Tamgram: A Frontend for Large-scale Protocol Modeling in Tamarin
文献分享

汇报人：林雅佳

日期：2025.09.25

Part1 研究背景

1. 协议验证工具的应用

• 协议验证器被用于验证复杂的现实世界协议，包含复杂的状态、共享资源、mutable state、fresh名称、并发交互等特性；

2. 手工写Tamarin规格的难度

• 大型协议规格往往非常庞杂，包含很多重复的结构、命名空间冲突、状态变量管理困难等问题，手工维护出错的可能性高；

• 控制结构、模块化复用、命名约定等在Tamarin的原始语法里支持不够直观或不够便利，写大模型时这些缺陷尤为显著；

3. 缺乏高层语言支持与形式语义保证

• 用户常通过宏或脚本等方式尝试模块化，但这些方式通常缺乏形式语义保证，不一定能保证翻译后的Tamarin规格与用户意图一致！

已有方案对可读性/模块化/抽象化的尝试

♠ 用模板、脚本pre-processor等工具来简化重复部分或命名空间管理，能在一定程度辅助重用或减少重复，但形式语义保证通常欠缺，翻译成标准规则后可能偏离意图、难以调试；

♠ 也有研究把协议设计、建模规范从结构化规格、元模型出发，再自动生成形式语言模型。

性能与规模的挑战

⇨ 随着协议状态复杂性上升，Tamarin的饱和、推导、规则交互等步骤非常耗时，模型难以维护、理解；

⇨ 手工写规格重复性高、易错，且在扩展时困难，已有工作中很少有兼顾“高层抽象+语义正确性+性能接近手工规格”的“前端”语言。

Part2 核心贡献

Contributions

① 高级“前端”语言设计

提出Tamgram作为Tamarin的高级“前端”语言，写可读性更高的复杂协议模型

② 模块化与可重用性增强

引入模块化理念、用户定义的谓词符号、扩展的let绑定点，以及支持hygienic macros，增强了代码的结构化和可重用性

③ 简化复杂过程建模

提供了一种进程语法，允许以易于使用的方式指定复杂的进程流，并支持进程局部内存的操作，从而简化了复杂过程的建模

④ 与Tamarin的紧密集成与形式化保证

Tamgram将Tamarin规则作为核心语法，允许访问几乎所有Tamarin特性，作者形式化证明了Tamgram进程的执行轨迹与Tamarin规则轨迹之间存在严格映射，确保建模直观、验证结果可靠。

Part3 技术实现-Syntax

A. Syntax（High-level abstraction for protocol modeling）

语法类别	关键语法项	符号	含义	核心用途
状态管理	单元格（Cell）	cell ::= 'x	进程本地内存载体，动态作用域，支持赋值（'x := y）与删除（undef('x)）	替代Tamarin的状态事实，封装协议状态
模块化与复用	模块（Module）	module name { decl ... } /import name	ML风格命名空间，支持定义、导入、别名	解决命名冲突，拆分复用代码
进程控制流	顺序执行	P ::= Pscoped; P	先执行Pscoped，再执行后续进程	显式串联协议步骤
	非确定性选择	P ::= choice{Pscoped, ...}; P	从多个分支中选一个执行	描述协议分支逻辑
	条件循环	P ::= while cond {P}; P	基于cond	处理超时重试等循环逻辑，替代Tamarin的递归模拟
规则兼容	MSR嵌入	rule ::= ruleL → ruleR /ruleL − ruleAR → ruleR	复用Tamarin多集重写规则结构，支持动作标签	衔接Tamarin，保留原生规则能力

Part3 技术实现-CFG

B. Control flow graph representation

Tamgram过程的语义不直接基于语法，而是通过CFG表示实现——CFG将过程的控制流（顺序、分支、循环）可视化、形式化，同时更贴近Tamarin的MSR规则语义

CFG 构造示例：

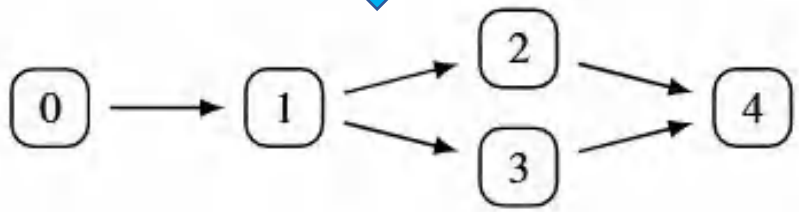
Tangram process code:

```
process A =
[ In(x) ]->[ 'a := x ];
choice {
{ [ 'a cas "1" ]->[ Out("2") ] };
{ [ 'a cas "A" ]->[ Out("B") ] }; };
 []->[ Out("End") ]
```

