Intersection and Rotation of Assumption Literals Boosts Bug-Finding

Rohit Dureja*, Jianwen Li*, Geguang Pu#,

Moshe Y. Vardi+, Kristin Y. Rozier*

*Iowa State University, Ames, USA

#East China Normal University, Shanghai, China

+Rice University, Houston, USA

Verified Software: Theories, Tools, and Experiments (VSTTE)

New York, NY
July 16, 2018

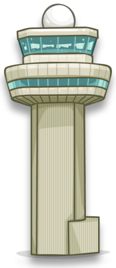
Design-Space Exploration

Design-Space Exploration

What is a design-space?

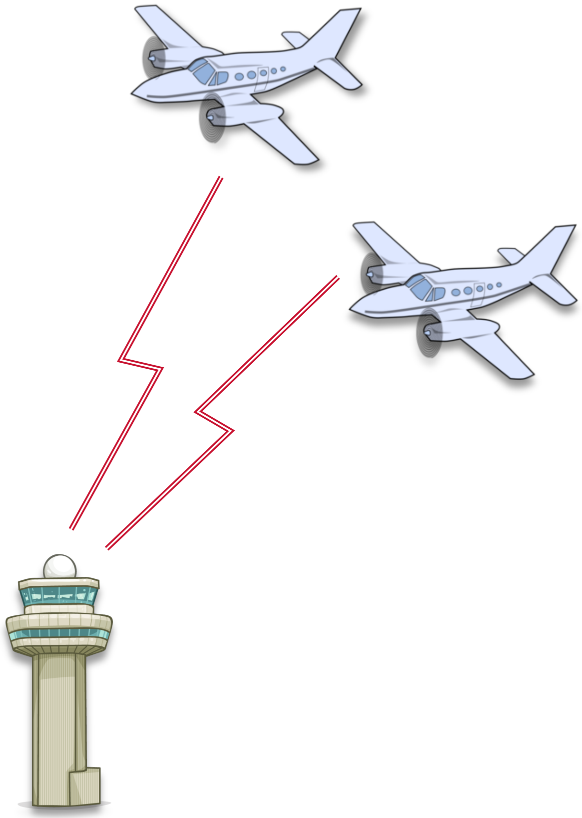
Design Space

Airspace Allocation



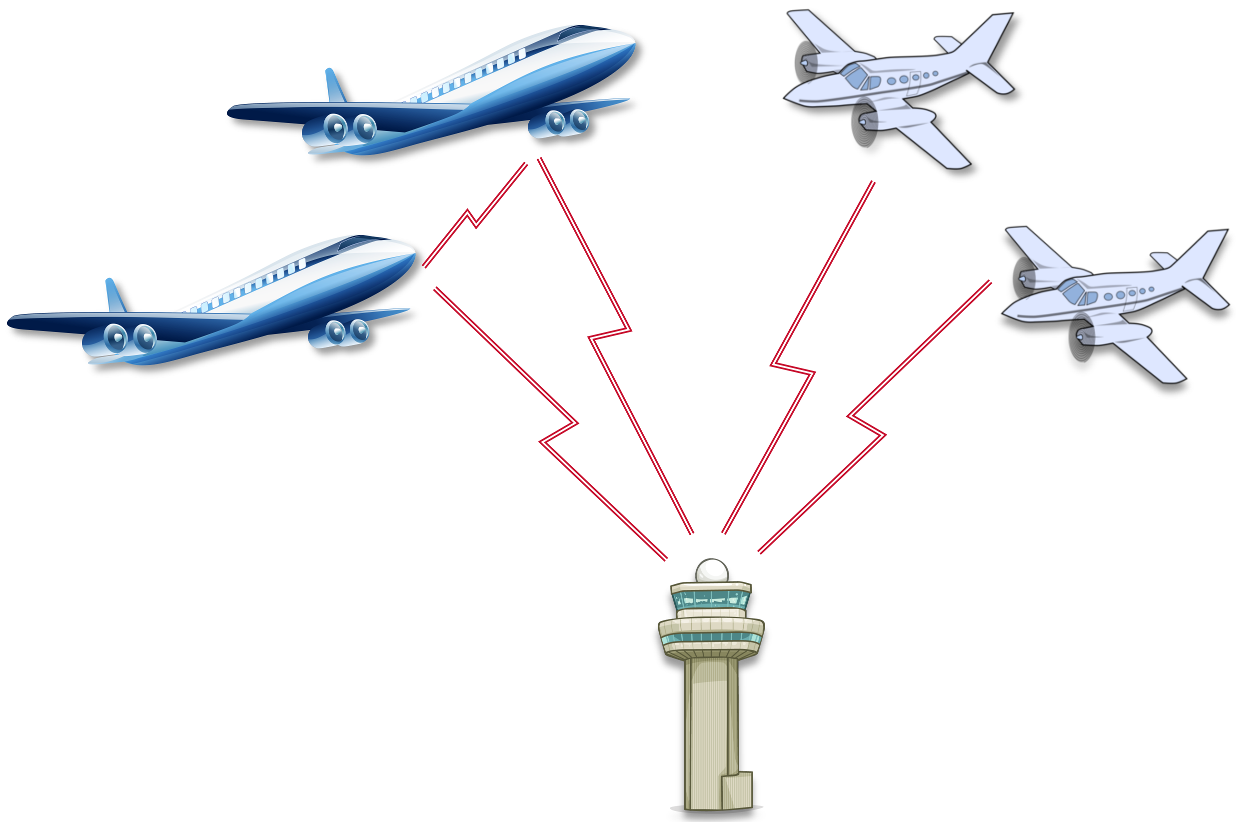
Design Space

Airspace Allocation



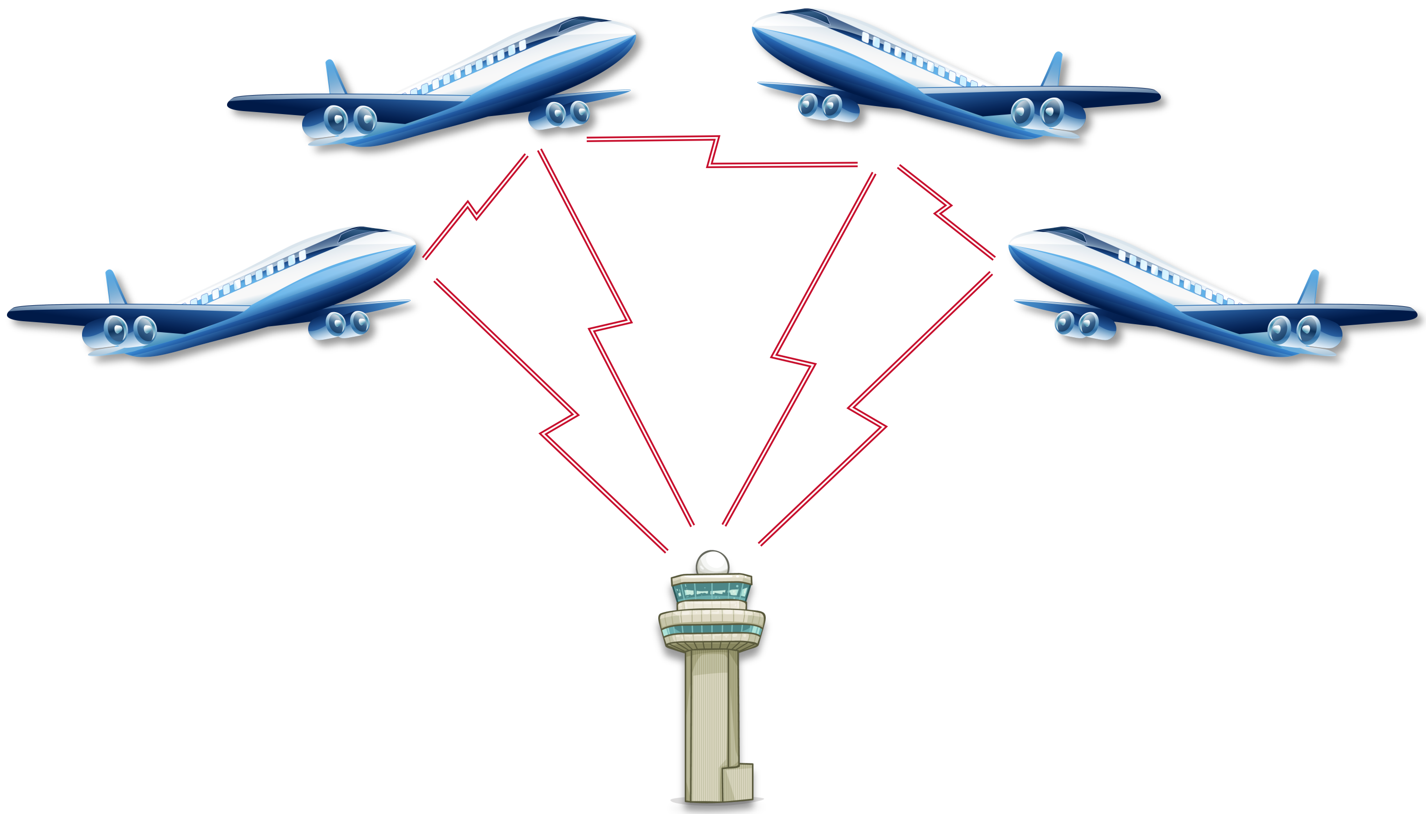
Design Space

Airspace Allocation



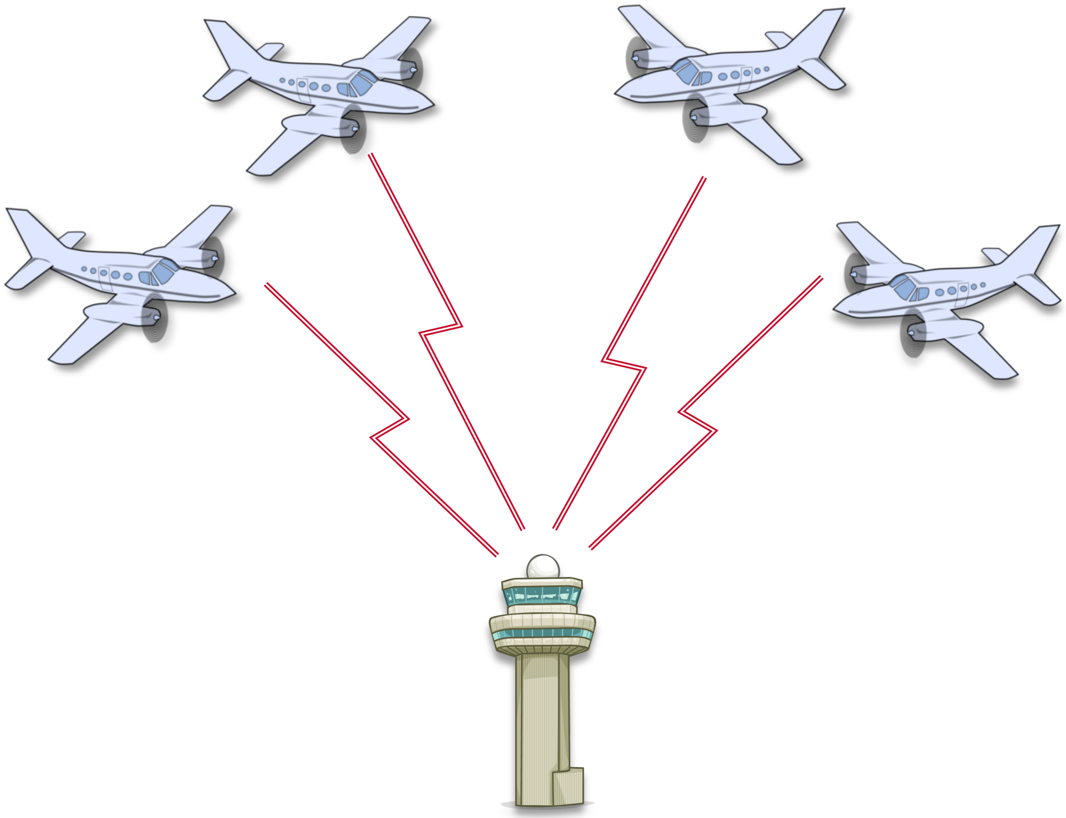
Design Space

Airspace Allocation



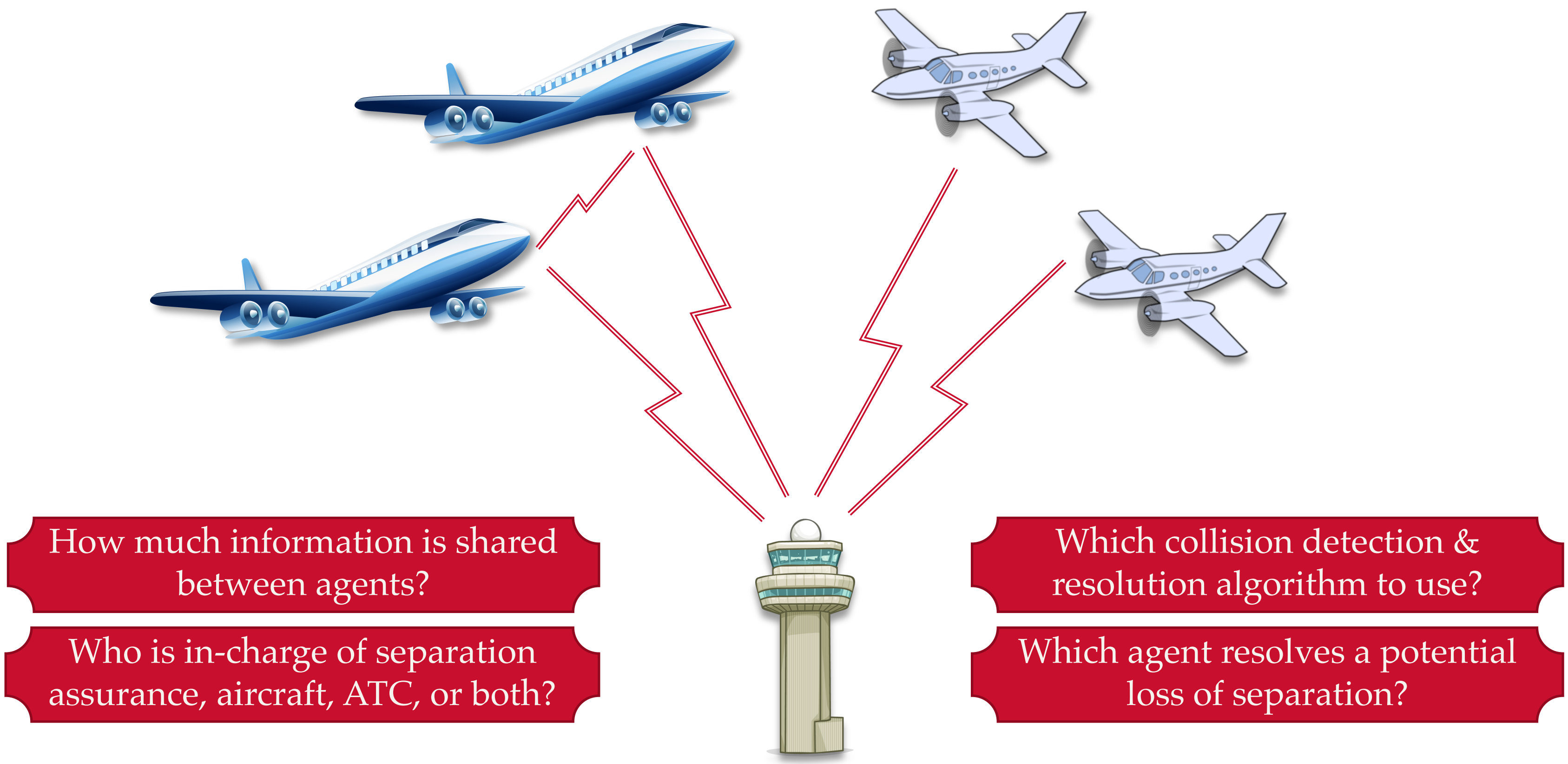
Design Space

Airspace Allocation



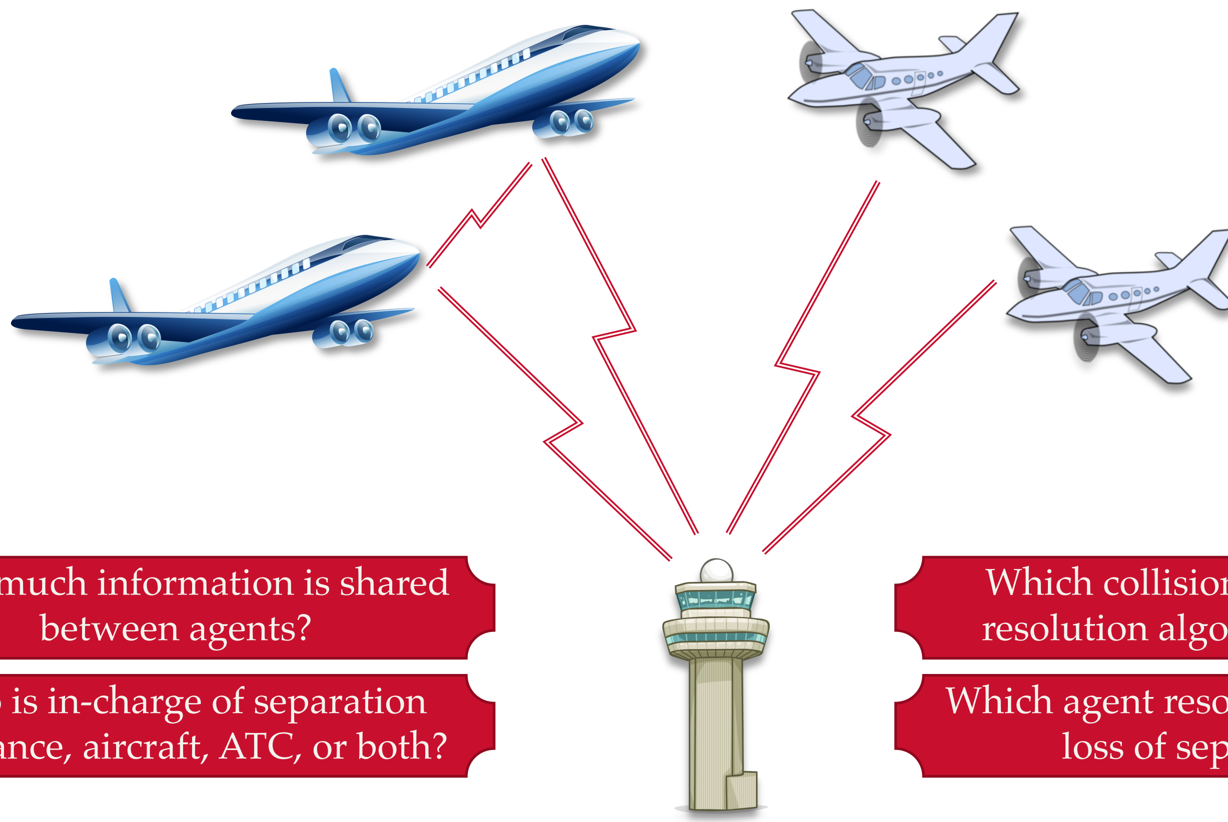
Design Space

Airspace Allocation



Design Space

Airspace Allocation



How much information is shared between agents?

Who is in-charge of separation assurance, aircraft, ATC, or both?

