Public PhD defense
2020/11/24

Effective Domain-Specific Formal Verification Techniques

Ákos Hajdu

Advisor: Zoltán Micskei, PhD

ftsrg Research Group
Department of Measurement and Information Systems
Budapest University of Technology and Economics

MŰEGYETEM 1782

Hungarian Academy of Sciences

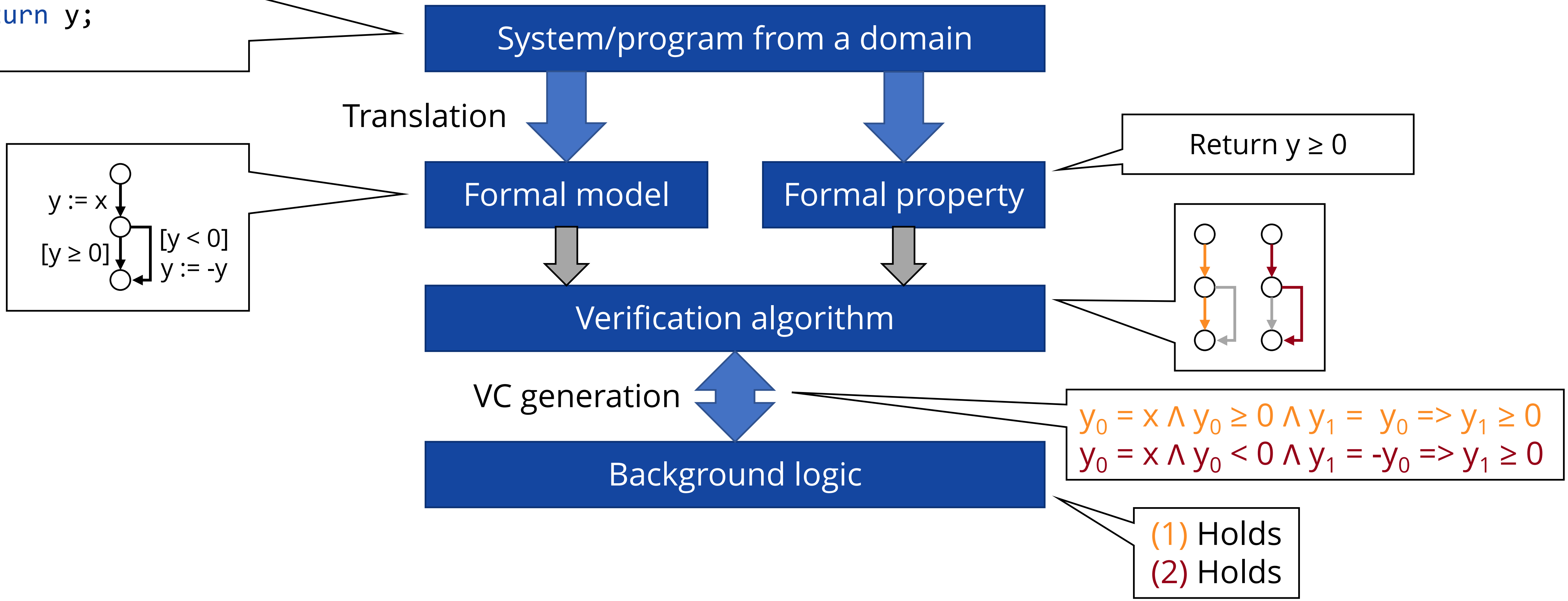
Scope and Motivation

- Critical systems and programs
  - Serious damage
  - Financial consequences

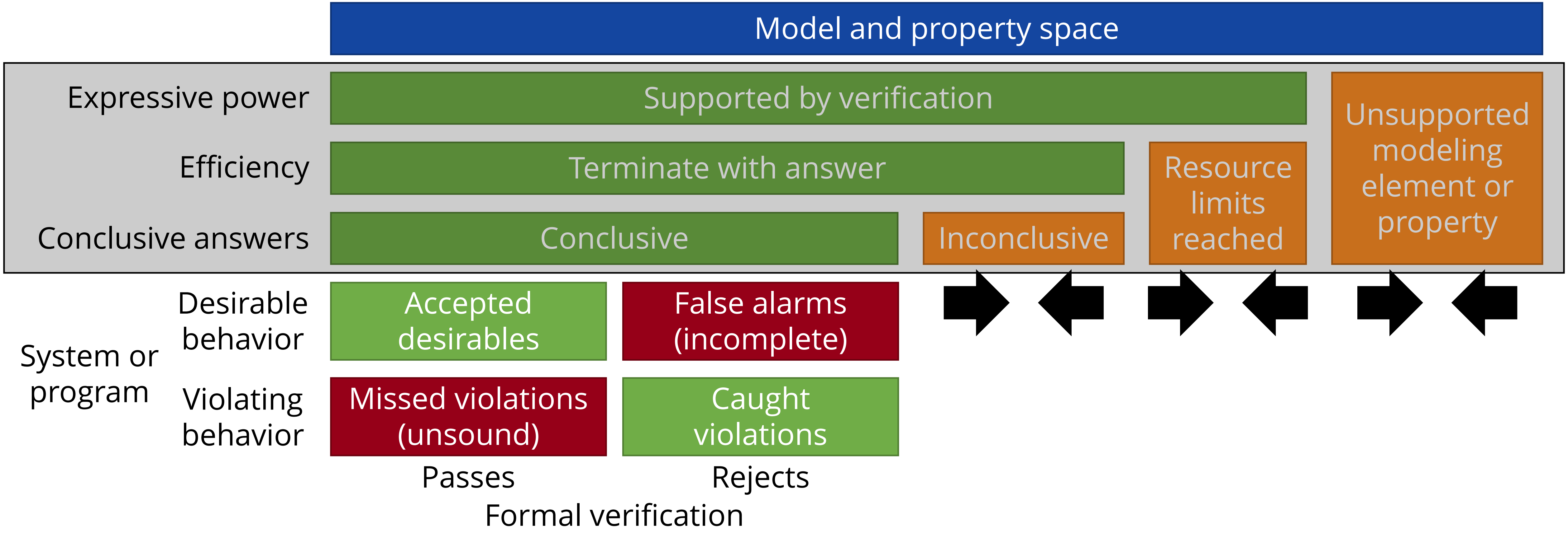
- Formal verification
  - Rigorous reasoning
  - Find errors
  - Prove correctness

Formal Verification

```
int abs(int x) {
 int y = x;
 if (y < 0) y = -y;
 return y;
}
```