Convert to CFG vertex (V)

The set V then consists of:
(0, [Fr(pid)] -> [’pid := pid])
(1, [In(x)] -> [’a := x])
(2, [’a cas “1’’] -> [Out(“2’’)])
(3, [’a cas “A’’] -> [Out(“B’’)])
(4, [] -> [Out(“End’’)])

Convert to CFG edge (E)



CFG是语法 -> 语义 -> Tamarin翻译的桥梁

Part3 技术实现-Labeled Operational Semantics

C. Labeled operational semantics of Tamgram

$$(\textsc{Fresh})\ \frac{}{\langle (S,K,M), [] \dashv[\,]\mapsto [Fr(x)], (S,K,M)\rangle}$$
where x is a fresh name

$$(\textsc{Start})\ \frac{(k, [Fr(id)] \dashv[\,]\mapsto ['pid := id]) \in V}{\langle (S,K,M), [Fr(id)] \dashv[\,]\mapsto ['pid := id], \quad (S,K',M')\rangle}$$
where $k \in roots(E)$
$k' \in succ(E,k)$
$K' = K \cup \{id' \mapsto k'\}$,
$M' = M \cup \{id' \mapsto \{'pid \mapsto id'\}\}$

$$(\textsc{Rule})\ \frac{\begin{array}{c}(k, l \dashv[\,a\,]\mapsto r) \in V \quad l' \dashv[\,a'\,]\mapsto r' \in ginsts(l \dashv[\,a\,]\mapsto r) \\ deref(m,l') \subseteq^{+} S \quad K = K' \cup \{id \mapsto k\} \quad M = M' \cup \{id \mapsto m\} \quad cu_r(l \dashv[\,a\,]\mapsto r) \subseteq \mathbb{D}(m)\end{array}}{\langle (S,K,M), l' \dashv[\,a'\,]\mapsto r', \quad (S', K' \cup \{id \mapsto k'\}, M' \cup \{id \mapsto m_{old} \cup m_{new}\})\rangle}$$
where $k' \in succ(E,k)$ is picked non-deterministically $\quad m_{new} = defs(r')$

这三个规则共同构建了Tamgram系统的标记转移语义（理论层面的形式化定义）：

1. FRESH提供 “新鲜值” 的基础能力；
2. START启动新进程，初始化执行环境；
3. RULE是核心执行逻辑，驱动协议按MSR规则完成 “事实->动作->新事实 + 内存变化” 的循环

Part4 案例分析

Original Tamarin Model：

```
rule Terminal_Sends_GPO:
  let ... in
  [Fr(~UN), !Value($amount, value)]
  --[OneTerminal(), Role('Terminal')]->
  [Out(<'GET_PROCESSING_OPTIONS', PDOL>),
   Terminal_Sent_GPO($Terminal, PDOL)]  // 手动创建状态事实
rule Terminal_Sends_ReadRecord:
  [Terminal_Sent_GPO($Terminal, PDOL),  // 手动消费状态事实
   In(<AIP, 'AFL'>)]
  -->
  [Out(<'READ_RECORD', 'AFL'>),
   Terminal_Sent_ReadRecord($Terminal, PDOL, AIP)]  // 创建新状态事实
```

问题：必须手动管理状态事实（如Terminal_Sent_GPO），参数顺序必须精确匹配，否则会导致协议链断裂

Tamgram Code：

```
process Terminal =
  "Terminal_Sends_GPO":
    [Fr(~UN), !Value($amount, value)]
    --let ... in ... [ .... ]->
    [Out(<"GET_PROCESSING_OPTIONS", PDOL>),
     'PDOL := PDOL, 'Role0 := $Terminal];  // 使用Cell存储状态

  "Terminal_Sends_ReadRecord":
    ['Role0 cas $Terminal,  // 读取Cell并模式匹配
     In(<AIP, "AFL">)]
    -->
    [Out(<"READ_RECORD", "AFL">), 'AIP := AIP];  // 更新Cell
```