Which collision detection & resolution algorithm to use?

Which agent resolves a potential loss of separation?

Lots of Design Choices!

Design-Space Exploration

What is a design-space?

What is a design-space?

Set of possible design choices for a system.

Design-Space Exploration

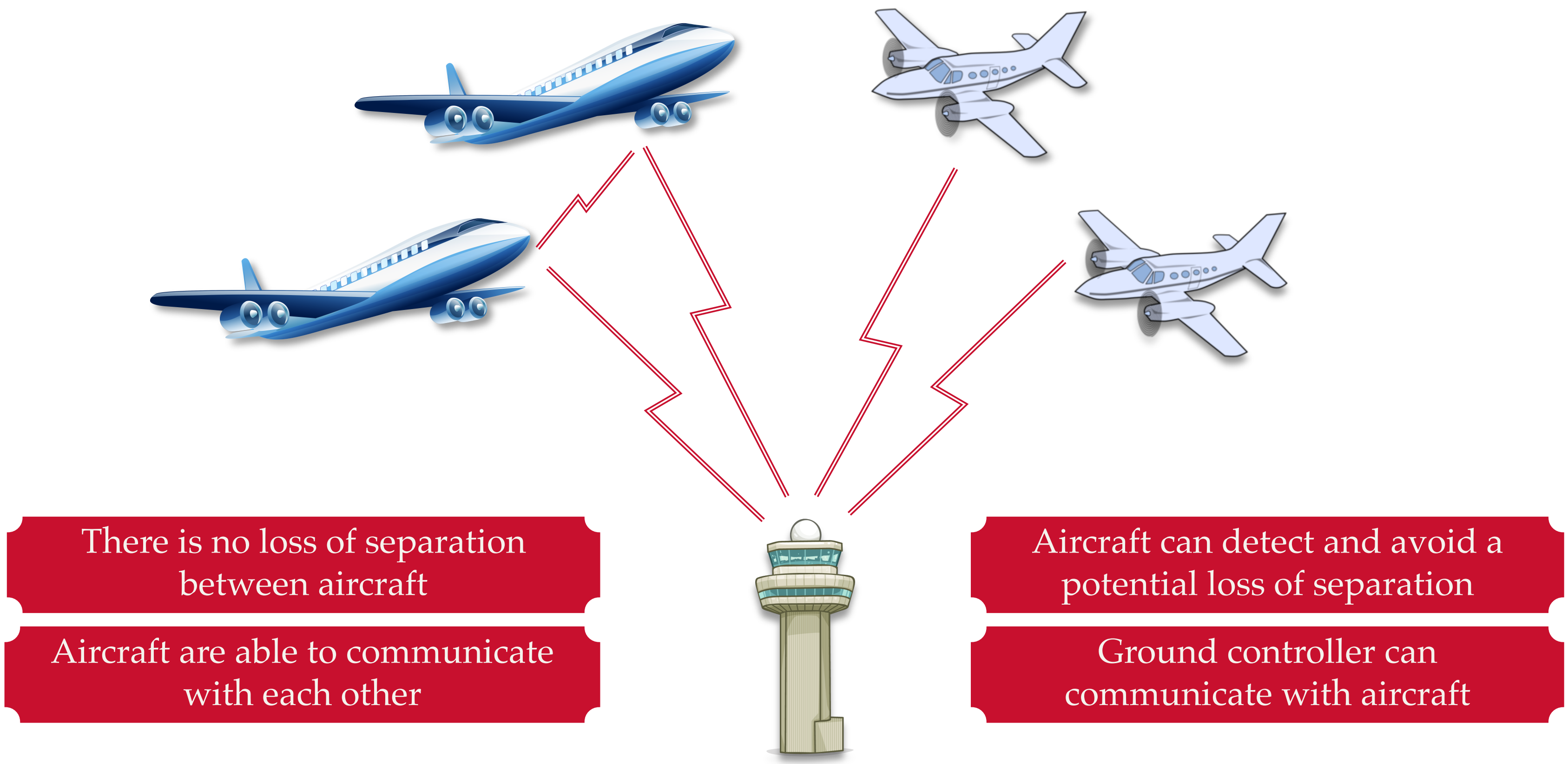
What is a design-space?

Set of possible design choices for a system.

What is a design-space exploration?

Design-Space Exploration

Airspace Allocation



Find design choices that satisfy requirements

Design-Space Exploration

What is a design space?

Set of possible design choices for a system.

What is a design-space exploration?

Design-Space Exploration

What is a design space?

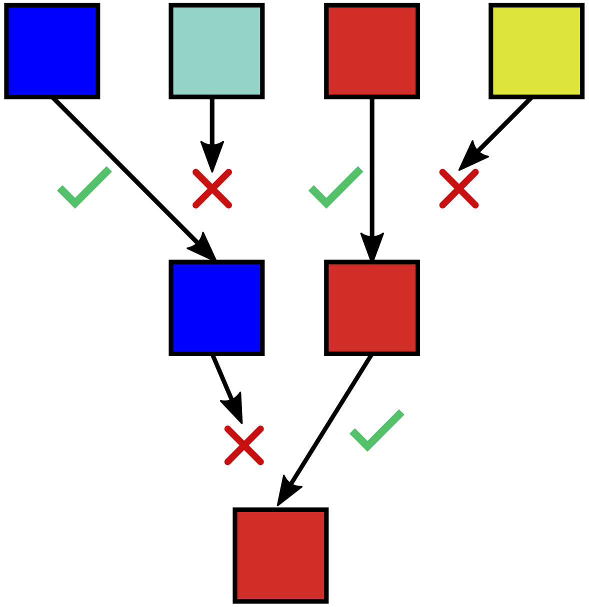
Set of possible design choices for a system.

What is a design-space exploration?

Design-time analysis to evaluate design choices exhaustively.

Design-Space Exploration

Complex systems are modeled as design spaces.

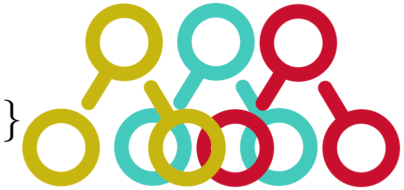


Alternative comparison via design space exploration

Model Checking!

