Existing miners

SpecForge

Synergizing Specification Miners through Model Fissions and Fusions

Tien-Duy B. Le[1], Xuan-Bach D. Le[1], David Lo[1], Ivan Beschastnikh[2]

[1] *Singapore Management University*

[2] *University of British Columbia*

Existing miners

SpecForge

Synergizing Specification Miners through Model Fissions and Fusions

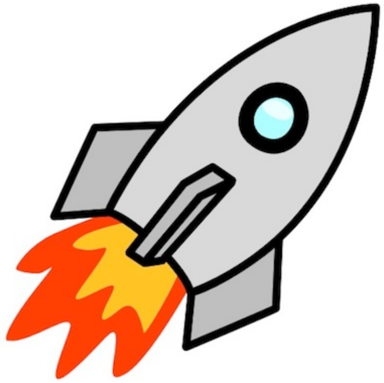
Tien-Duy B. Le[1], Xuan-Bach D. Le[1], David Lo[1], Ivan Beschastnikh[2]

[1] *Singapore Management University*

[2] *University of British Columbia*

Software Specifications

Software systems and libraries usually lack up-to-date formal specifications.



Rapid Software Evolution

Formal specifications are non-trivial to write down

Software Specifications

Lack of Formal Specifications

↓

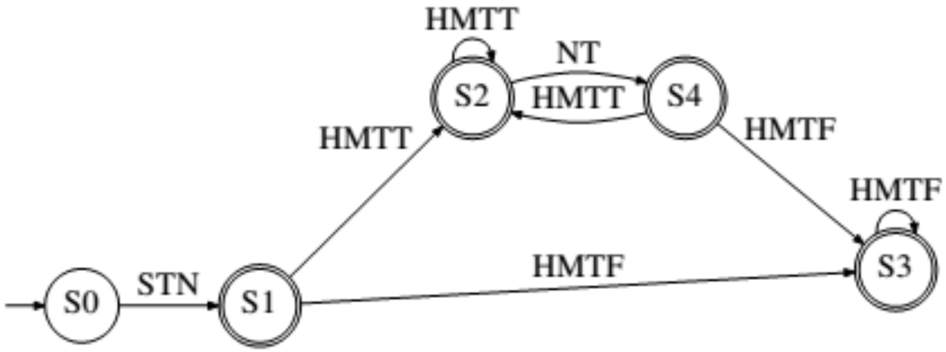
Maintainability & Reliability Challenges

- Reduced code comprehension
- Implicit assumptions may cause bugs
- Difficult to identify regressions

Software Specification Mining

Software Specification Mining

- Many existing specification mining algorithms
  - Most automatically infer specs from *execution traces*
- Our focus: tools that mine FSAs



Finite State Automata (FSA)

Examples: k-tail, CONTRACTOR++, SEKT, TEMI, Synoptic

No Perfect Specification Miner

- Existing miners make complex trade-offs
  - Some use temporal constraints (k-tails)
  - Others use mined data invariants (SEKT)
  - Vary in their robustness to incomplete traces

…

- A proliferation of spec miners
  - Which one to use?

No Perfect Specification Miner

- Existing miners make complex trade-offs

Let's take advantage of this proliferation!
Our contribution: SpecForge

...

- Proliferation of spec miners
  - Which one to use?