改进：Cells（如'PDOL）自动处理状态存储和传递，无需手动定义事实，减少了错误风险。

Part4 实验与评估

	Original	Cell-by-cell	Forward	Backward	Hybrid
executable	55.5	TIMEOUT	91.2	78.3	63.8
bank_accepts	707.7	TIMEOUT	1018.9	1006.7	932.4
auth_to_terminal_minimal	TIMEOUT	TIMEOUT	TIMEOUT	TIMEOUT	TIMEOUT
auth_to_terminal	TIMEOUT	TIMEOUT	TIMEOUT	TIMEOUT	TIMEOUT
auth_to_bank_minimal	60.9	TIMEOUT	98.5	89.7	66.9
auth_to_bank	619.0	TIMEOUT	206.8	203.4	172.1
secrecy_MK	46.9	TIMEOUT	77.6	62.5	49.2
secrecy_privkCard	45.1	TIMEOUT	81.8	74.7	47.4
secrecy_PAN	46.0	TIMEOUT	76.4	68.0	47.9

Figure1: EMVerify Contactless Mastercard CDA NoPIN Low benchmark (seconds)

	Original	Cell-by-cell	Forward	Backward	Hybrid
executable	54.6	TIMEOUT	91.3	77.3	59.4
bank_accepts	1261.0	TIMEOUT	951.6	933.5	872.1
auth_to_terminal_minimal	TIMEOUT	TIMEOUT	978.2	972.4	923.4
auth_to_bank_minimal	82.4	TIMEOUT	154.2	103.5	91.7
auth_to_bank	309.3	TIMEOUT	672.1	383.8	334.2
secrecy_MK	47.2	TIMEOUT	71.8	62.4	47.8
secrecy_privkCard	47.7	TIMEOUT	73.6	69.8	48.4
secrecy_PIN	44.4	TIMEOUT	83.1	76.5	47.0
secrecy_PAN	43.9	TIMEOUT	80.8	69.2	51.2

Figure2: EMVerify Contactless Visa DDA Low benchmark (seconds)

- “we show the practicality of Tamgram with a set of small case studies and one large scale case study.”

- 翻译策略对性能有巨大影响

Cell-by-cell翻译策略表现最差，所有案例均超时（标记为TIMEOUT），无法在合理时间内完成验证；

Forward和Backward翻译策略虽能完成验证，但相比原始手工模型会产生显著的性能开销

- Hybrid策略的有效性

提出的Hybrid翻译策略表现最佳，其验证时间与Original模型最接近；

它成功地将高级语言的便利性与近乎手写代码的性能结合在一起

Part5 总结

(Author's FUTURE WORK)

复杂案例的启发式调优支持

在处理复杂协议时，需要手工调整Tamarin的启发式参数，Tamgram目前缺乏对这些高级启发式配置的语法支持，未来可研究如何在翻译过程中更好地处理这些提示；

自动证明启发式的改进

Tamgram虽然改进了建模体验，但目前并没有自带新的proof heuristics，可研究以Tamgram流程形式更清晰地规范协议控制流是否允许更好地自动推导证明启发式，从而加快验证速度、提高自动化程度；

证明结果的可解释性

Tamarin的证明是基于编译后的模型，Tamgram用户看到的是抽象层级，导致两者之间有“断层”，因此作者也提出研究“back translation”：把Tamarin的证明结果直观地映射回Tamgram层让结果更直观、更容易解读。

Part5 总结

研究启示：

选择分享该文献的重点不在于其具体的技术实现思路，而是其解决问题的新视角

在面对现有工具的局限性时，两种不同的思维方式：

1. “低层”思维： 执着于工具低层原理，试图通过复杂的变通方法来解决问题，这种方法有时虽能勉强完成任务，但会使得系统变得脆弱、难以维护，无法从根本上解决问题；

2. “高层”思维： 作者从协议建模的本质出发，思考“理想的协议建模语言应该是什么样子？”，跳出工具的具体实现细节，引入了模块化、本地内存、以及控制流结构等高级概念，构建了一个新的语言Tamgram，作为工具的“前端”。

一个有效的思路是：当本领域工具存在局限性时，可以在现有框架基础上做补充性创新，尝试从更成熟的相邻领域（如编程语言设计、软件工程）中借鉴经过验证的抽象和范式，来构建更高级的抽象工具，从而降低本领域的认知负荷和工程门槛