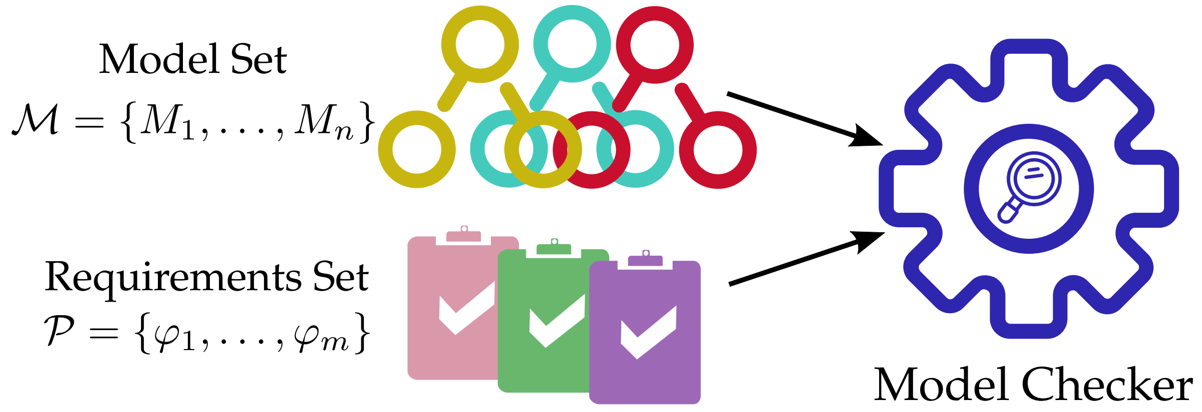
Model Checking Design Spaces

Model Set
$\mathcal{M} = \{M_1, \ldots, M_n\}$

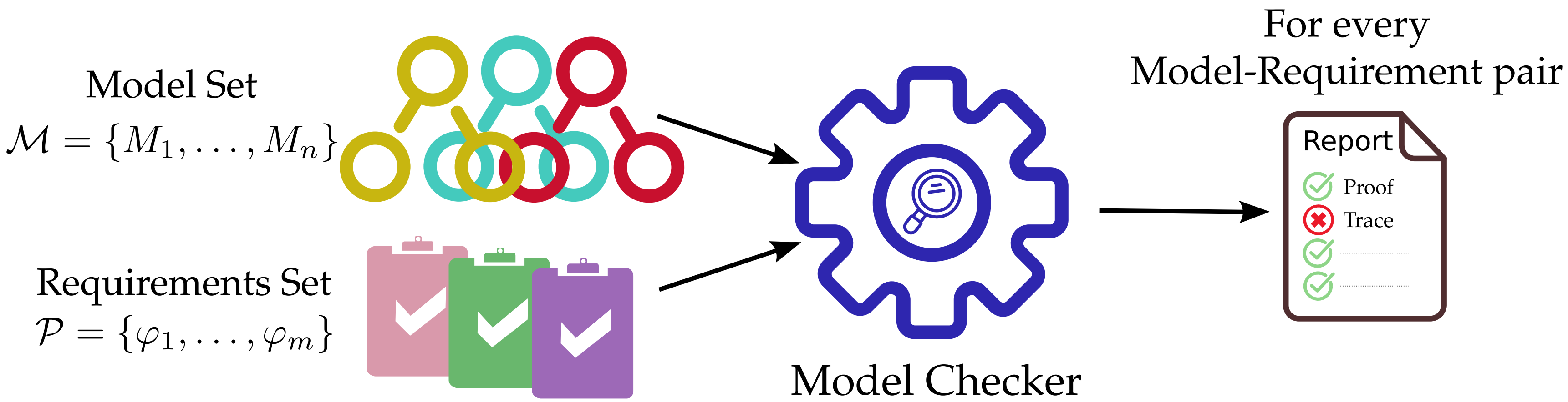


Requirements Set
$\mathcal{P} = \{\varphi_1, \ldots, \varphi_m\}$

Model Checking Design Spaces



Model Checking Design Spaces



Design-space model checking entails
multi-model/requirement checking

Our Goal
Make model-checking for design-spaces more scalable

Multi-model/requirement Checking

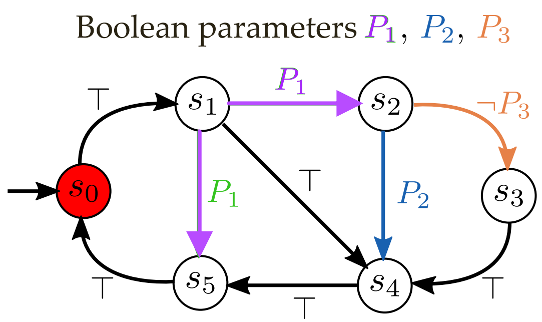
Design-space reduction	Incremental Verification	Improved Orchestration	Model-checking algorithms

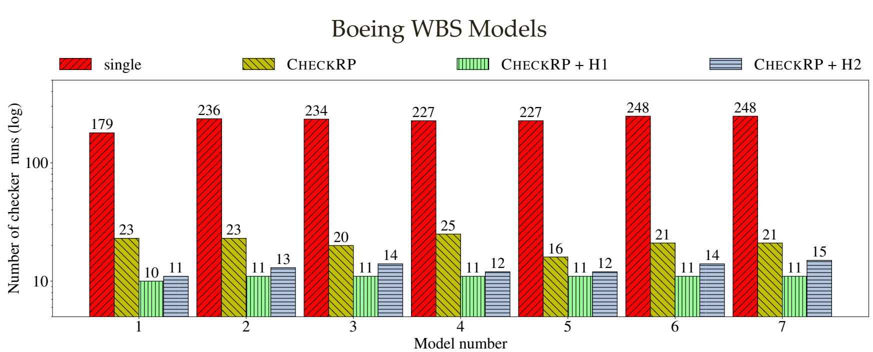
Multi-model/requirement Checking

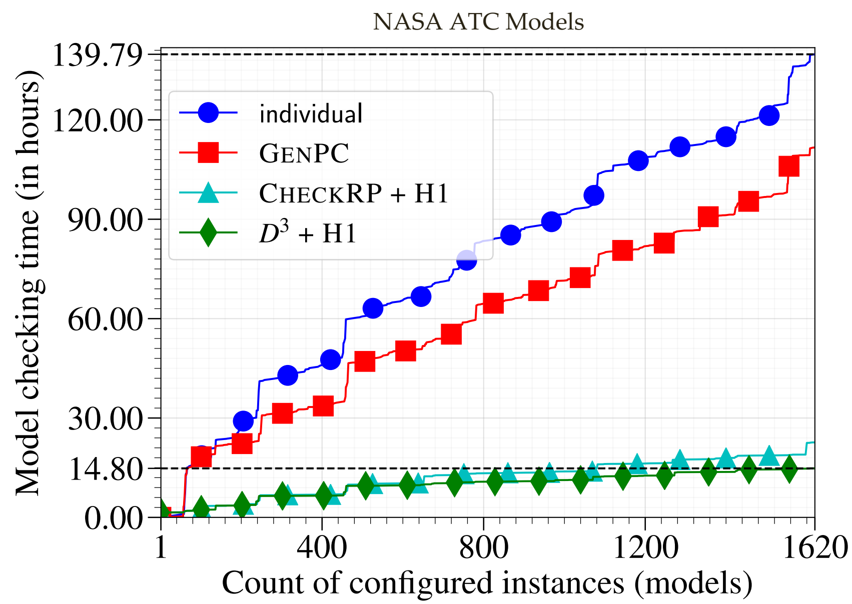
Design-space reduction	Incremental Verification	Improved Orchestration	Model-checking algorithms

Design-space Reduction[1]

- Generate design-space models from a meta-model
  - Combinatorial transitions systems (CTS), behavior enabled by parameters
- D^3 algorithm to reduce number of model-property pairs
  1. Finding redundant models, or models with exact same behavior (GenPC)
  2. Reducing number of requirements by finding logical dependencies (CheckRP)







Upto 9.0x speedup

[1] R. Dureja and K. Y. Rozier. "More Scalable LTL Model Checking via Discovering Design-Space Dependencies" (TACAS 2017)

Multi-model/requirement Checking

Design-space reduction	Incremental Verification	Improved Orchestration	Model-checking algorithms

Multi-model/requirement Checking

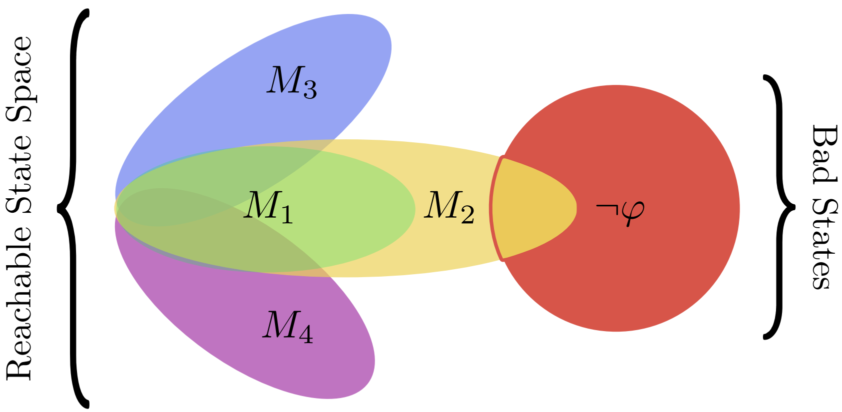
Design-space reduction	Incremental Verification	Improved Orchestration	Model-checking algorithms

Incremental Verification[2,3]

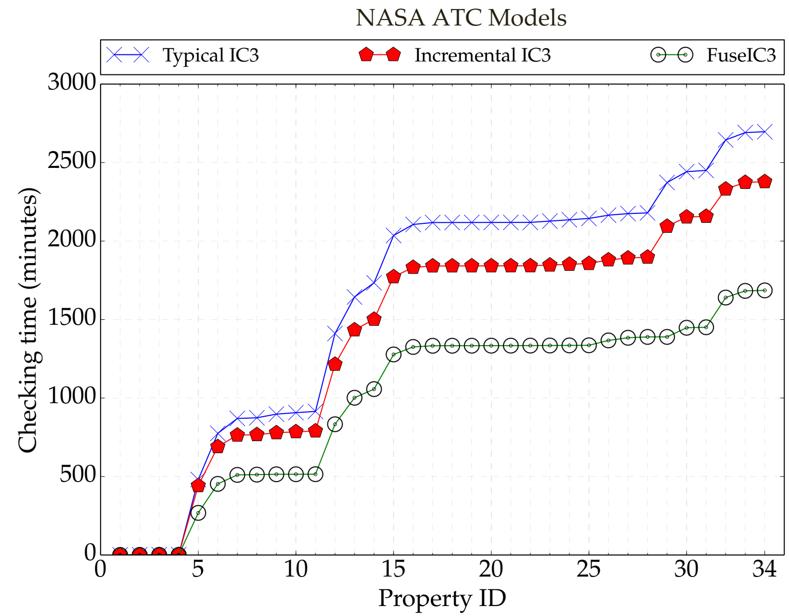
- The different design-space models have overlapping state spaces
  - Generated from the same meta-model, overlapping behavior
- FuseIC3 algorithm algorithm reuses reachable state approximations
  1. IC3 frames are stored and "repaired" across multiple model-checking runs [2]
  2. Very fast verification when model-delta is small, regressions runs [3]

Set of related models $\{M_1, M_2, M_3, M_4\}$

Safety property φ



1. Check M_1 with $\varphi \longrightarrow$ $M_1 \models \varphi$
2. Check M_2 with $\varphi \longrightarrow$ $M_2 \not\models \varphi$



Upto 5.48x speedup

[2] R. Dureja and K. Y. Rozier. "FuseIC3: An Algorithm for Checking Large Design Spaces" (FMCAD 2017)

[3] R. Dureja and K. Y. Rozier. "Incremental Design-Space Model Checking via Reusable Reachable State Approximations." (under submission)

Multi-model/requirement Checking

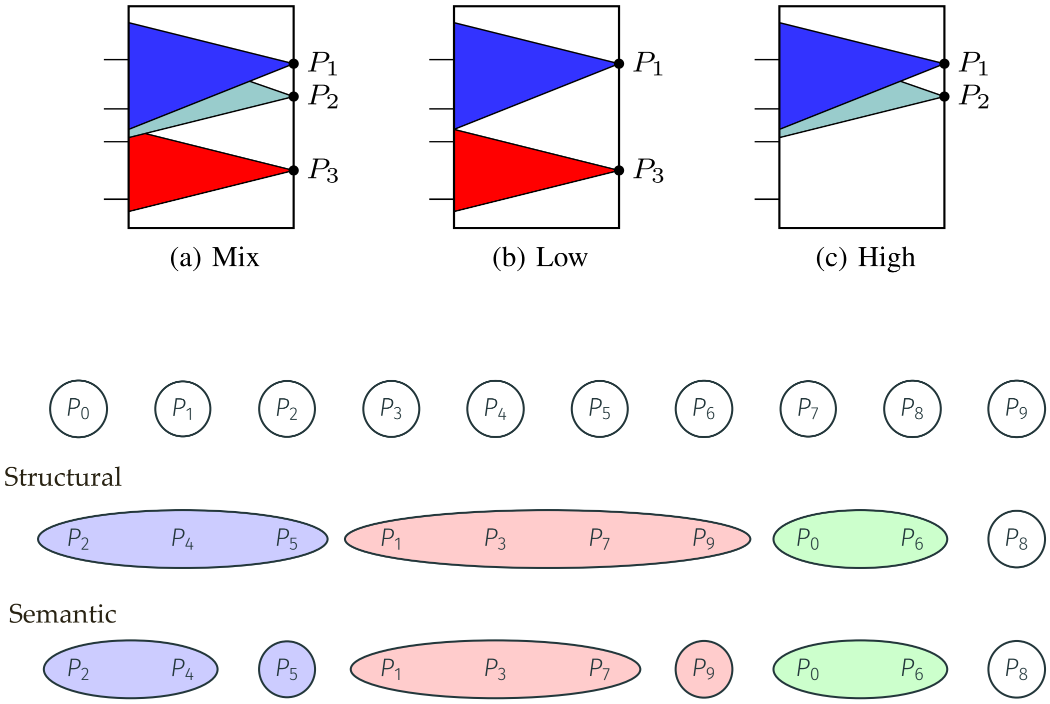
Design-space reduction	Incremental Verification	Improved Orchestration	Model-checking algorithms