Properties and Challenges

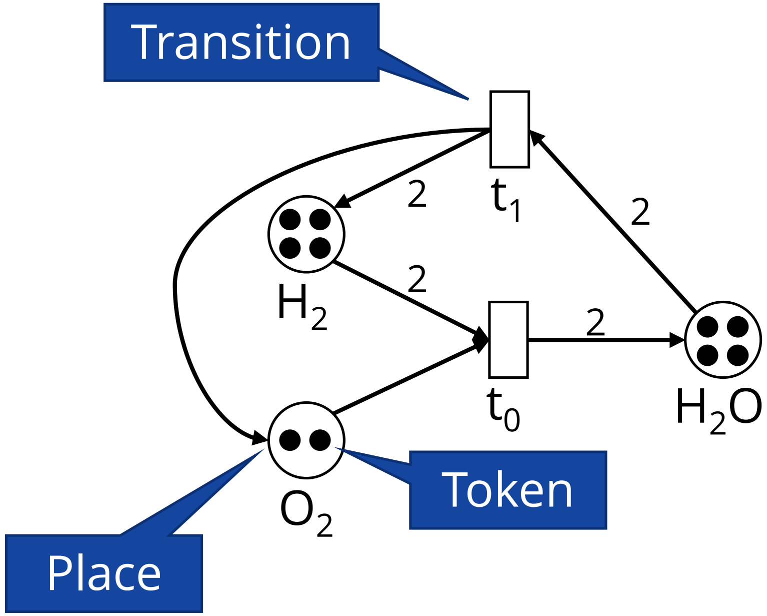


Objective: effective trade-off in practice by balancing the challenges

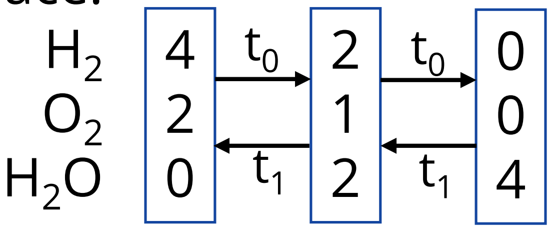
Thesis 1

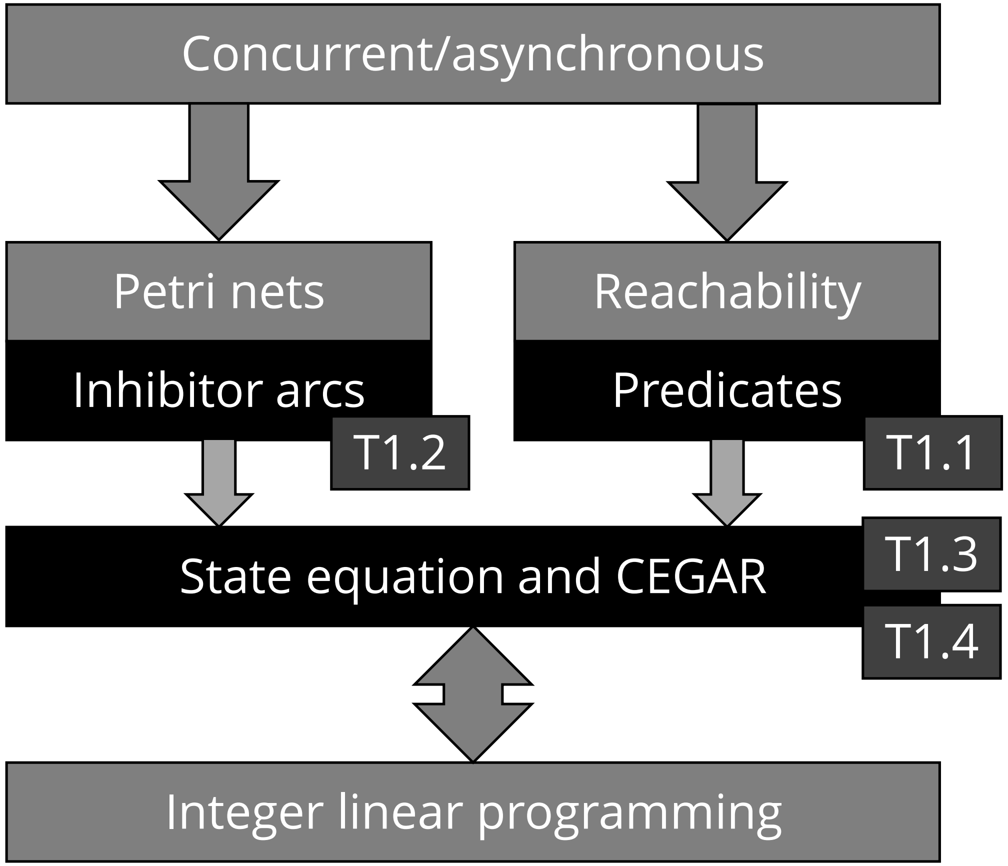
Extensions to the CEGAR Approach on Petri Nets

Background



State space:





CEGAR Approach for Petri Nets

- State equation: structural abstraction for reachability
  - Integer linear programming problem
  - Encodes the acyclic part
  - Necessary but not sufficient criterion