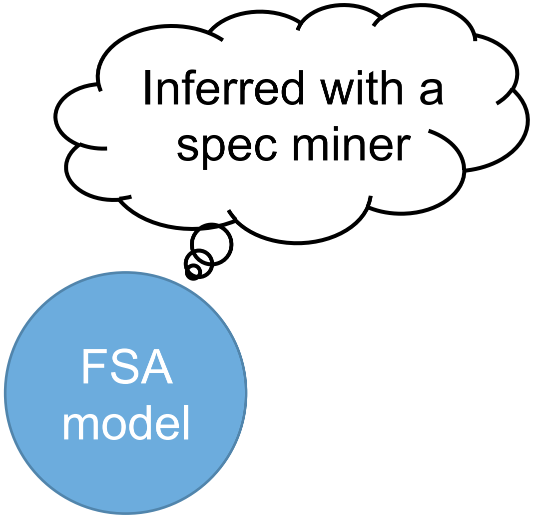
SpecForge overview

Existing miners

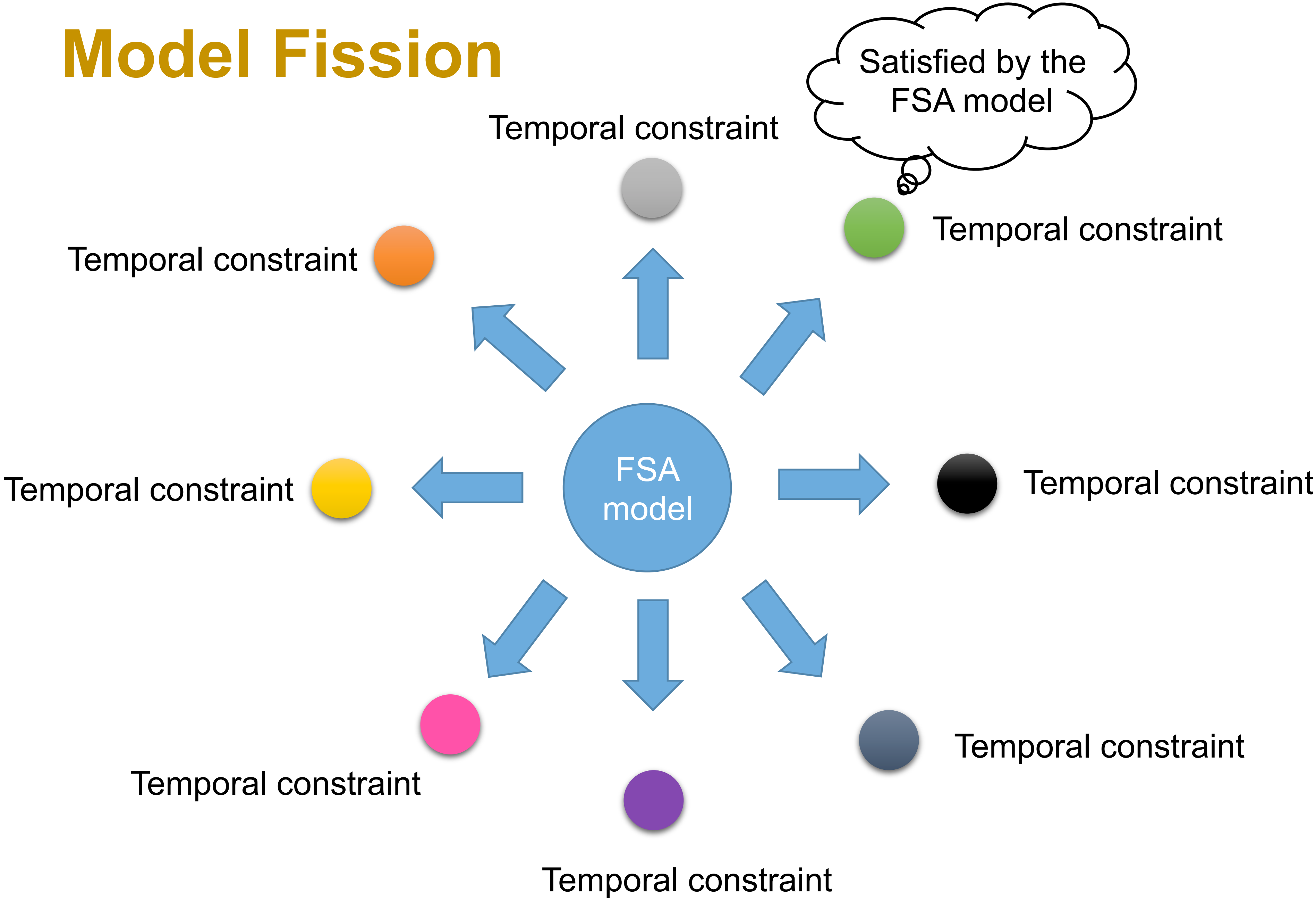
SpecForge

- SpecForge synergizes many FSA-based specification mining algorithms
- New concepts:
  - Model fission & model fusion