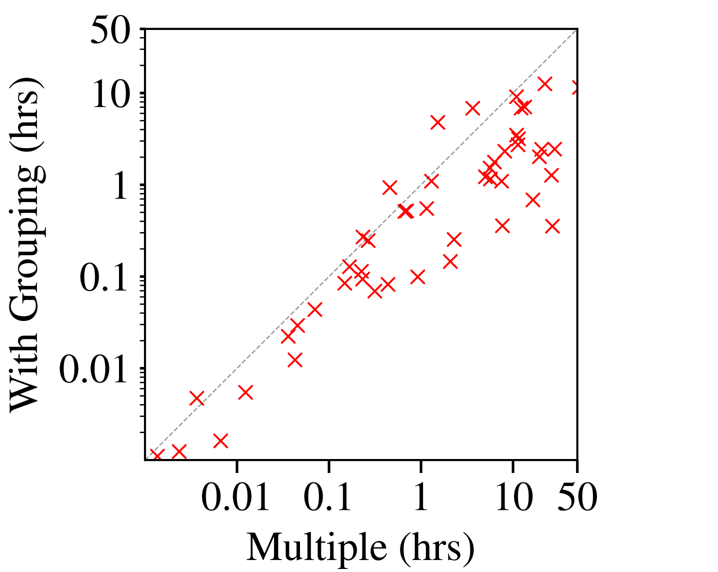
Multi-model/requirement Checking

Design-space reduction	Incremental Verification	Improved Orchestration	Model-checking algorithms

Improved Orchestration[4]

- Partially-order models/requirements to maximize reuse
  - Requirement grouping based on COI (structural and semantic)
- Improved localization abstraction
  - Semantically similar requirements are localized concurrently





Upto 72x speedup

[4] R. Dureja, J. Baumgartner, A. Ivrii, R. Kanzelman, and K. Y. Rozier. "Boosting Verification Scalability via Structural Grouping and Semantic Partitioning of Properties" (FMCAD 2019)

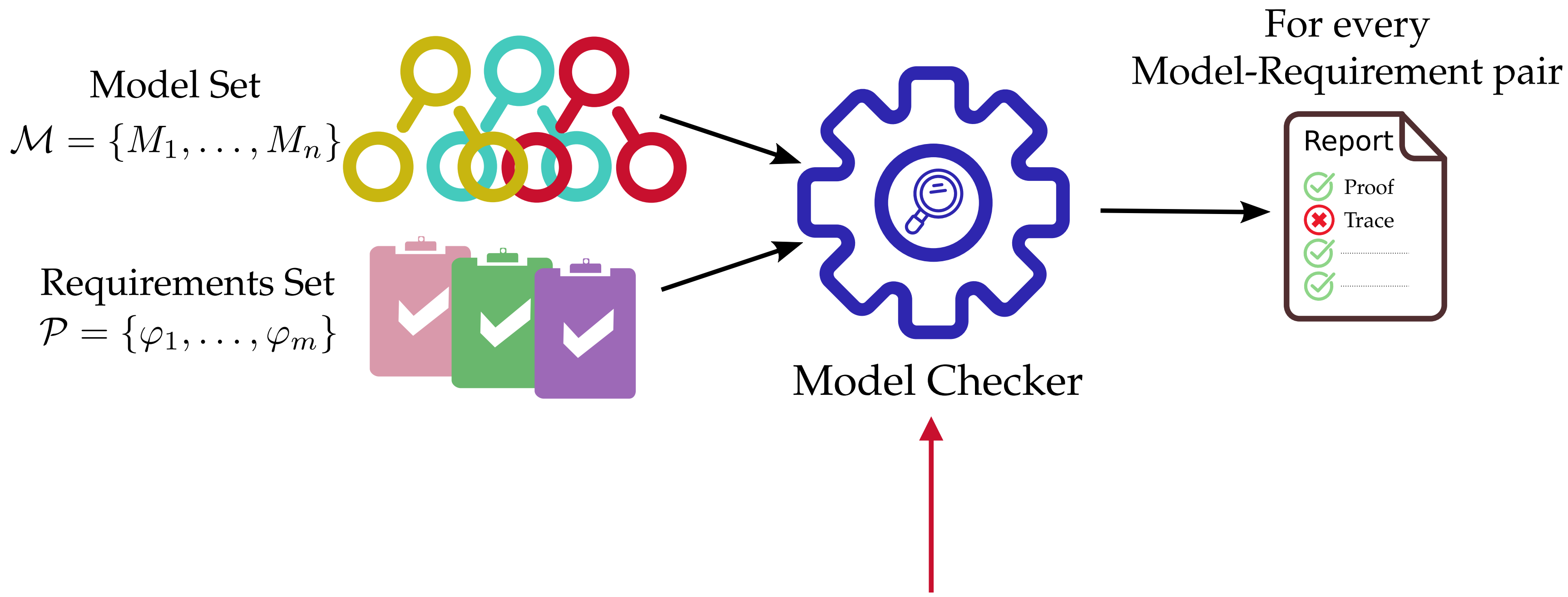
Multi-model/requirement Checking

Design-space reduction	Incremental Verification	Improved Orchestration	Model-checking algorithms

Multi-model/requirement Checking

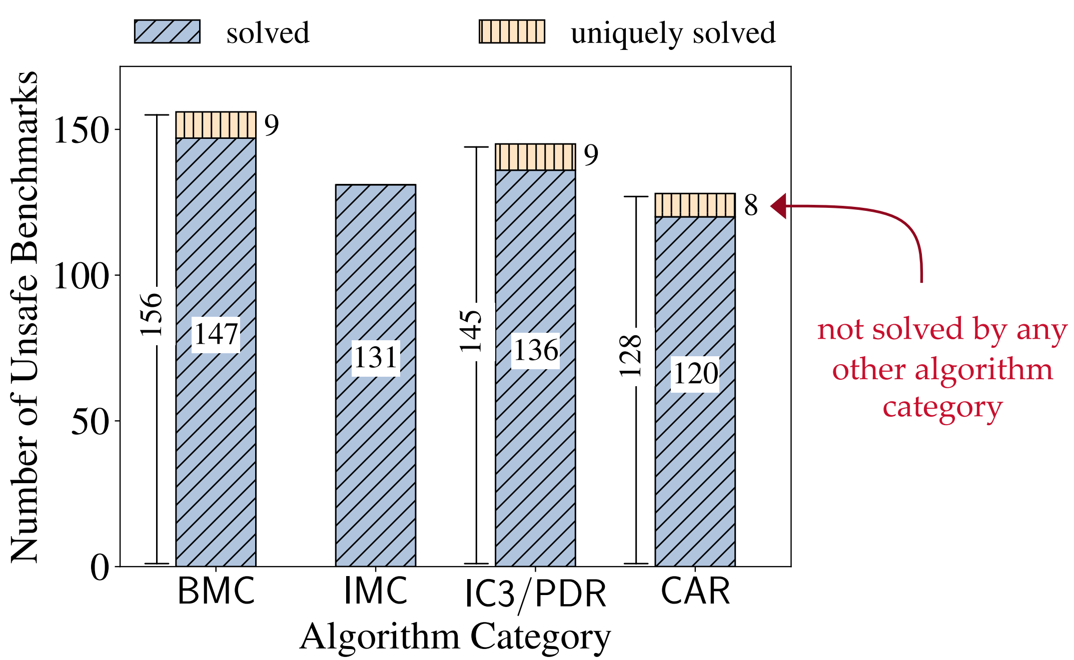
Design-space reduction	Incremental Verification	Improved Orchestration	Model-checking algorithms

Model Checking Algorithms



Model Checking Algorithms[6,7]

- Improve SAT-based model checking algorithms
  - Complementary approximate reachability (CAR) as proof-of-concept [5]
- Heuristics to improve bug-finding performance of CAR
  - SimpleCAR can find bugs not found by IC3/BMC [6]; slow convergence
  - Better SAT-query to improve performance of SimpleCAR [7]
- Also applicable to IC3; more scalable design-space checking



[5] J. Li, S. Zhu, Y. Zhang, G. Pu, and M. Y. Vardi. "Safety model checking with complementary approximations" ICCAD (2017)
[6] J. Li, R. Dureja, G. Pu, K. Y. Rozier, M. Y. Vardi. "SimpleCAR: An Efficient Bug-Finding Tool Based on Approximate Reachability" (CAV 2018)
[7] R. Dureja, J. Li, G. Pu, M. Y. Vardi, K. Y. Rozier. "Intersection and Rotation of Assumption Literals Boosts Bug-Finding" (VSTTE 2019)