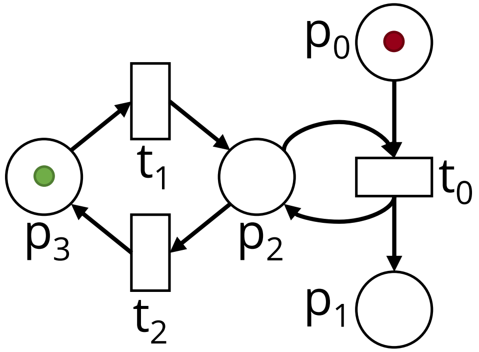
Transitions to be fired

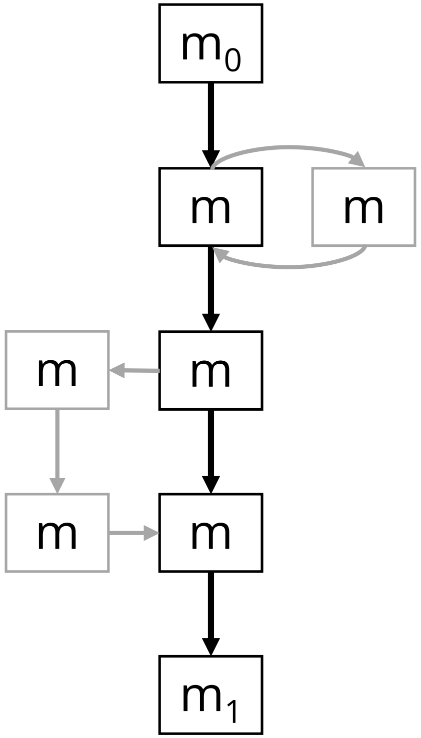
Initial state

$$m_0 + Cx = m_1$$

Target state

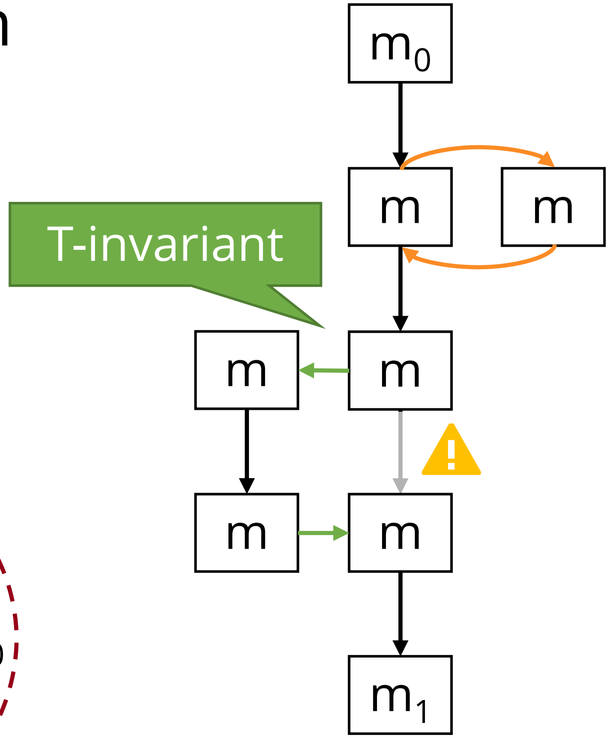
$$\begin{bmatrix} -1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & -1 \\ 0 & -1 & 1 \end{bmatrix}$$





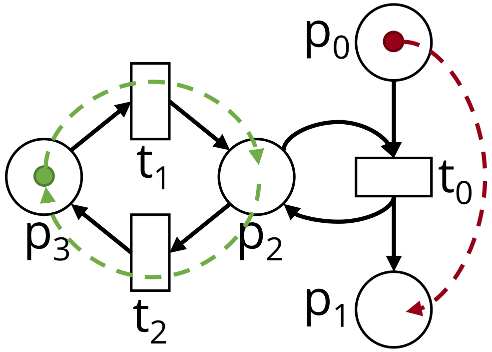
CEGAR Approach for Petri Nets

- Infeasible solution: introduce cyclical behavior
  - Extend equation with constraints: T-invariants
  - Iterative process: Counterexample-Guided Abstraction Refinement (CEGAR)



$$m_0 + Cx = m_1$$

Objective: increase expressive power and conclusive answers



T1.1/T1.2 Improving Expressive Power

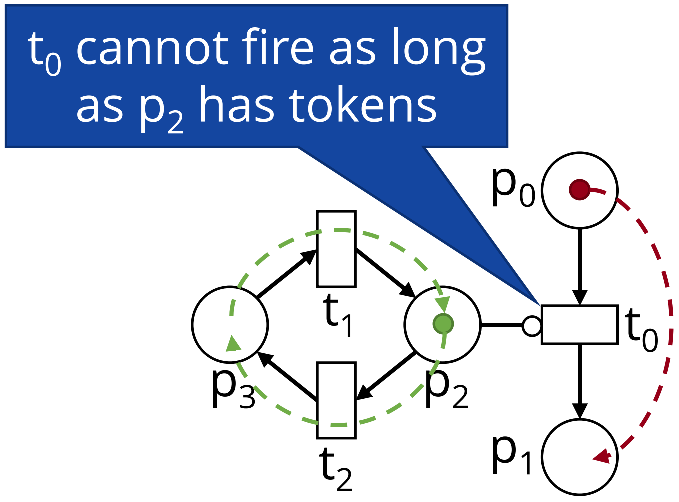
- Reachability of predicates
  - Linear predicate over state to be reached
  - E.g., define target state in one component
  - Transform predicates over places to predicates over transitions

$$Am_1 \geq b$$

$$m_0 + Cx = m_1$$

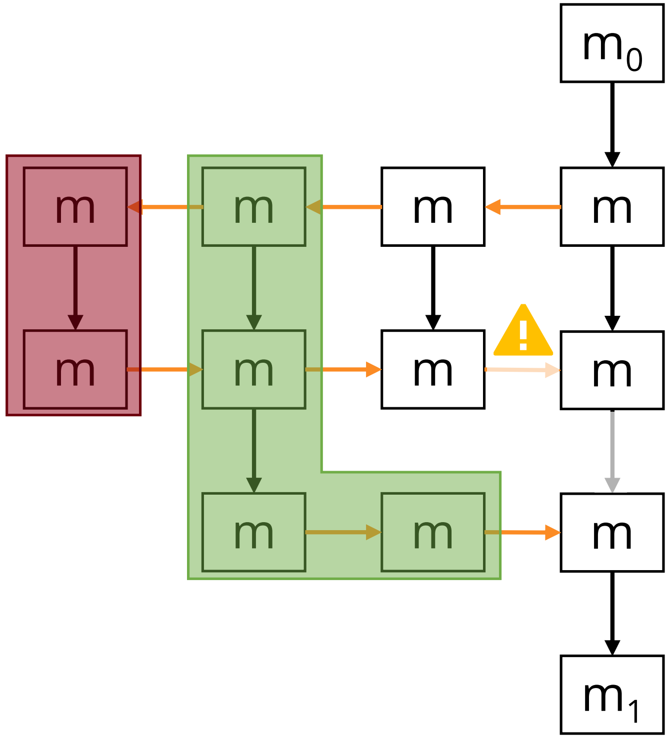
$$(AC)x \geq b - Am_0$$

- Inhibitor arcs
  - Allow testing for emptiness
    - Turing complete expressive power
    - Reachability undecidable
  - Use cycles to “move tokens away”



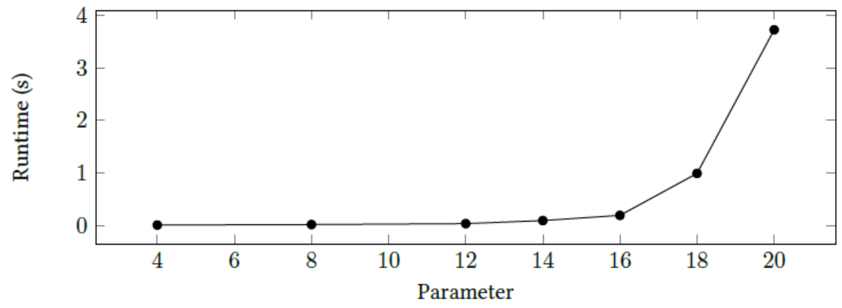
T1.3/T1.4 Increasing Conclusive Answers

- Involving an invariant might not help
  - Algorithm stops with inconclusive answer
- Proposed approach
  - Involve another “distant” (indirect) invariant
  - Proper termination criterion needed
    - Keeping track of refinement progress
- Search strategy
  - Standard: BFS, DFS
  - Hybrid strategy