Model Fission

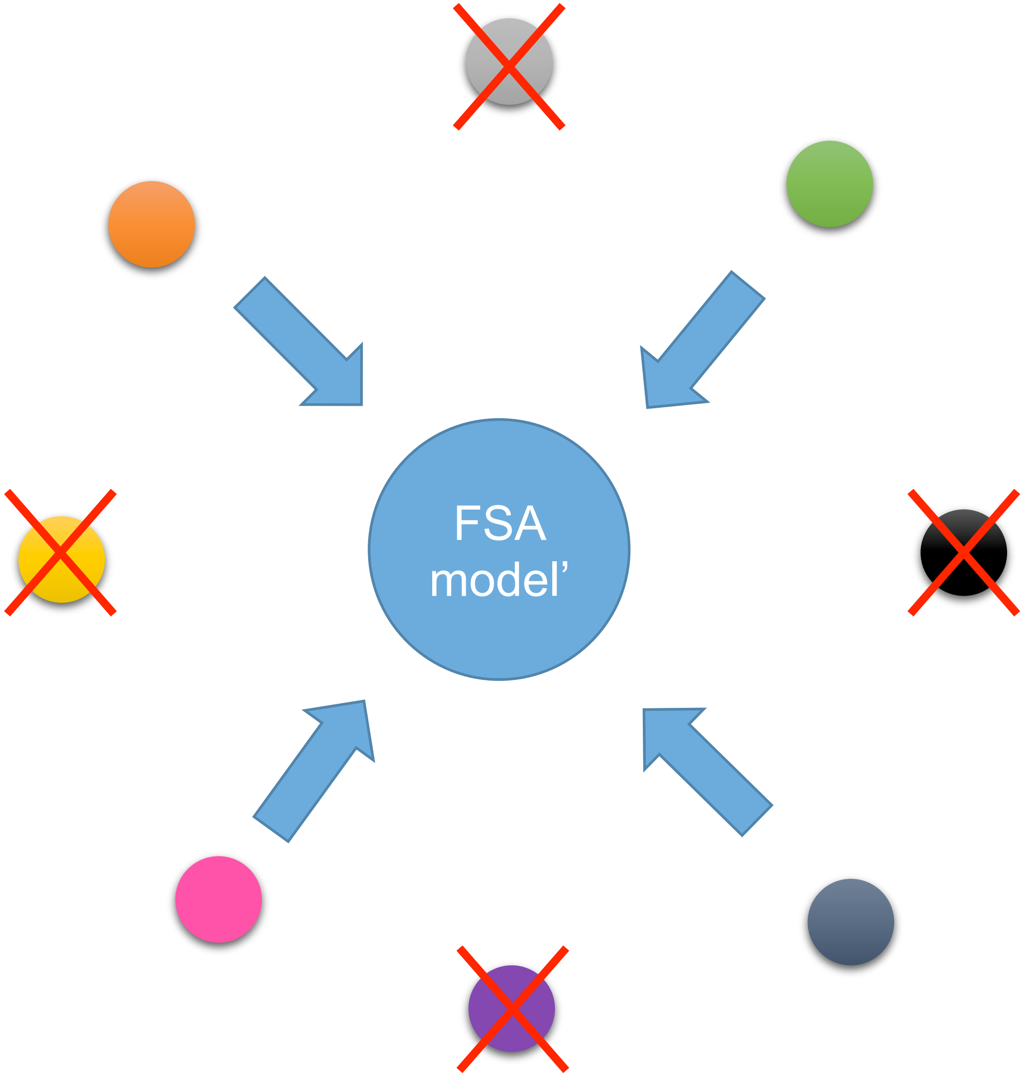


Model Fission



Model Fusion

1. Select temporal constraints
2. Fuse constraints into a new FSA



SpecForge: Overall Framework

Execution traces

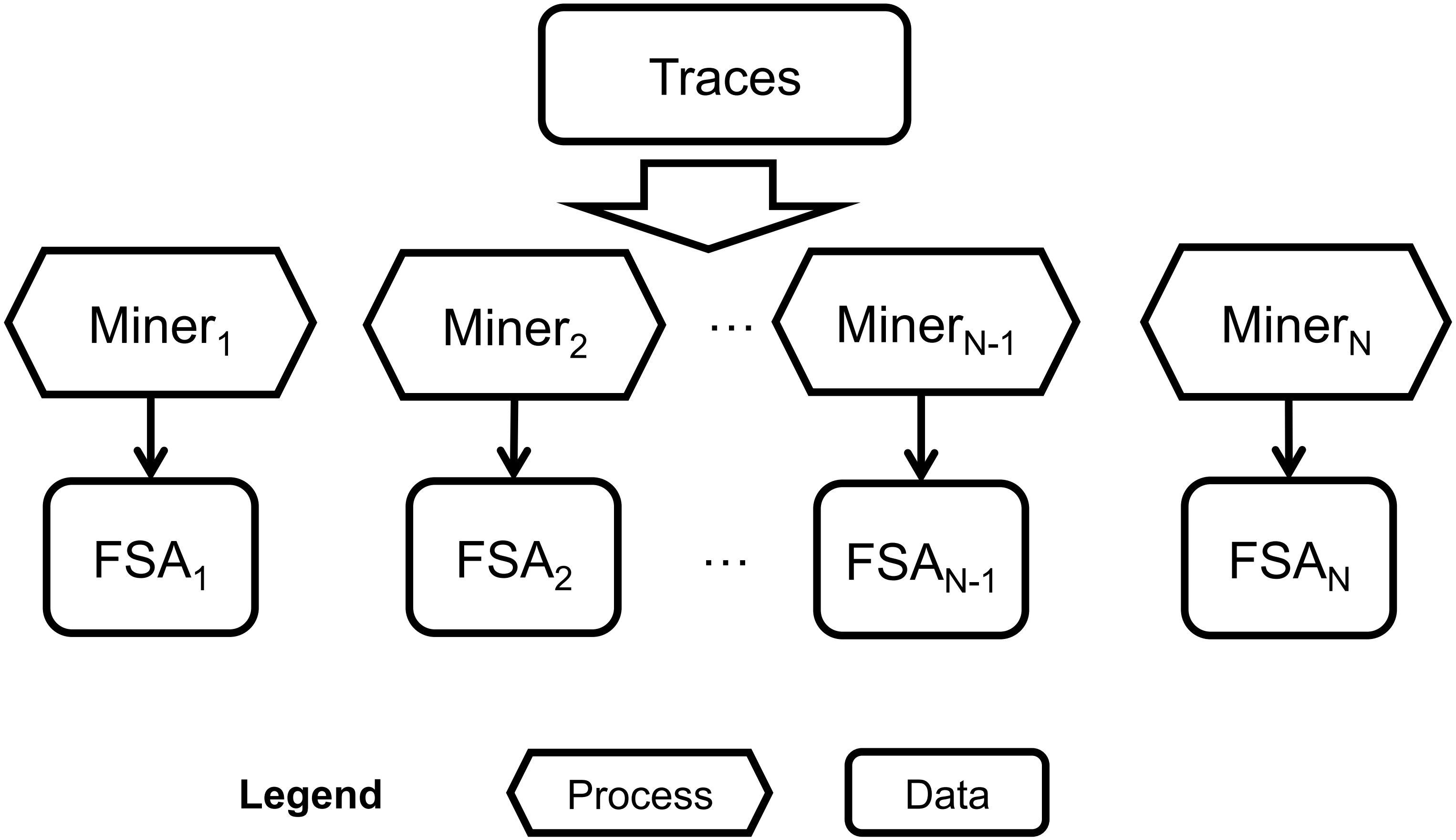
FSA miners

SpecForge

1. Run each spec miner on traces
2. Decompose generated models with fission
3. Build new model using fusion

Phase 1: Models Construction

- Given ***N*** miners, construct ***N*** different FSAs



Phase 2: Models Fission

- Decompose each FSA_i into a set of binary temporal constraints
- Each constraint is expressed in Linear Temporal Logic (LTL)
- In this work we use 6 LTL constraint types

[1] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, “Patterns in property specifications for finite-state verification”. ICSE 1999

[2] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst, “Leveraging existing instrumentation to automatically infer invariantconstrained models,” ESEC/FSE 2011

[3] I. Beschastnikh, Y. Brun, J. Abrahamson, M. D. Ernst, and A. Krishnamurthy, “Using declarative specification to improve the understanding, extensibility, and comparison of model-inference algorithms,” TSE 2015