Standard Reachability Analysis

Standard Reachability Analysis

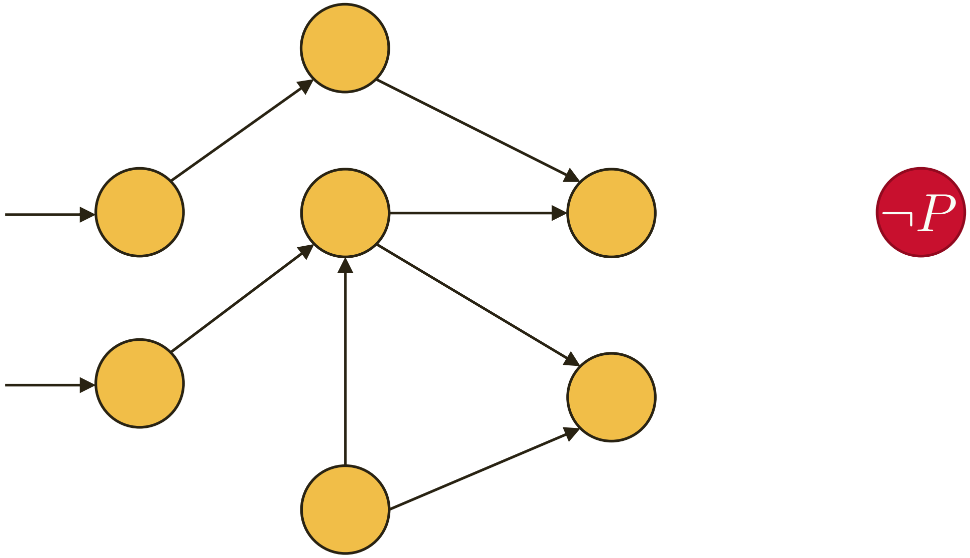
Model $M = (V, I, T)$

Safety Property P

Standard Reachability Analysis

Model $M = (V, I, T)$

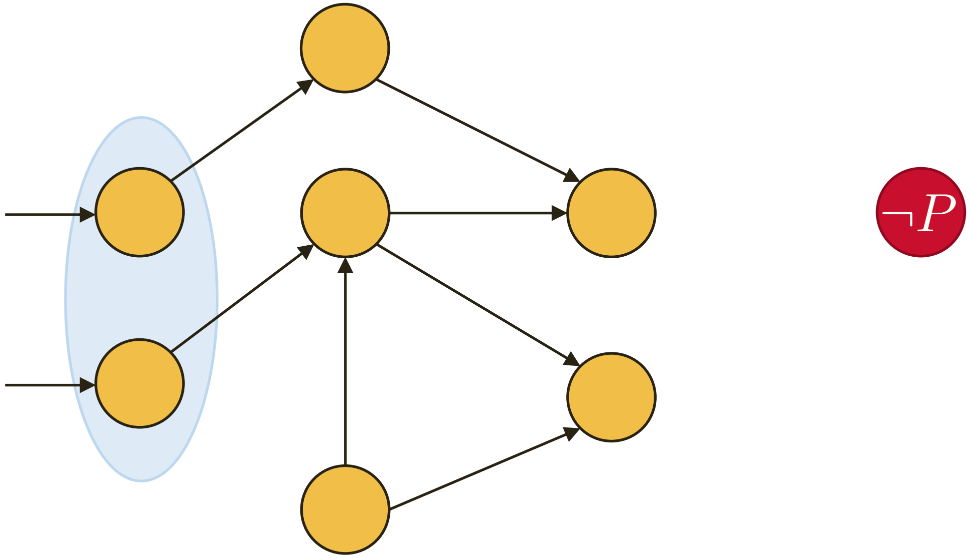
Safety Property P



Standard Reachability Analysis

Model $M = (V, I, T)$

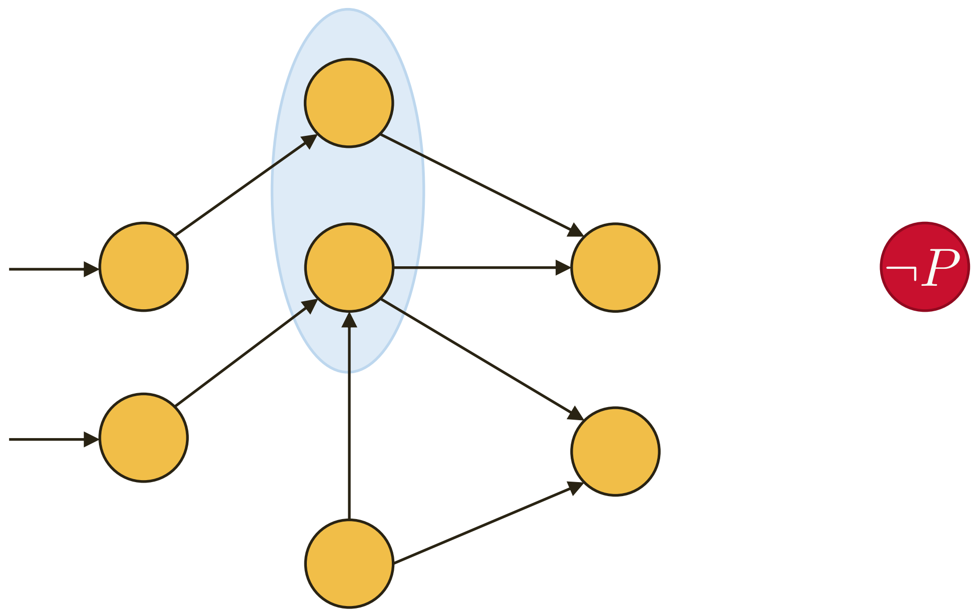
Safety Property P



Standard Reachability Analysis

Model $M = (V, I, T)$

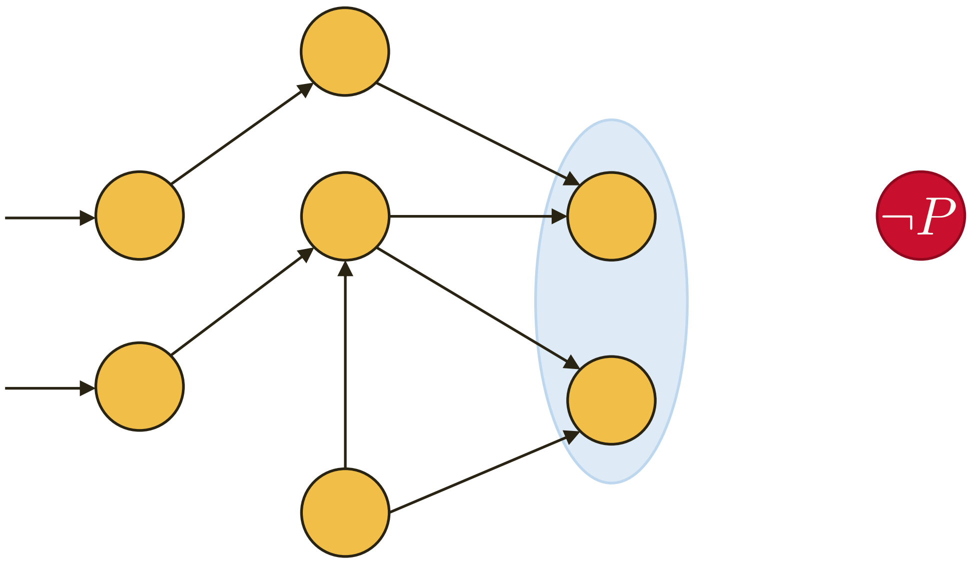
Safety Property P



Standard Reachability Analysis

Model $M = (V, I, T)$

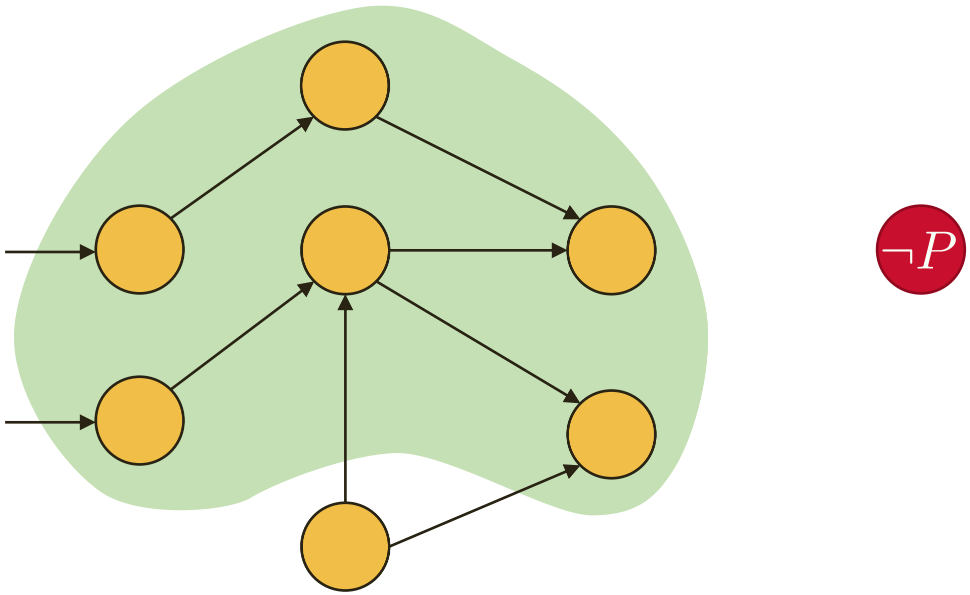
Safety Property P



Standard Reachability Analysis

Model $M = (V, I, T)$

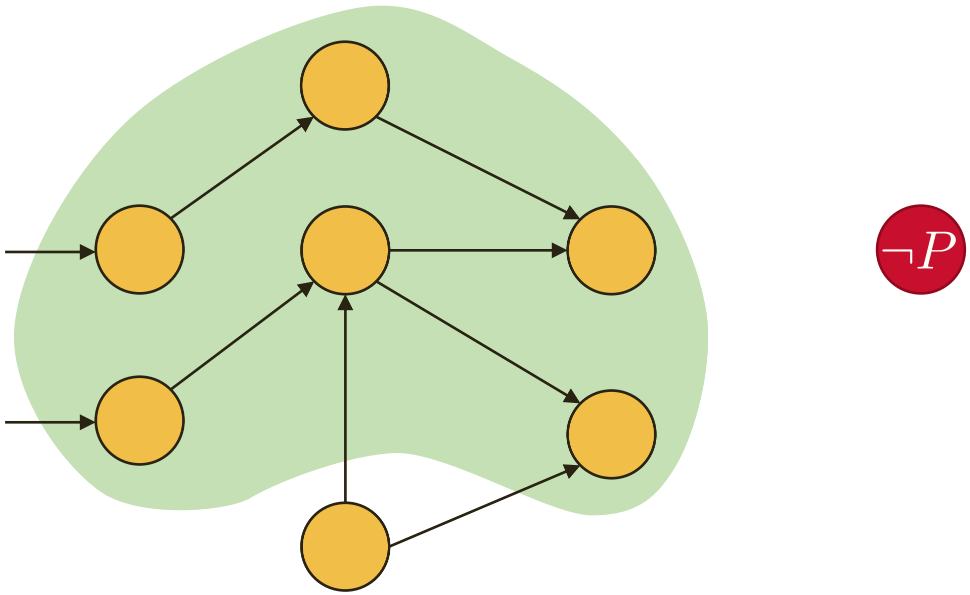
Safety Property P



Standard Reachability Analysis

Model $M = (V, I, T)$

Safety Property P

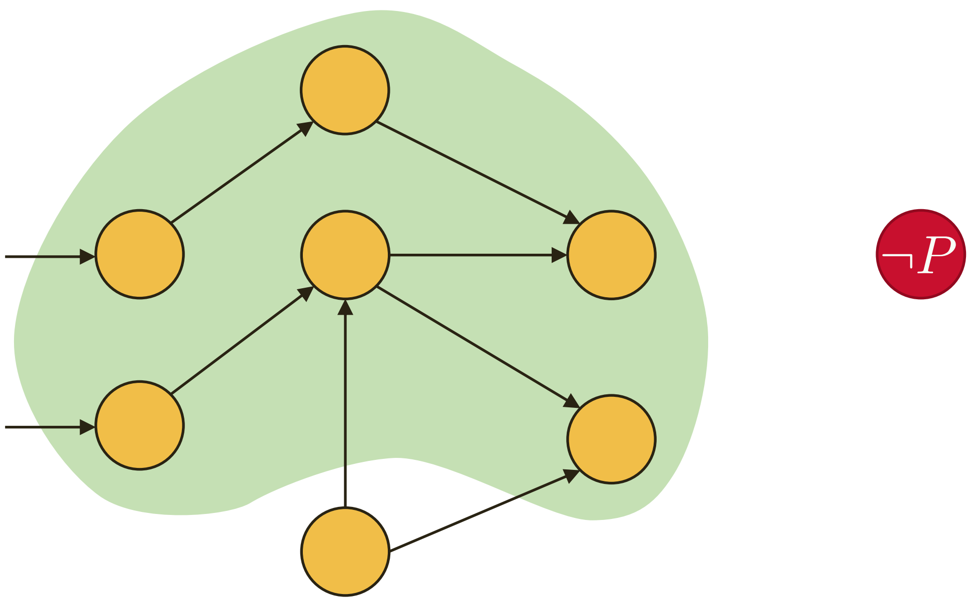


$M \models P$

Standard Reachability Analysis

Model $M = (V, I, T)$

Safety Property P



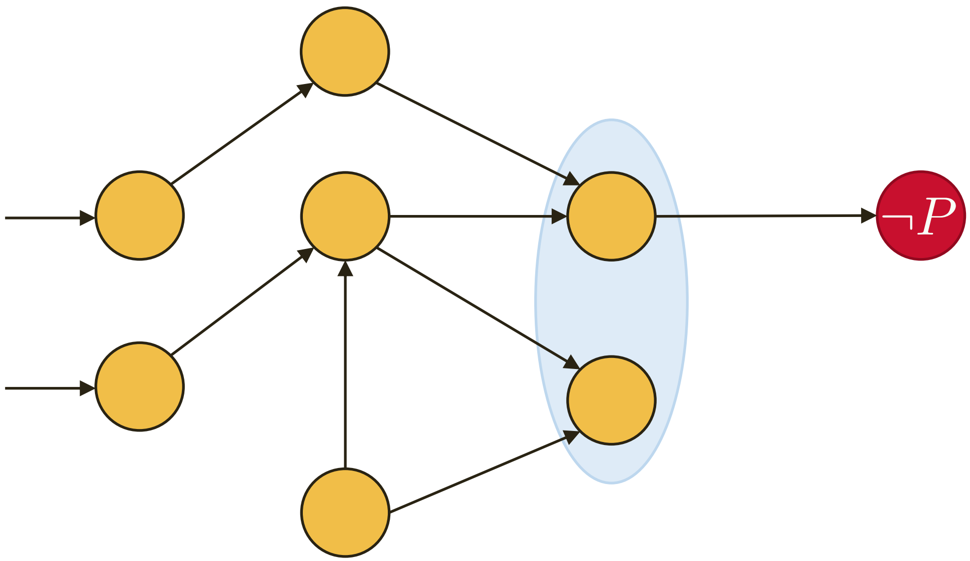
$$M \models P$$

M is **safe** with respect to P

Standard Reachability Analysis

Model $M = (V, I, T)$

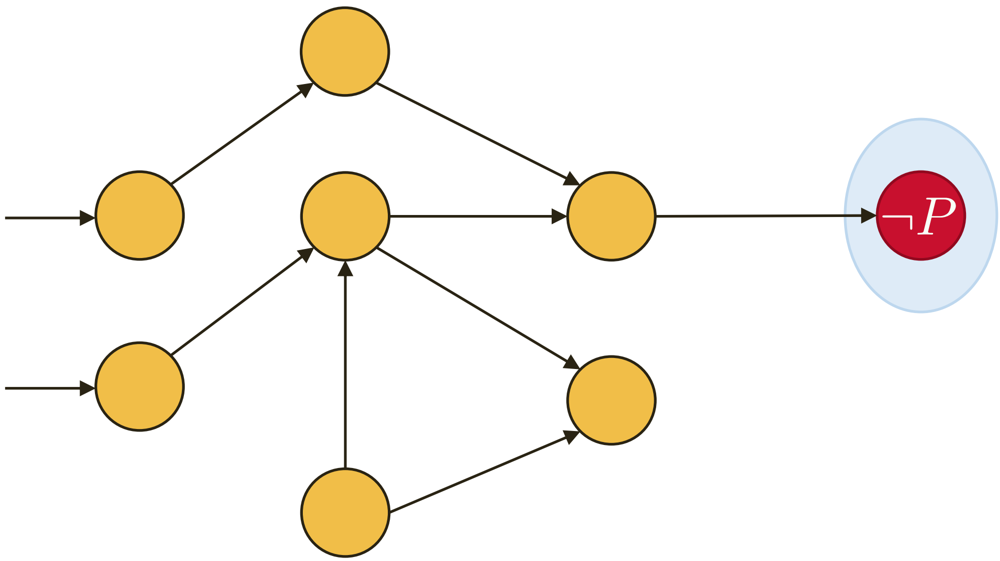
Safety Property P



Standard Reachability Analysis

Model $M = (V, I, T)$

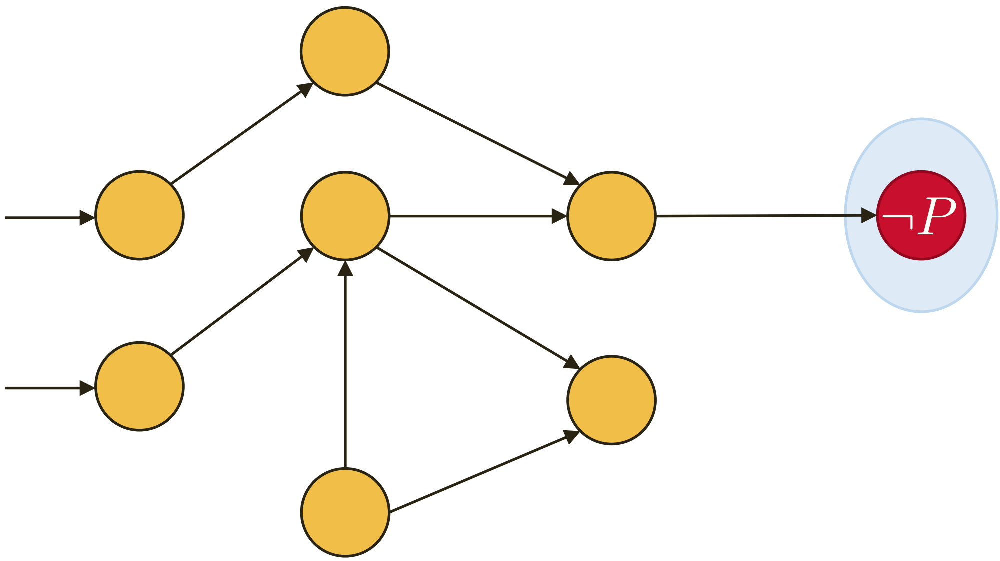
Safety Property P



Standard Reachability Analysis

Model $M = (V, I, T)$

Safety Property P

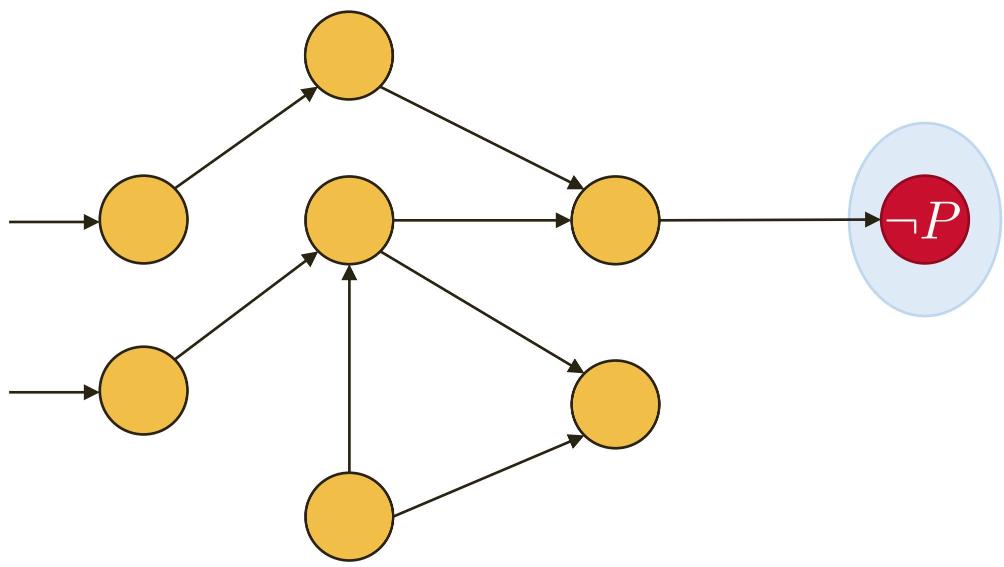


$M \not\models P$

Standard Reachability Analysis

Model $M = (V, I, T)$

Safety Property P

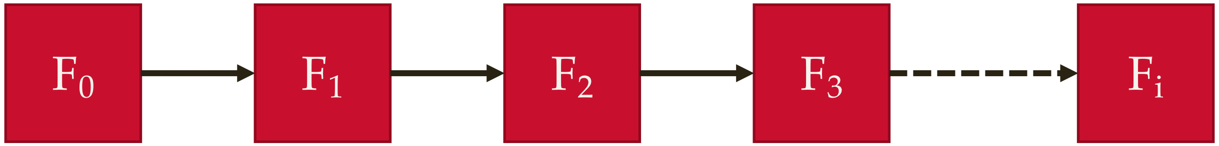


$$M \not\models P$$

M is **unsafe** with respect to P

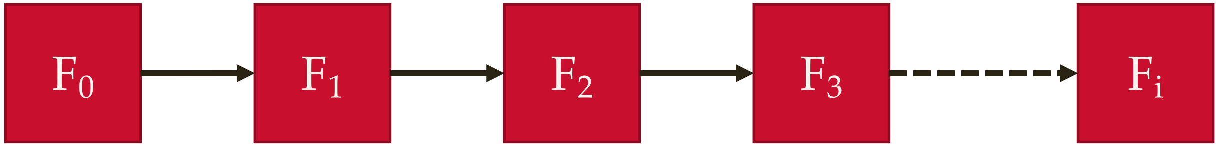
Complementary Approximate Reachability

Standard Reachability Analysis



Complementary Approximate Reachability

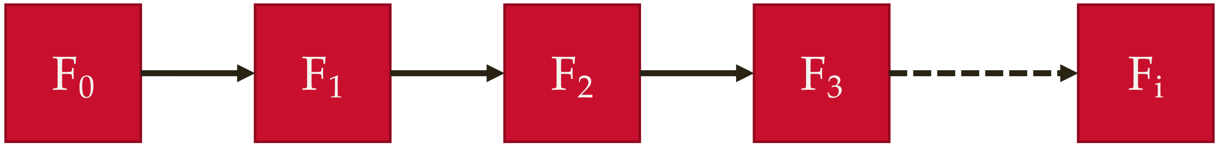
Standard Reachability Analysis



Basic: $F_0 = I$

Complementary Approximate Reachability

Standard Reachability Analysis

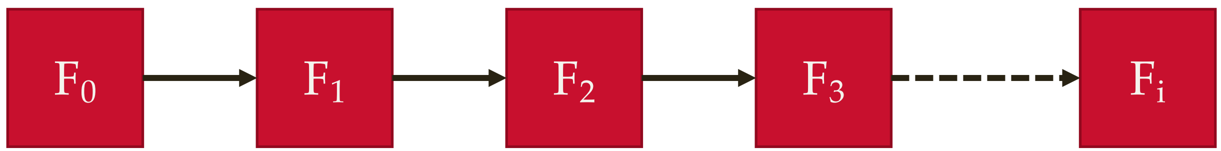


Basic: $F_0 = I$

Induction: $F_{i+1} = Reach(F_i)$

Complementary Approximate Reachability

Standard Reachability Analysis



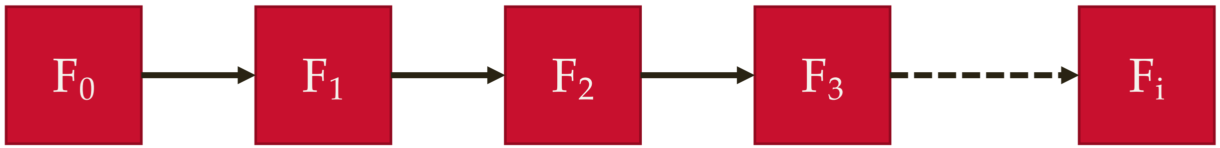
Basic: $F_0 = I$

Induction: $F_{i+1} = Reach(F_i)$

Terminate: $F_{i+1} \subseteq \bigcup_{0 \leq j \leq i} F_j$

Complementary Approximate Reachability

Standard Reachability Analysis



Basic: $F_0 = I$

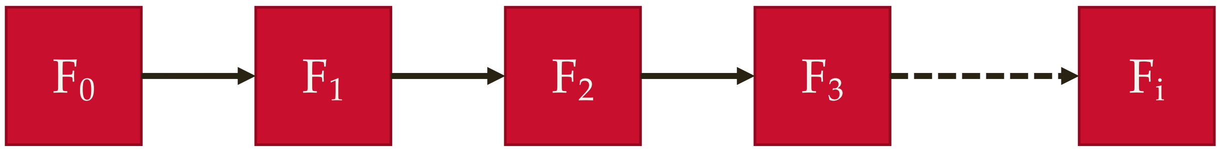
Induction: $F_{i+1} = Reach(F_i)$

Terminate: $F_{i+1} \subseteq \bigcup_{0 \leq j \leq i} F_j$

Check: $F_i \cap \neg P \neq \emptyset$

Complementary Approximate Reachability

Standard Reachability Analysis



Basic: $F_0 = I$

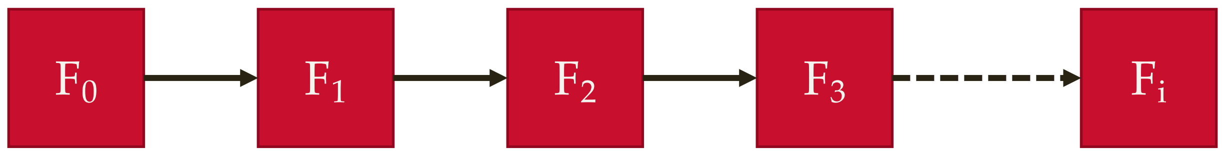
Induction: $F_{i+1} = Reach(F_i)$