Evaluation

- Scalability
  - Usually linear scalability w.r.t. marking
  - Often exponential w.r.t. net structure
- Comparison
  - Original algorithm is more efficient, but the extended can answer more problems
  - Complementary to saturation (symbolic)
- Search strategies
  - Hybrid strategy converges faster with less inconclusive results



	DFS		BFS		Hybrid	
Model	Time	Cost	Time	Cost	Time	Cost
Chain 1+2	0.04 s	7	0.055 s	7	0.039 s	7
Chain 1+3	0.095 s	13	0.828 s	13	0.100 s	13
Chain 1+4	0.291 s	21	85.24 s	21	0.288 s	21
Chain 1+4*	24.2 s	35	55.28 s	21	1.498 s	29
Chain 1+5	54.59 s	39	TO	31	56.36 s	39
Chain 2+2	0.076 s	11	0.277 s	11	0.074 s	11
Chain 2+3	0.197 s	19	12.768 s	19	0.288 s	23
Chain 2+3*	2.28 s	29	5.288 s	19	1.387 s	23

Thesis 1 – Summary

I proposed various extensions and improvements to the CEGAR-based reachability analysis of Petri nets, lifting its expressive power and increasing the amount of conclusive answers.

1.1 I generalized the algorithm to be able to solve reachability of predicates, where the target state to be reached can be described with a set of linear constraints.

1.2 I extended the algorithm to be able to handle Petri nets with inhibitor arcs, raising its expressive power.

1.3 I defined the concept of distant invariants and proposed a new iteration strategy, which extended the kind of problems the algorithm could solve.

1.4 I defined a new ordering between partial solutions and a corresponding hybrid search strategy that can speed up the convergence of the algorithm without losing solutions.

Publications: ActaCyb'14, ICATPN'15, SPLST'13, ICATPN'16, SCP'18

Σ Extensions to the efficient CEGAR-based analysis of Petri nets improve expressive power and increase conclusive answers.

Thesis 2

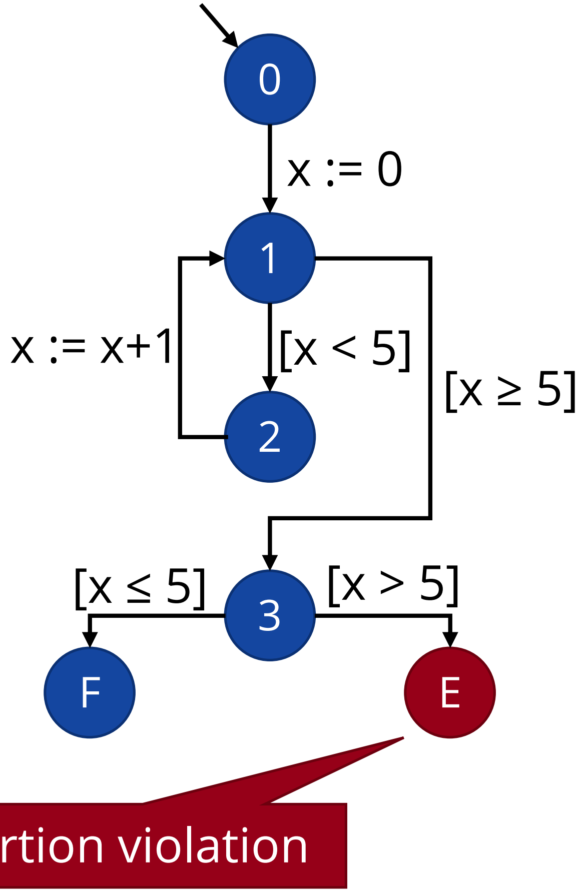
Efficient Strategies for CEGAR-based Software Model Checking

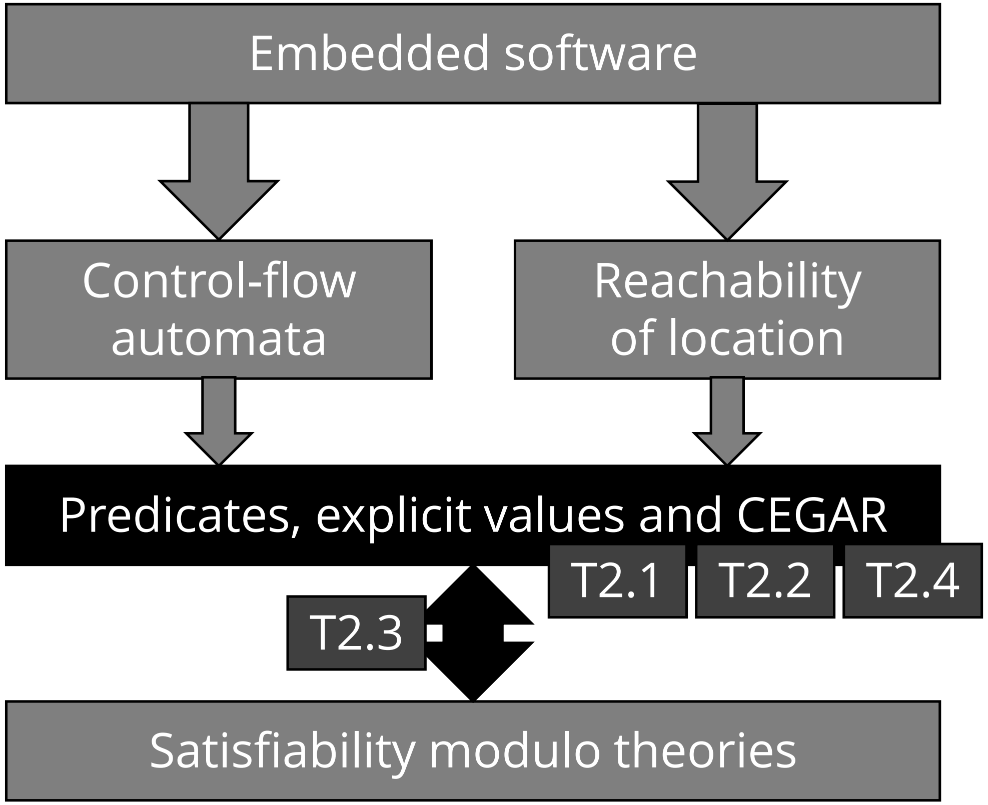
Background

Program

```
   int x;
0: x = 0;
1: while (x < 5) {
2:    x = x + 1;
   }
3: assert (x <= 5);
```

