and reverse PDR for hardware model checking. In: Proc. of the 2018 Design, Automation & Test in Europe Conf. & Exhibition (DATE). IEEE, 2018. 49−54.

[41] Dureja R, Gurfinkel A, Ivrii A, *et al*. IC3 with Internal signals. In: Proc. of the 2021 Formal Methods in Computer Aided Design (FMCAD). IEEE, 2021. 63−71.

[42] Seufert T, Scholl C, Chandrasekharan A, *et al*. Making PROGRESS in property directed reachability. In: Proc. of the Int'l Conf. on Verification, Model Checking, and Abstract Interpretation. Cham: Springer, 2022. 355−377.

[43] Brayton R, Mishchenko A. ABC: An academic industrial-strength verification tool. In: Proc. of the Int'l Conf. on Computer Aided Verification. Berlin, Heidelberg: Springer, 2010. 24−40.

[44] Cimatti A, Clarke E, Giunchiglia E, *et al*. Nusmv 2: An opensource tool for symbolic model checking. In: Proc. of the Int'l Conf. on Computer Aided Verification. Berlin, Heidelberg: Springer, 2002. 359−364.

[45] Kupferman O, Vardi MY. Model checking of safety properties. Formal Methods in System Design, 2001, 19(3): 291−314.

[46] Nieuwenhuis R, Oliveras A, Tinelli C. Solving SAT and SAT modulo theories: From an abstract davis-putnam-logemann-loveland procedure to DPLL (T). Journal of the ACM (JACM), 2006, 53(6): 937−977.

[47] Marques-Silva J, Lynce I, Malik S. Conflict-driven Clause Learning SAT Solvers. In: Handbook of Satisfiability. IOS Press, 2021. 133−182.

[48] Eén N, Sörensson N. An extensible SAT-solver. In: Proc. of the Int'l Conf. on Theory and Applications of Satisfiability Testing. Berlin, Heidelberg: Springer, 2003. 502−518.

[49] Audemard G, Simon L. Glucose: A solver that predicts learnt clauses quality. SAT Competition, 2009, 7–8.

[50] Clarke EM. The birth of model checking. In: Grumberg O, Veith H, eds. 25 Years of Model Checking Festschrift. LNCS 5000, Berlin, Heidelberg: Springer, 2008. 1–26.

[51] Burch JR, Clarke EM, McMillan KL, *et al*. Symbolic model checking: 10^{20} states and beyond. Information and Computation, 1992, 98(2): 142–170.

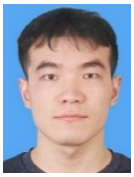
[52] Yu Y, Subramanyan P, Tsiskaridze N, *et al*. All-SAT using minimal blocking clauses. In: Proc. of the 27th Int'l Conf. on VLSI Design and the 13th Int'l Conf. on Embedded Systems. IEEE, 2014. 86–91.

[53] Biere A, Claessen K. Hardware model checking competition. http://fmv.jku.at/hwmcc15/

[54] Biere A. Aiger format. http://fmv.jku.at/aiger/FORMAT

附中文参考文献:

[24] 杨孟飞, 顾斌, 段振华, 等. 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.



于忠祺(1997－), 男, 硕士生, 主要研究领域为形式化验证, 模型检测.

李建文(1987－), 男, 博士, 研究员, 博士生导师, CCF 专业会员, 主要研究领域为形式化方法, 软硬件安全.

张小禹(1997－), 男, 博士生, CCF 学生会员, 主要研究领域为形式化方法, 模型检测算法.