Terminate: $F_{i+1} \subseteq \bigcup_{0 \leq j \leq i} F_j$ ← Safety

Check: $F_i \cap \neg P \neq \emptyset$ ← Unsafety (bug-finding)

Complementary Approximate Reachability

Standard Reachability Analysis



Basic: $F_0 = I$

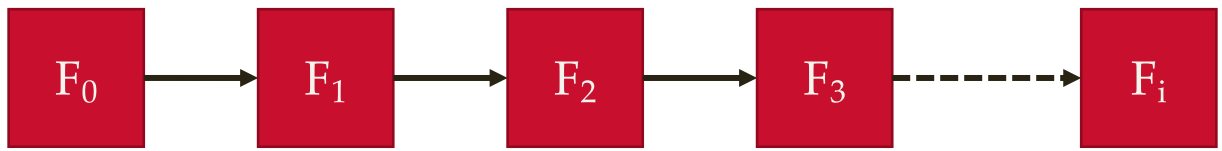
Induction: $F_{i+1} = Reach(F_i)$

Terminate: $F_{i+1} \subseteq \bigcup_{0 \leq j \leq i} F_j$ ← Safety

Check: $F_i \cap \neg P \neq \emptyset$ ← Unsafety (bug-finding)

Complementary Approximate Reachability

Standard Reachability Analysis



Basic: $F_0 = I$

Induction: $F_{i+1} = Reach(F_i)$

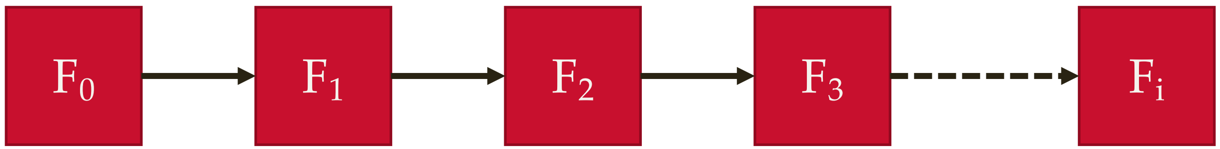
Terminate: $F_{i+1} \subseteq \bigcup_{0 \leq j \leq i} F_j$ ← Safety

Check: $F_i \cap \neg P \neq \emptyset$ ← Unsafety (bug-finding)

Maintaining exact frame sequences is hard; more states in memory

Complementary Approximate Reachability

Standard Reachability Analysis



Basic: $F_0 = I$

Induction: $F_{i+1} = Reach(F_i)$

Terminate: $F_{i+1} \subseteq \bigcup_{0 \leq j \leq i} F_j$ ← Safety

Check: $F_i \cap \neg P \neq \emptyset$ ← Unsafety (bug-finding)

Maintaining exact frame sequences is hard; more states in memory

CAR uses approximate sequences

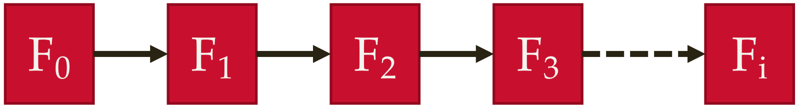
Complementary Approximate Reachability

Maintains two approximate sequences

Complementary Approximate Reachability

Maintains two approximate sequences

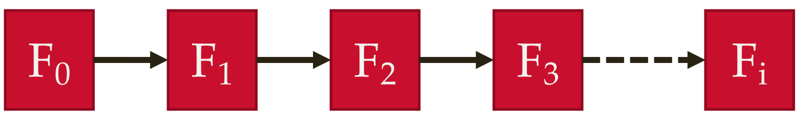
Forward Sequence



Complementary Approximate Reachability

Maintains two approximate sequences

Forward Sequence
(over-approximate)



Basic: $F_0 = I$

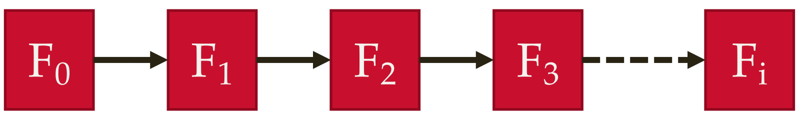
Induction: $F_{i+1} \supseteq Reach(F_i)$

Terminate: $F_{i+1} \subseteq \bigcup_{0 \leq j \leq i} F_j$

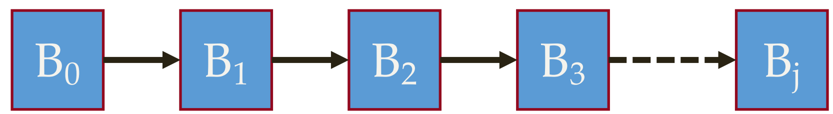
Complementary Approximate Reachability

Maintains two approximate sequences

Forward Sequence
(over-approximate)



Backward Sequence
(under-approximate)



Basic:	$F_0 = I$
Induction:	$F_{i+1} \supseteq Reach(F_i)$
Terminate:	$F_{i+1} \subseteq \bigcup_{0 \leq j \leq i} F_j$

Basic:	$B_0 = \neg P$
Induction:	$B_{j+1} \subseteq Reach^{-1}(B_j)$
Check:	$B_j \cap I \neq \emptyset$

Inverse transition

Complementary Approximate Reachability

Maintains two approximate sequences

Forward Sequence (over-approximate)

$F_0 \rightarrow F_1 \rightarrow F_2 \rightarrow F_3 \dashrightarrow F_i$

Basic: $F_0 = I$

Induction: $F_{i+1} \supseteq Reach(F_i)$

Terminate: $F_{i+1} \subseteq \bigcup_{0 \leq j \leq i} F_j$

Safety Checking

Backward Sequence (under-approximate)

$B_0 \rightarrow B_1 \rightarrow B_2 \rightarrow B_3 \dashrightarrow B_j$

Basic: $B_0 = \neg P$

Induction: $B_{j+1} \subseteq Reach^{-1}(B_j)$ (Inverse transition)