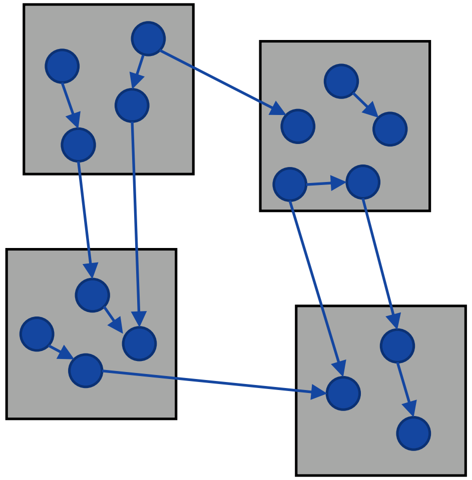
Control-flow automata





Abstract Domains

- Tackle complexity with abstraction
  - Represent states w.r.t. an abstract domain



- Explicit-value abstraction
  - Subset of variables is tracked
  - Others are unknown

```
int x = 0;
for (int i = 0; i < 10000; i++) {
    x = (x + 1) % 10;
}
assert (x < 10);
```

No need to track *i* to prove safety

- Predicate abstraction
  - Track predicates instead concrete values

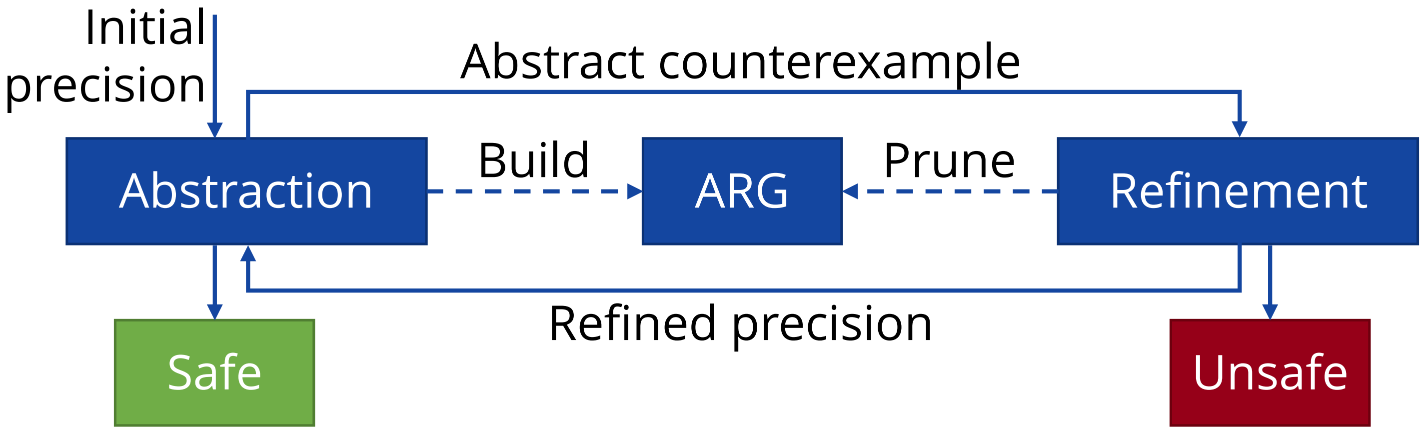
```
int x = 0;
while (x < 1000) {
    x++;
}
assert (x <= 1000);
```

Track $x < 1000$ for loop exit

Track $x = 1000$ for precise exit

CEGAR

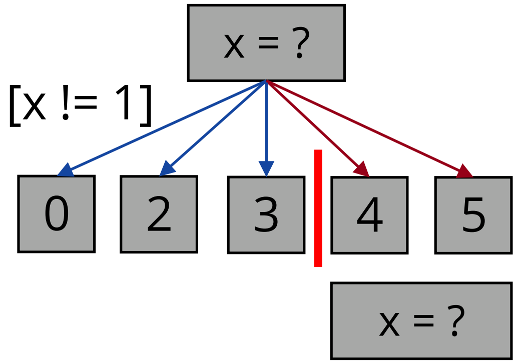
- Counterexample-Guided Abstraction Refinement
  - Iteratively build and refine abstraction
  - ARG: Abstract Reachability Graph (state space)



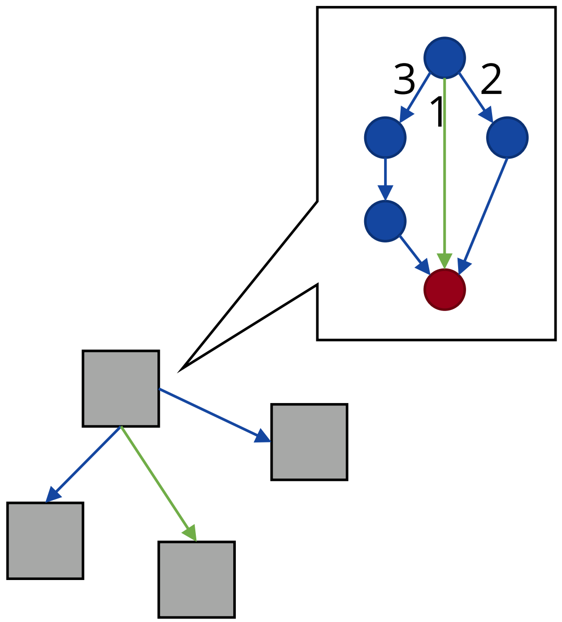
Objective: Make CEGAR more efficient in software model checking

T2.1/T2.2 More Efficient Abstraction

- Configurable explicit domain
  - Unknown values (e.g., input, abstraction)
    - Try enumerating up to limit, propagate unknown above
  - Finer grained control of the level of abstraction

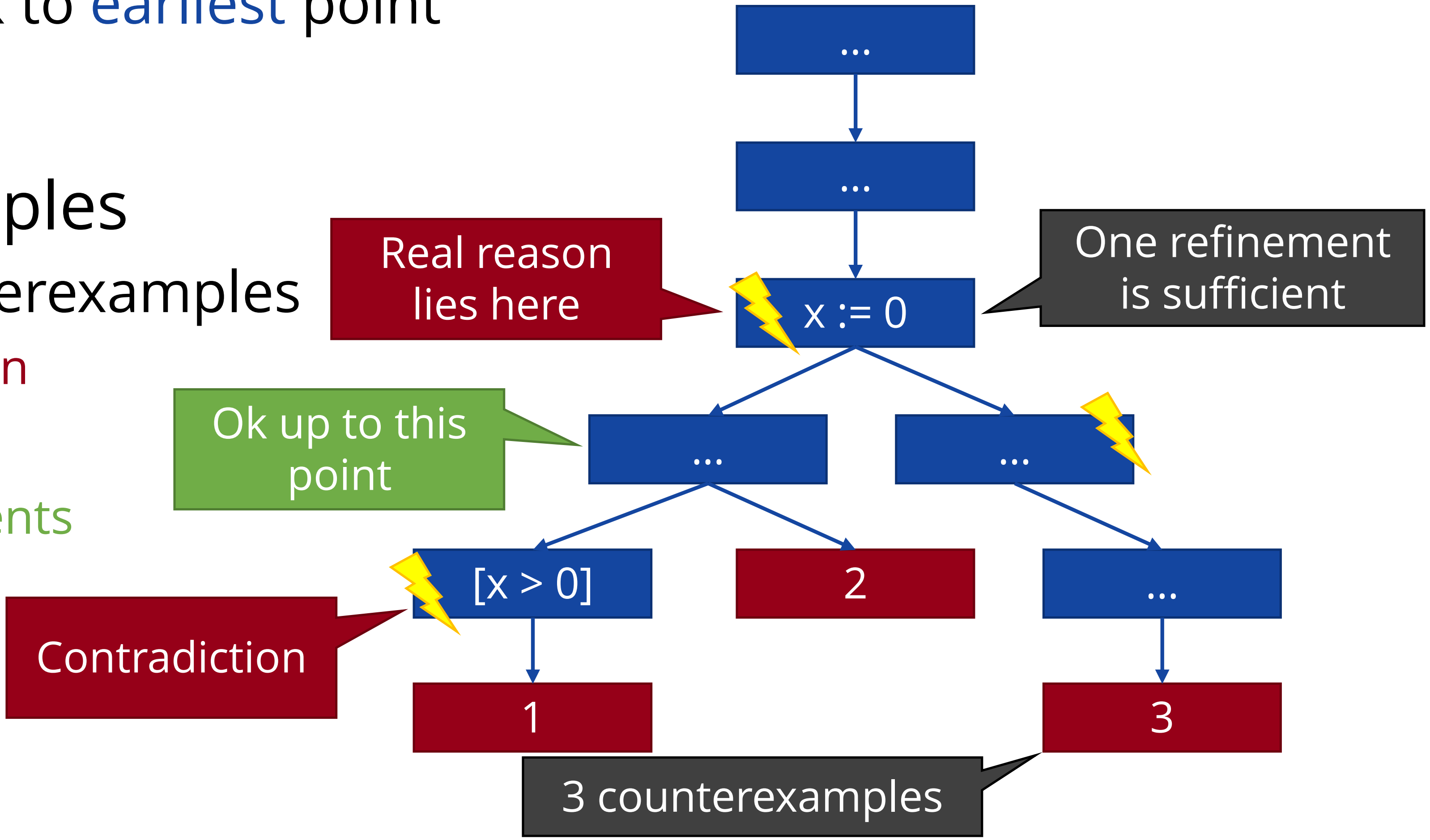


- Error-based search strategy
  - Estimate distance to error using program graph
    - Under-approximation (A* search)
  - Safe programs also have intermediate (abstract) counterexamples



T2.3/T2.4 More Efficient Refinement

- Backward binary interpolation
  - Trace infeasibility back to earliest point
- Multiple counterexamples
  - Collect more/all counterexamples
    - Overhead at abstraction
  - Refine at once
    - Better quality refinements
    - Shared refinements



Evaluation

- Input models
  - 445 C tasks from SV-Comp
  - 90 PLC models from CERN
  - 300 benchmarks from HWMCC
- Results
  - Configurable explicit domain combines advantages of abstraction and enumeration
  - Error-based search improves convergence
  - Backward analysis outperforms forward
  - Multiple counterexamples efficient on complex models

Configuration	143 locks	105 loops	180 eca	17 ssh	90 plc	300 hwmcc	835 TOTAL
PRED_CART_SEQ_ITP	143 192s 55.5Mb	88 300s 196Mb	119 9040s 371Mb	8 111s 209Mb	85 784s 736Mb	54 1390s 3340Mb	497 11800s 3340Mb
PRED_CART_FW_BIN_ITP	4 4.84s 53.8Mb	14 17.5s 53.9Mb	0 0s 0Mb	0 0s 0Mb	0 0s 0Mb	23 316s 644Mb	41 338s 644Mb
PRED_CART_BW_BIN_ITP	143 191s 55.3Mb	90 179s 207Mb	127 7990s 364Mb	11 448s 208Mb	85 521s 439Mb	51 1370s 1760Mb	507 10700s 1760Mb
PRED_BOOL_SEQ_ITP	143 195s 56.8Mb	89 243s 3010Mb	120 8160s 693Mb	8 112s 355Mb	83 968s 2980Mb	52 1410s 3760Mb	495 11100s 3760Mb
PRED_BOOL_FW_BIN_ITP	4 4.83s 53Mb	14 17.5s 54.4Mb	0 0s 0Mb	0 0s 0Mb	0 0s 0Mb	23 312s 659Mb	41 334s 659Mb
PRED_BOOL_BW_BIN_ITP	143 195s 56.5Mb	88 160s 366Mb	121 9510s 2450Mb	11 462s 414Mb	79 961s 3880Mb	48 1190s 3460Mb	490 12500s 3880Mb
EXPL_SEQ_ITP	143 196s 56.8Mb	22 87.4s 324Mb	133 4100s 2250Mb	14 239s 637Mb	65 626s 2990Mb	48 1780s 3150Mb	425 7030s 3150Mb
EXPL_FW_BIN_ITP	4 5.02s 54.5Mb	13 51.5s 263Mb	0 0s 0Mb	0 0s 0Mb	0 0s 0Mb	45 1170s 2550Mb	62 1230s 2550Mb
EXPL_BW_BIN_ITP	143 194s 56.6Mb	20 100s 330Mb	80 2110s 2510Mb	12 154s 639Mb	70 496s 2950Mb	50 1930s 3320Mb	375 4980s 3320Mb
virtual best	143 190s 55Mb	96 233s 1310Mb	140 4660s 2230Mb	16 216s 637Mb	86 408s 736Mb	65 2170s 3340Mb	546 7880s 3340Mb

Category

Thesis 2– Summary

I proposed various improvements and strategies to CEGAR-based software model checking, increasing the efficiency of the algorithm.

2.1 I generalized explicit-value analysis to be able to enumerate a predefined, configurable number of successor states, improving its precision, but avoiding state space explosion.

2.2 I adapted a search strategy to the context of CEGAR that estimates the distance from the erroneous state in the abstract state space based on the structure of the software, efficiently guiding exploration towards counterexamples.

2.3 I introduced an interpolation strategy based on backward reachability, that traces back the reason of infeasibility to the earliest point in the program, yielding faster convergence.

2.4 I described an approach for refinement based on multiple counterexamples, which allows exchanging information between counterexamples and provides better refinements.

Publications: JAR'19, FORTE'16, VPT'17, FMCAD'17, MiniSym'17, MiniSym'18 ! OpenMBEE'20 !