Check: $B_j \cap I \neq \emptyset$

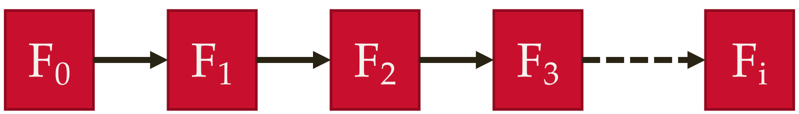
Unsafety Checking

Complementary Approximate Reachability

Maintains two approximate sequences

Forward-CAR

Forward Sequence (over-approximate)



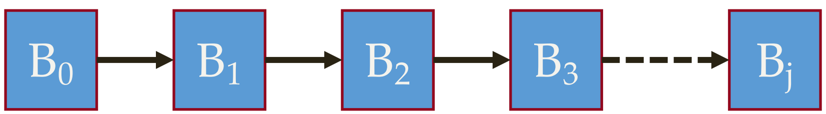
Basic: $F_0 = I$

Induction: $F_{i+1} \supseteq Reach(F_i)$

Terminate: $F_{i+1} \subseteq \bigcup_{0 \leq j \leq i} F_j$

Safety Checking

Backward Sequence (under-approximate)



Basic: $B_0 = \neg P$

Inverse transition

Induction: $B_{j+1} \subseteq Reach^{-1}(B_j)$

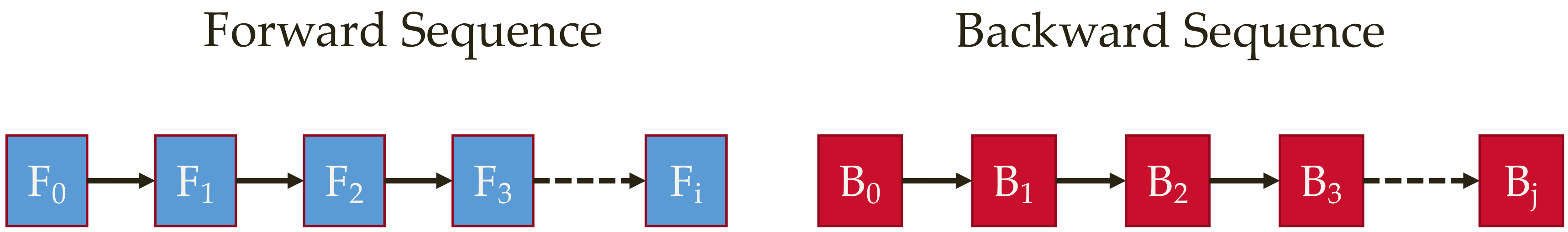
Check: $B_j \cap I \neq \emptyset$

Unsafety Checking

Complementary Approximate Reachability

Maintains two approximate sequences

Backward-CAR



Complementary Approximate Reachability

Maintains two approximate sequences

Backward-CAR

Forward Sequence
(under-approximate)

Backward Sequence

Basic: $F_0 = I$

Induction: $F_{i+1} \subseteq Reach(F_i)$

Check: $F_i \cap \neg P \neq \emptyset$

Complementary Approximate Reachability

Maintains two approximate sequences

Backward-CAR

Forward Sequence
(under-approximate)

Backward Sequence
(over-approximate)

Basic:	$F_0 = I$
Induction:	$F_{i+1} \subseteq Reach(F_i)$
Check:	$F_i \cap \neg P \neq \emptyset$

Basic:	$B_0 = \neg P$
Induction:	$B_{j+1} \supseteq Reach^{-1}(B_j)$
Terminate:	$B_{j+1} \subseteq \bigcup_{0 \leq k \leq j} B_k$

Complementary Approximate Reachability

Maintains two approximate sequences

Backward-CAR

Forward Sequence
(under-approximate)

Backward Sequence
(over-approximate)

Basic:	$F_0 = I$
Induction:	$F_{i+1} \subseteq Reach(F_i)$
Check:	$F_i \cap \neg P \neq \emptyset$

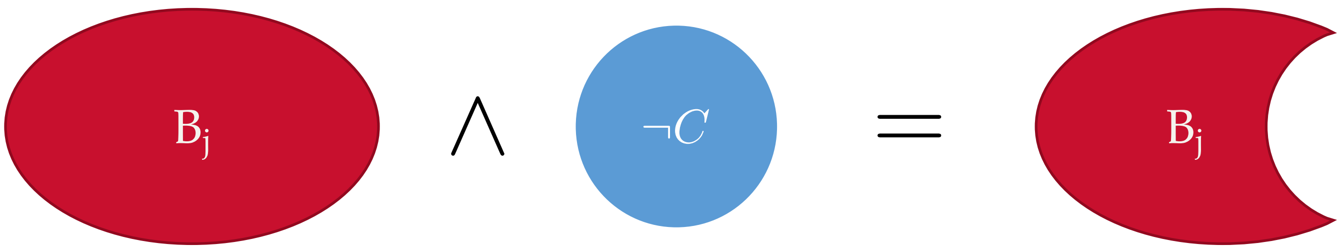
Unsafety Checking

Basic:	$B_0 = \neg P$
Induction:	$B_{j+1} \supseteq Reach^{-1}(B_j)$
Terminate:	$B_{j+1} \subseteq \bigcup_{0 \leq k \leq j} B_k$

Safety Checking

Unsat Cores and CAR

- Unsat cores play a critical role in the performance of CAR
  - Iteratively blocking overapproximate states (B-sequence), much like IC3



- Our quest for smallest unsat cores
  - CARChecker (ICCAD 2017) uses minimal unsat cores – slow!
  - SimpleCAR (CAV 2018) uses first unsat core– fast, but slow convergence
- Tradeoff – smaller v/s faster
  - Find smaller (not minimal) unsat cores fast
- We propose heuristics that find smaller cores; negligible overhead

Assumptions and SAT Solver

$$\mathrm{SAT}(\varphi, A) \equiv \mathrm{SAT}(\varphi \wedge A)$$

$\varphi =$ Boolean formula in CNF

$A =$ Set of assumption literals

- Query UNSAT → Core $C \subseteq A$ and $\varphi \wedge C$ is UNSAT
- C is not necessarily minimal
- Assumption literals are stored in a vector (e.g., MiniSAT)

Let $A = \{a_0, a_1, a_2, a_3, a_4, a_5, \ldots, a_n\}$

a_0	a_1	a_2	a_3	a_4	a_5	$\cdots$	a_n

- Solver propagates each literal one-by-one; left → right

Assumptions and SAT Solver

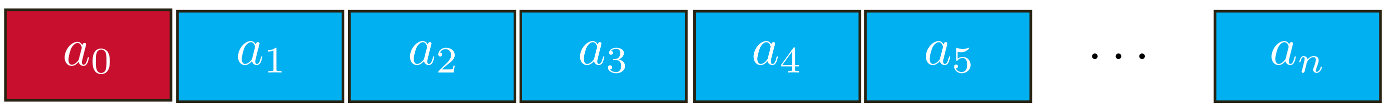
$$\text{SAT}(\varphi, A) \equiv \text{SAT}(\varphi \wedge A)$$

$\varphi =$ Boolean formula in CNF

$A =$ Set of assumption literals

- Query UNSAT → Core $C \subseteq A$ and $\varphi \wedge C$ is UNSAT
- C is not necessarily minimal
- Assumption literals are stored in a vector (e.g., MiniSAT)

Let $A = \{a_0, a_1, a_2, a_3, a_4, a_5, \ldots, a_n\}$



- Solver propagates each literal one-by-one; left → right

Assumptions and SAT Solver

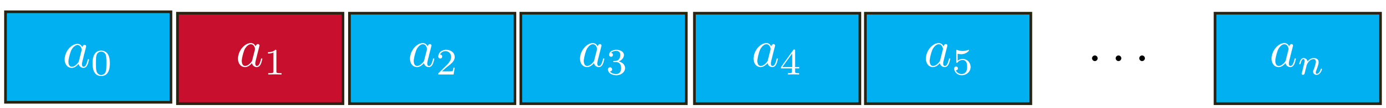
$$\mathrm{SAT}(\varphi, A) \equiv \mathrm{SAT}(\varphi \wedge A)$$

$\varphi =$ Boolean formula in CNF

$A =$ Set of assumption literals

- Query UNSAT → Core $C \subseteq A$ and $\varphi \wedge C$ is UNSAT
- C is not necessarily minimal
- Assumption literals are stored in a vector (e.g., MiniSAT)

Let $A = \{a_0, a_1, a_2, a_3, a_4, a_5, \ldots, a_n\}$



- Solver propagates each literal one-by-one; left → right

Assumptions and SAT Solver

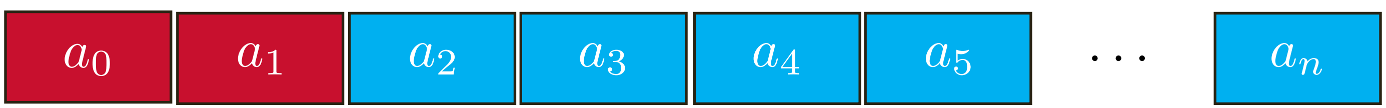
$$\mathrm{SAT}(\varphi, A) \equiv \mathrm{SAT}(\varphi \wedge A)$$

$\varphi =$ Boolean formula in CNF

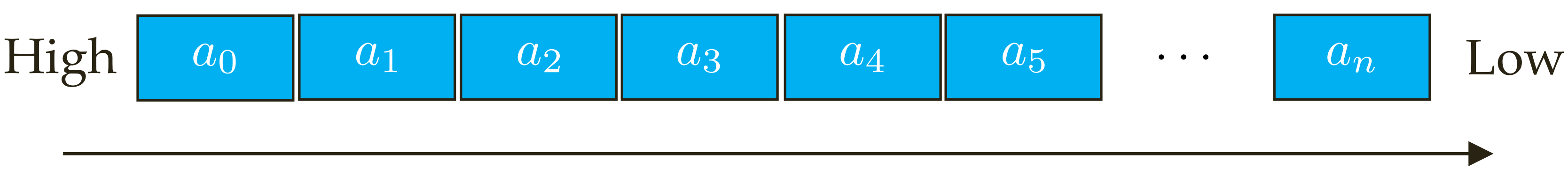
$A =$ Set of assumption literals

- Query UNSAT → Core $C \subseteq A$ and $\varphi \wedge C$ is UNSAT
- C is not necessarily minimal
- Assumption literals are stored in a vector (e.g., MiniSAT)

Let $A = \{a_0, a_1, a_2, a_3, a_4, a_5, \ldots, a_n\}$



- Solver propagates each literal one-by-one; left → right
- Front literals have higher chance to be in unsat core C



Proposed Heuristics

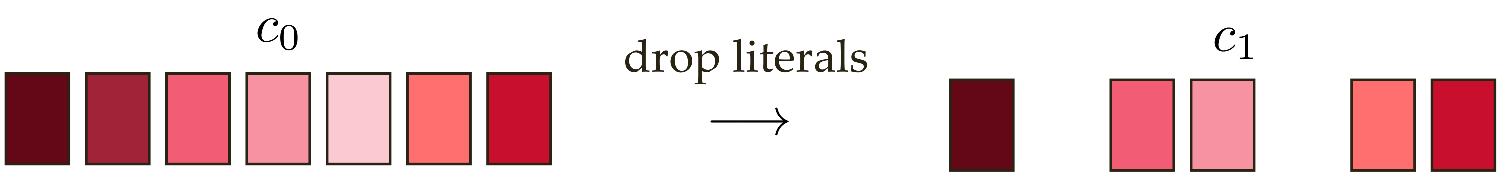
- Carefully reorder the assumption literals
  - Drives SAT solvers to return smaller unsat cores
- Intuition
  - Use **old** unsat cores to drive search for **new** unsat cores

Blocking Step

For some state s, if $\mathrm{SAT}(T \wedge B_j, s)$ is UNSAT, add $c \subseteq s$ to B_{j+1}

Let $\neg c_0$ be the last-added clause to $B_{j+1} \leftarrow c_0 \wedge T \wedge B_j$ is UNSAT

(some state s)



$c_1 \wedge T \wedge B_j$ is UNSAT

c_1 is weaker than c_0, and blocks more states at B_{j+1}

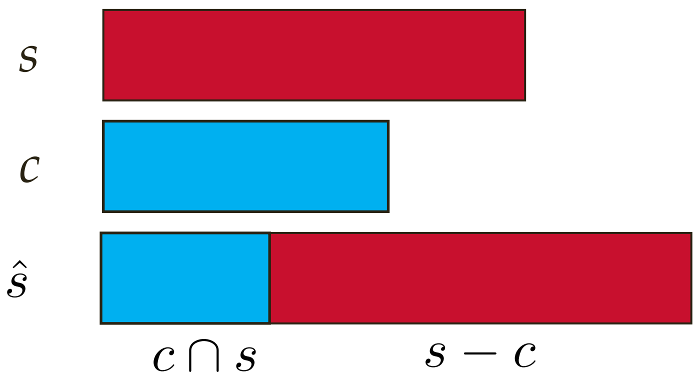
Heuristic I - Intersection

- **Default:** Let s be a state to be blocked at B_{j+1} (s picked from F-sequence)

$$\text{Check SAT}(T \wedge B_j, s)$$

- **Heuristic:** Reorder literals in s to generate $\hat{s}$

  Let $\neg c$ be the last clause added to B_{j+1}



$$\text{Check SAT}(T \wedge B_j, \hat{s})$$
$$(\text{note } \hat{s} = s)$$

- If UNSAT, higher chance of $c \cap s$ literals included in unsat core
- Weaker clause; more states than $\neg c$ blocked at B_{j+1}