∑ Efficient, CEGAR-based strategies help software model checking scale to industrial use cases.

Thesis 3

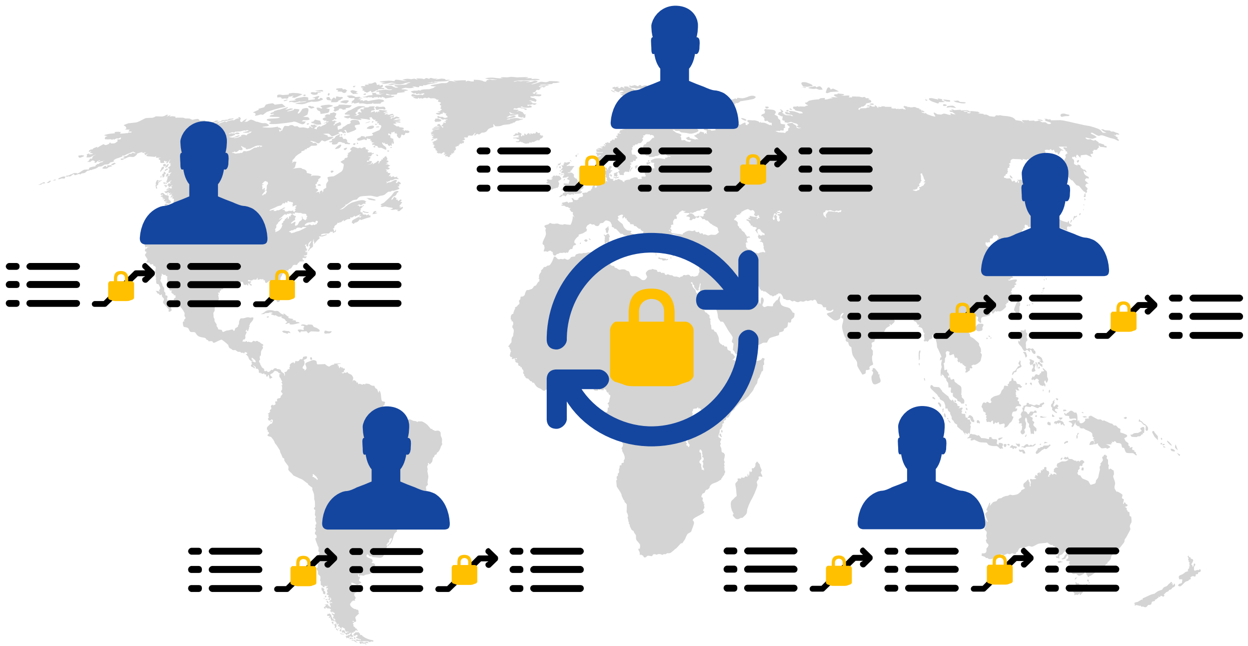
Modular Specification and Verification of Smart Contracts

The author was also affiliated with SRI International during the work described in this thesis.

Background

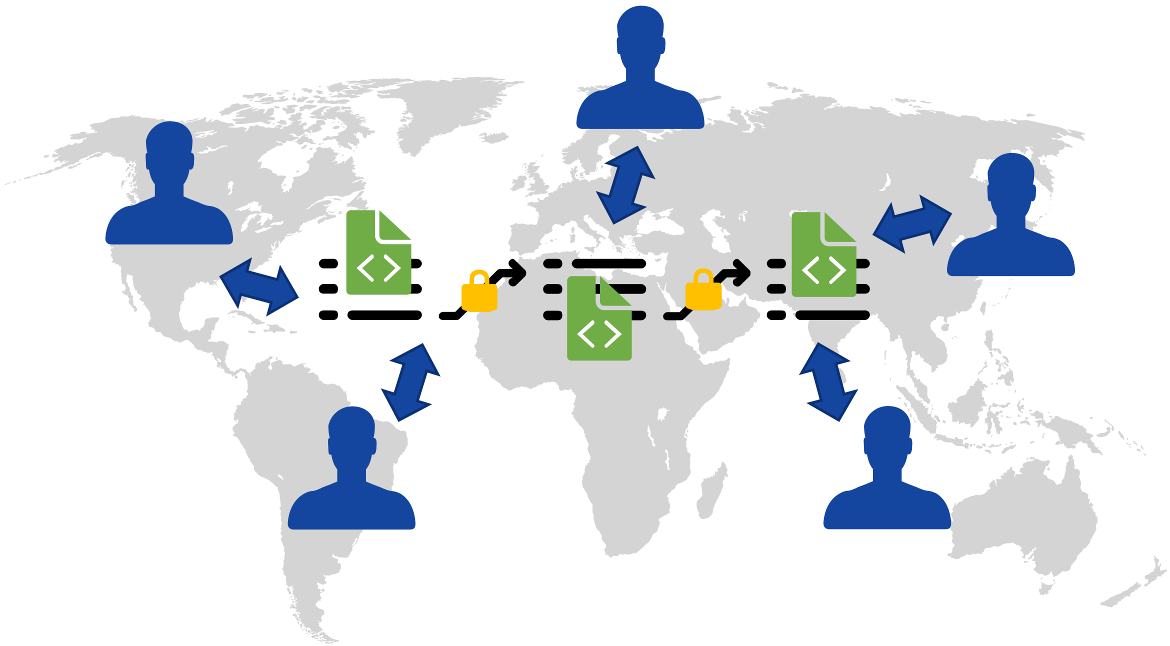
- The blockchain 
- Distributed ledgers

Decentralized/blockchain



Background

- Conceptually a single global state
- Distributed computing platforms
  - Executable code on ledger



Decentralized/blockchain

Background

- Smart contracts (Solidity)

State variable

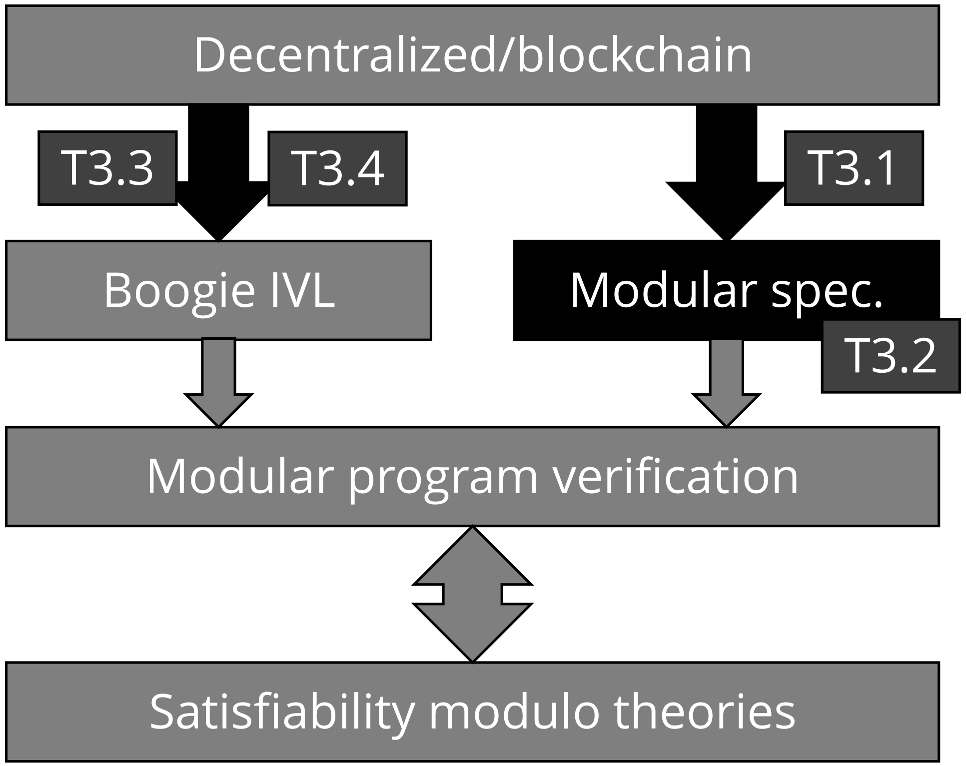
Function

Function

```
contract SimpleBank {
  mapping(address=>uint) balances;

  function deposit() public payable {
    balances[msg.sender] += msg.value;
  }

  function withdraw(uint amount) public {
    require(balances[msg.sender] >= amount);
    balances[msg.sender] -= amount;
    msg.sender.transfer(amount);
  }
}
```



Objective: Check high level, functional properties efficiently

T3.1/T3.2 Annotations

- Adapt modular properties
  - Contract level invariants
  - Pre/postconditions
  - Loop invariants

- Domain specific extensions
  - Balances, transactions, …
  - Sum over collections

```
/// invariant sum(balances) == this.balance
contract SimpleBank {
  mapping(address=>uint) balances;

  function deposit() public payable {
    balances[msg.sender] += msg.value;
  }

  function withdraw(uint amount) public {
    require(balances[msg.sender] >= amount);
    balances[msg.sender] -= amount;
    msg.sender.transfer(amount);
  }
}
```

T3.3/T3.4 Encoding to Boogie IVL

- Smart contracts → Boogie
  - SMT-based intermediate verification language

- Similar to program verification, but much more in the details
  - Balances, payments
  - Message passing
  - Transactional behavior
  - Large bit-widths (256)

Solidity

```
uint8 x = 255;
uint8 y = 1;
x + y == 0;
```

8-256 bits, Wraparound overflow

Boogie encoding

```
int x = 255;
int y = 1;
x + y == 256;
```

SMT integers
Scalable
Not precise

```
bv8 x = 255bv8;
bv8 y = 1bv8;
x + y == 0bv8;
```

SMT bitvectors
Precise
Not scalable

```
int x = 255;
int y = 1;
(x + y) % 256 == 0;
```

Modulo
Precise
Scalable

Evaluation

- Unannotated contracts
  - Implicit specification
    - require, assert, overflows
  - Mostly false alarms due to wrong usage
  - Found some overflow issues
- Annotated contracts
  - High level, functional properties
  - Detect, fix and prove real issues
    - Token overflow, reentrancy
  - Modular arithmetic is efficient
    - 256 bits, nonlinear properties

```
/// @notice invariant sum(balances) == totalSupply
contract BecToken {
  using SafeMath for uint256;

  uint256 totalSupply;
  mapping(address => uint256) balances;

  function batchTransfer(address[] _recvs, uint256 _value) {
    uint cnt = _recvs.length;
    uint256 amount = uint256(cnt) * _value;
    require(cnt > 0 && cnt <= 20);
    require(_value > 0 && balances[msg.sender] >= amount);

    balances[msg.sender] = balances[msg.sender].sub(amount);
    /// @notice invariant totalSupply ==
                          sum(balances) + (cnt - i) * _value

    /// @notice invariant i <= cnt
    for (uint i = 0; i < cnt; i++)
      balances[_recvs[i]] =
              balances[_recvs[i]].add(_value);
  }
}
```

Thesis 3 – Summary

I defined a modular specification and verification approach for smart contracts by annotating and translating them to an intermediate verification language.

3.1 I adapted existing modular specification constructs to the context of smart contracts.

3.2 I proposed domain-specific annotations for the modular specification and verification of smart contracts.

3.3 I introduced a mapping from the Solidity contract-oriented programming language to the Boogie intermediate verification language.

3.4 I described a modular arithmetic encoding that supports scalable bit-precise reasoning on arithmetic operations.

Publications: VSTTE'19, ESOP'20 ! FMBC'20, IEEE Access'20 !

Σ Modular specification and verification can check high level, functional properties of smart contracts efficiently.

Summary

Publications

Related to theses

	Thesis 1	Thesis 2	Thesis 3
Journal	**ActaCyb** SCP	**JAR**	
Conference	SPLST 2x **ICATPN**	FORTE VPT FMCAD	**VSTTE** ESOP
Local event		2x BME	

Highlights

23 publications

- 4 **journal** (incl. JAR, SCP)
- 8 **conference** (incl. ICATPN, FMCAD, ESOP)
- 5 **workshop** (incl. VPT, FESCA)
- 4 **local** event (BME)
- 1 **technical** report (CERN)
- *1* PGP *US patent*

100+ **citations** based on Google Scholar

40+ **independent** peer-reviewed

ftsrg

Applications

- Free and open source tools

petridotnet
petridotnet.inf.mit.bme.hu/en

theta
github.com/FTSRG/theta

solcverify
github.com/SRI-CSL/solidity

- Case studies

Education

Modeling and analysis of public transport

Formal verification of PLC codes

Analysis of public smart contracts

Fault injection

- Talks

Conclusions

Thesis 1

- Concurrent/asynchronous
  - Petri nets / Inhibitor arcs
  - Reachability / Predicates
- State equation and CEGAR
- Integer linear programming

Thesis 2

- Embedded software
  - Control-flow automata
  - Reachability of location
- Predicates, expl. vals. and CEGAR
- Satisfiability modulo theories

Thesis 3

- Decentralized/blockchain
  - Boogie IVL
  - Modular specification
- Modular program verification
- Satisfiability modulo theories

- Improve expressive power and conclusive answers
- Improve practical scalability
- Check high-level, functional properties efficiently

Ongoing work after the dissertation:

- Gazer: LLVM-based C frontend & SV-Comp
- Gamma: statechart frontend

- Spec and verif. of events
- Fault injection
- Upgrade Solidity version