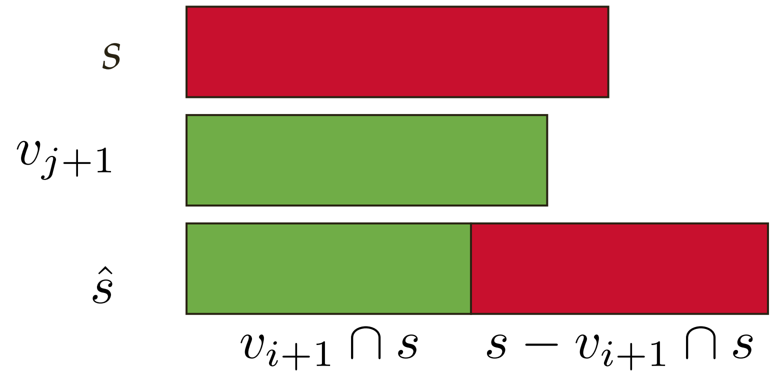
Heuristic II - Rotation

- CAR picks state from the F-sequence; checks intersection with bad states
  - Ideally, want states to explore disjoint parts of the state space
- **Default:** Let s be a state to be blocked at B_{j+1} (s picked from F-sequence)

$$\text{Check SAT}(T \wedge B_j, s)$$

If SAT, the assignment is a state t; can be reached from s. State t is added to F-sequence

- A set of states S is *diverse* if $\bigcap_{t \in S} t = \emptyset$; disjoint states
- **Heuristic:** Reorder literals in s to generate
  - Every B_j ($j > 0$) is associated with v_j to store assumptions from last B_{j-1} query



$$\text{Check SAT}(T \wedge B_j, \hat{s})$$
(note $\hat{s} = s$)

- Generate diverse states whenever query is SAT (proof in the paper)

Experimental Evaluation

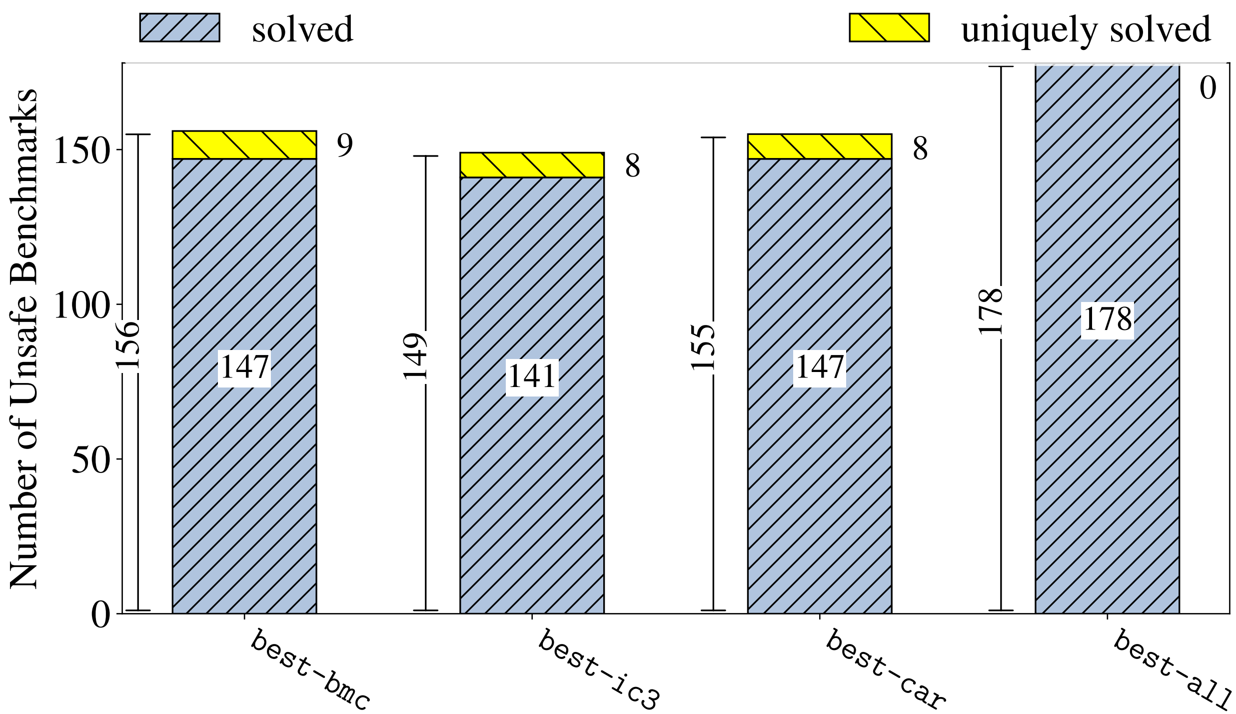
- Extended SimpleCAR to include proposed heuristics
  - Intersection, Rotation, Combination, or None
  - Order of state enumeration; pick *s* from F-sequence
- Tools and algorithm categories compared:
  - ABC `(pdr, 3 x bmc)`
  - IIMC `(bmc, ic3, Quip, ic3r)`
  - IC3Ref `(ic3)`
  - Simplic3 `(bmc, 3 x ic3, Avy)`
  - SimpleCAR `(8 x car)`
- 5 tools, 22 algorithms, 748 SINGLE property benchmarks from HWMCC
- 1 hour timeout
- Identified a bug, and counterexample generation errors
- **We focus on unsafety checking**

Open-source under GNU GPLv3

`http://temporallogic.org/research/VSTTE19/`

High-level Performance

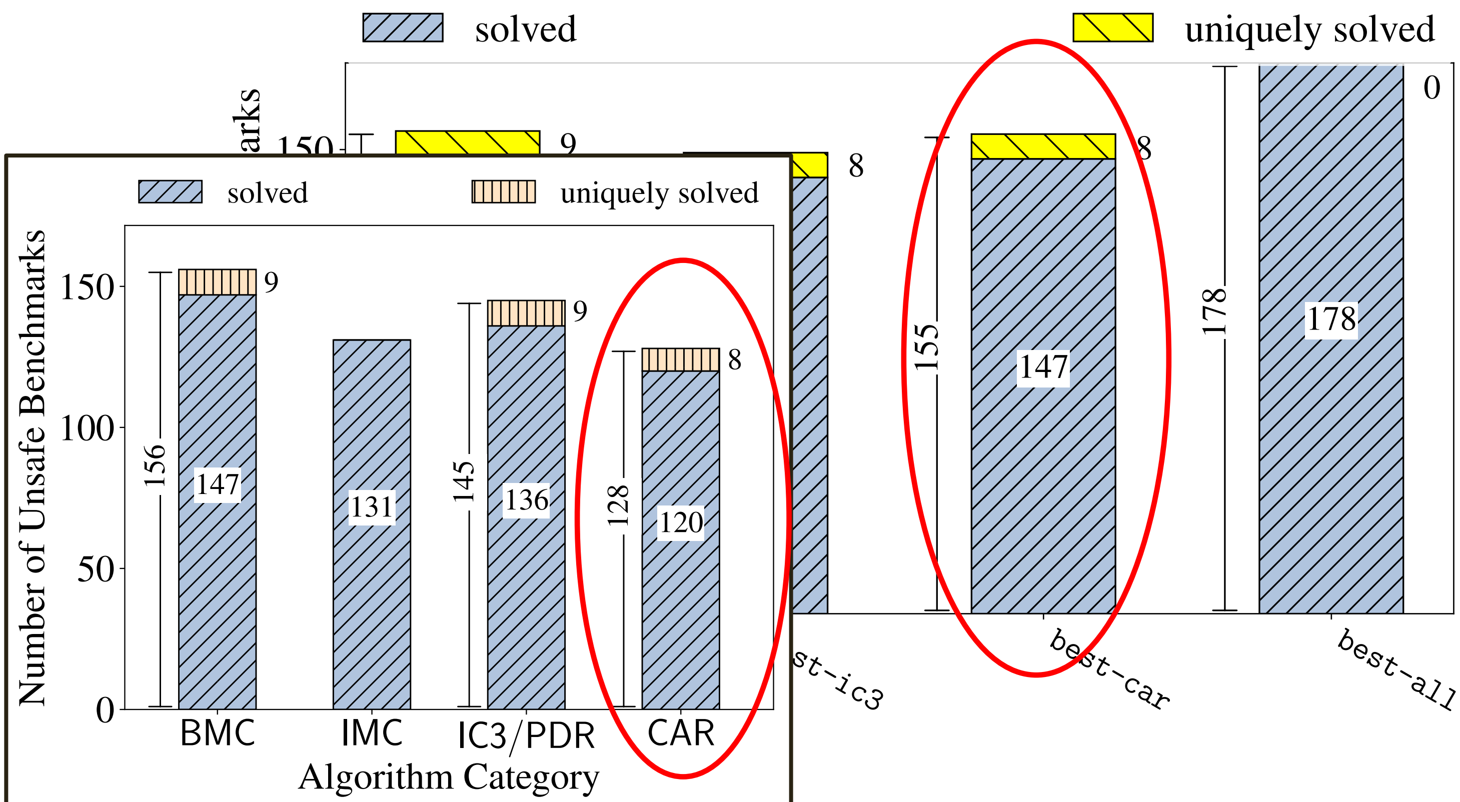
Algorithm Categories



not solved by any other algorithm category

High-level Performance

Algorithm Categories



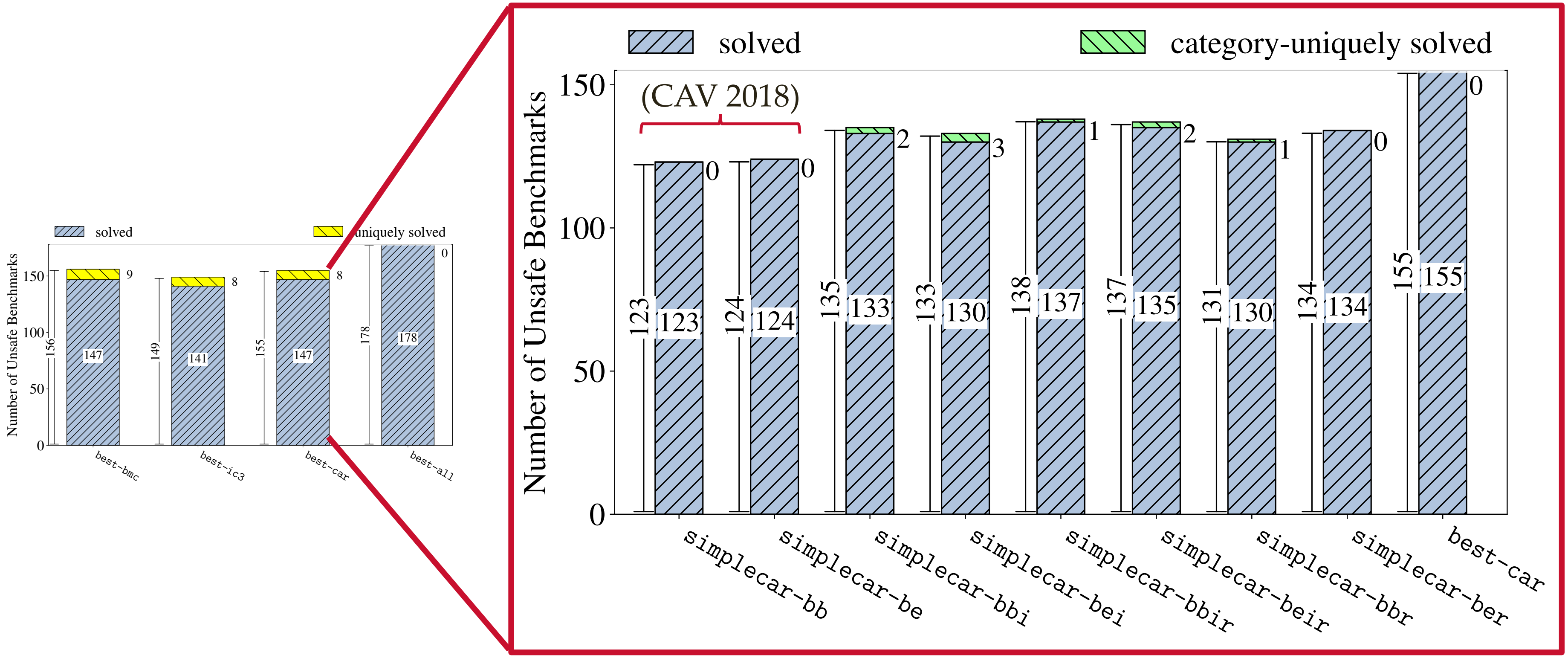
(CAV 2018)

not solved by any other algorithm category

25% more solves

High-level Performance

Virtual-best CAR



`simpcar-bbir` gives 20% smaller unsat cores

On-average 30% faster

Faster convergence!

Summary and Discussion

- Design-space exploration via model checking; many models/requirements
- Focus along four verticals
  - Design-space reduction
  - Incremental verification
  - Improved orchestration
  - Model checking algorithms
- Applicable to equivalence checking, product lines, regression runs, etc.
  - Extensions to existing algorithms, and new specialized algorithms
- Better handling of SAT queries improves model checking performance
  - Proposed two heuristics: Intersection and Rotation
- Heuristics can also be applied for clause generalization in IC3
- Future work and research questions
  - SAT-solver internal heuristics for literal scoring
  - Adapting CAR to handle multiple properties; clause sharing between properties
  - Improved synergy between model checking algorithms and SAT solvers

Thank You!

`http://temporallogic.org/research/VSTTE19/`