LTL Constraint Types

- AF(a,b): a is always followed by b

  a b a b ✓ a b b a ⊗

  c b b b ✓ c a a a ⊗

- NF(a,b): a is never followed by b

  b b a a ✓ a b b a ⊗

  a c a a ✓ c b a b ⊗

- AP(a,b): a is always preceded by b

  b b a a ✓ a b b b ⊗

  c b b b ✓ c a a b ⊗

LTL Constraint Types

- AF(a,b) : a is always followed by b

  a b a b ✓ a b b a ⊗

  c b b b ✓ c a a a ⊗

- NF(a,b): a is never followed by b

  b b a a ✓ a b b a ⊗

  a c a a ✓ c b a b ⊗

- AP(a,b): a is always preceded by b

  b b a a ✓ a b b b ⊗

  c b b b ✓ c a a b ⊗

LTL Constraint Types

- AF(a,b) : *a* is always followed by *b*

  a b a b ✓ a b b a ⊗

  c b b b ✓ c a a a ⊗

- NF(a,b): *a* is never followed by *b*

  b b a a ✓ a b b a ⊗

  a c a a ✓ c b a b ⊗

- AP(a,b): *a* is always preceded by *b*

  b b a a ✓ a b b b ⊗

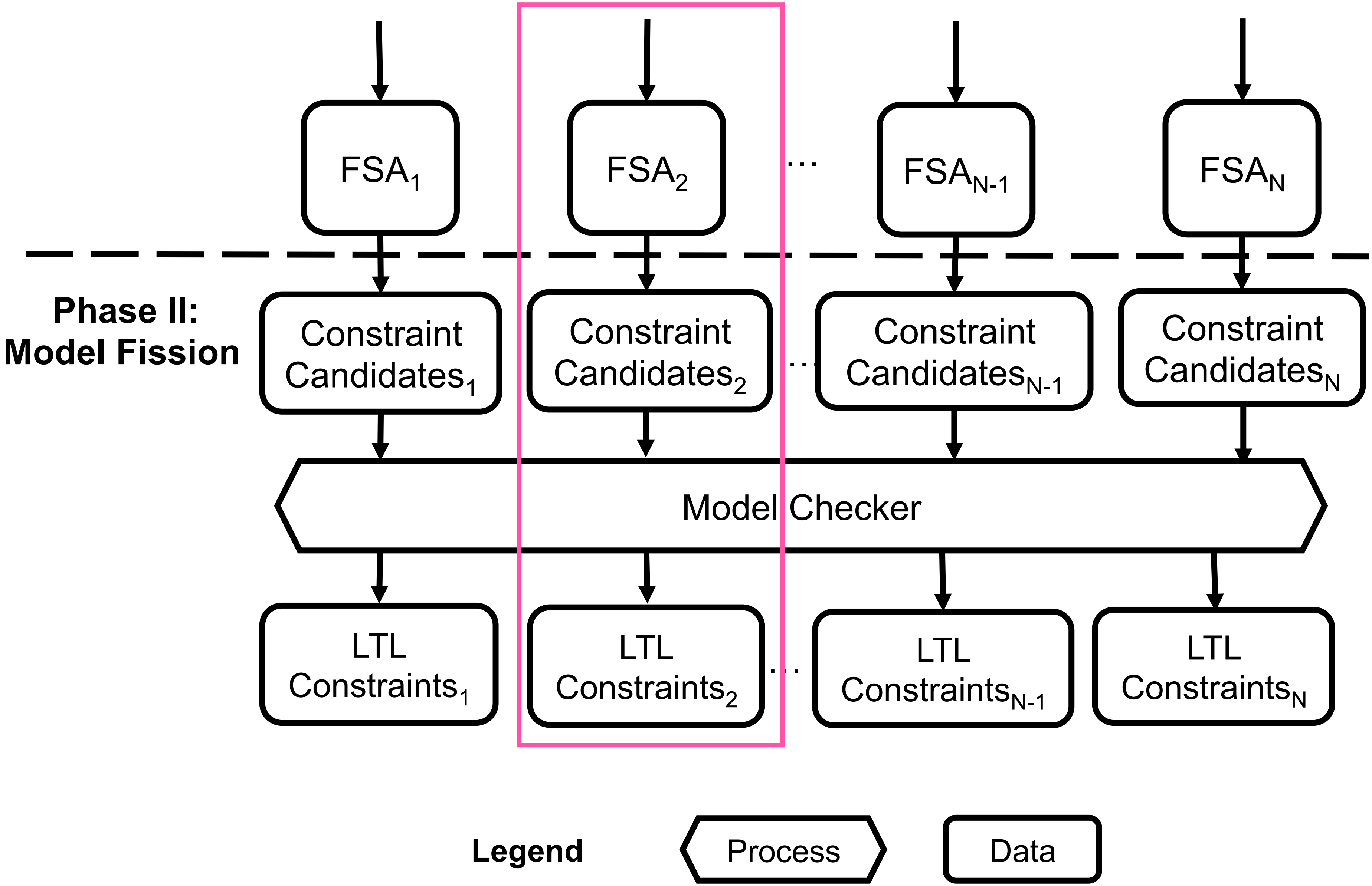
  c b b b ✓ c a a b ⊗

The immediate LTL Constraint Types

- AIF(a,b): a is always *immediately* followed by b
- NIF(a,b): a is never *immediately* followed by b
- AIP(a,b): a is always *immediately* preceded by b

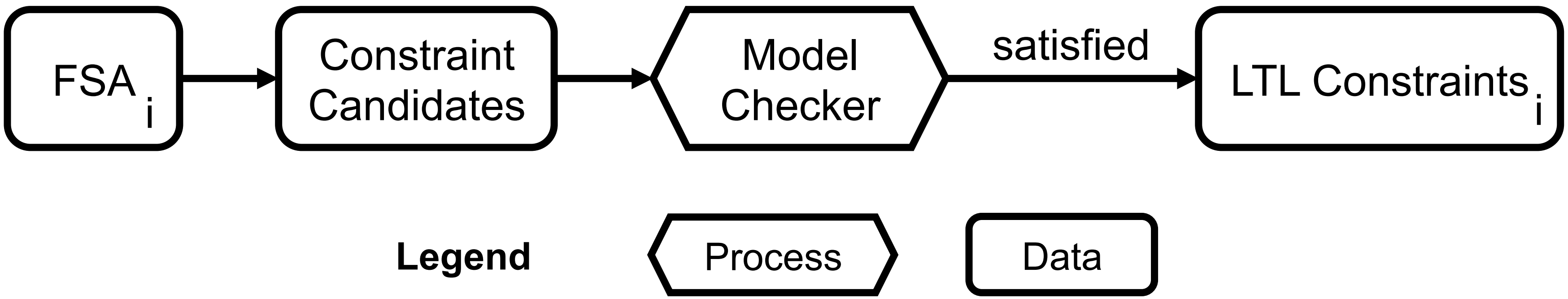
AIF, NIF, and AIP are extensions of AF, NF, and AP

Model Fission



$FSA_i \rightarrow$ LTL Constraints

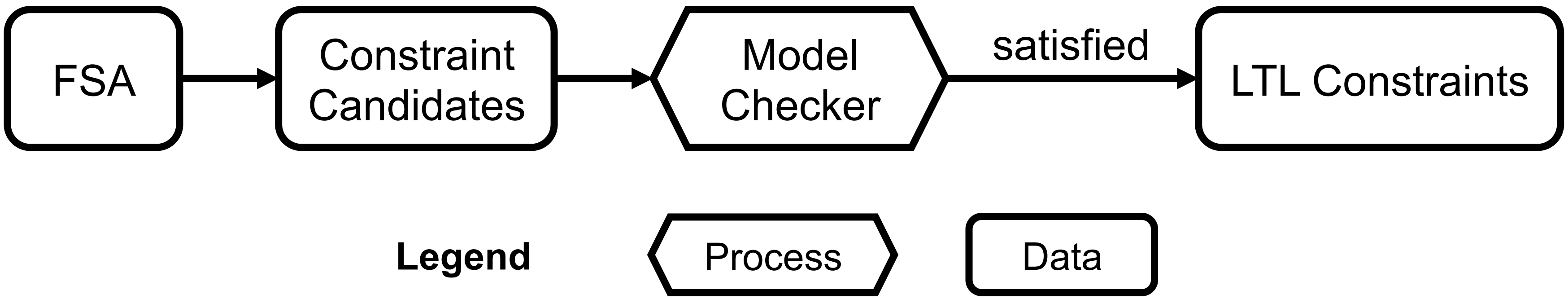
- For each constraint type
  - Enumerate constraint candidates (e.g., possible method call combinations)
  - Verify each candidate on FSA_i with a model checker
  - Retain just the constraints that hold in FSA_i



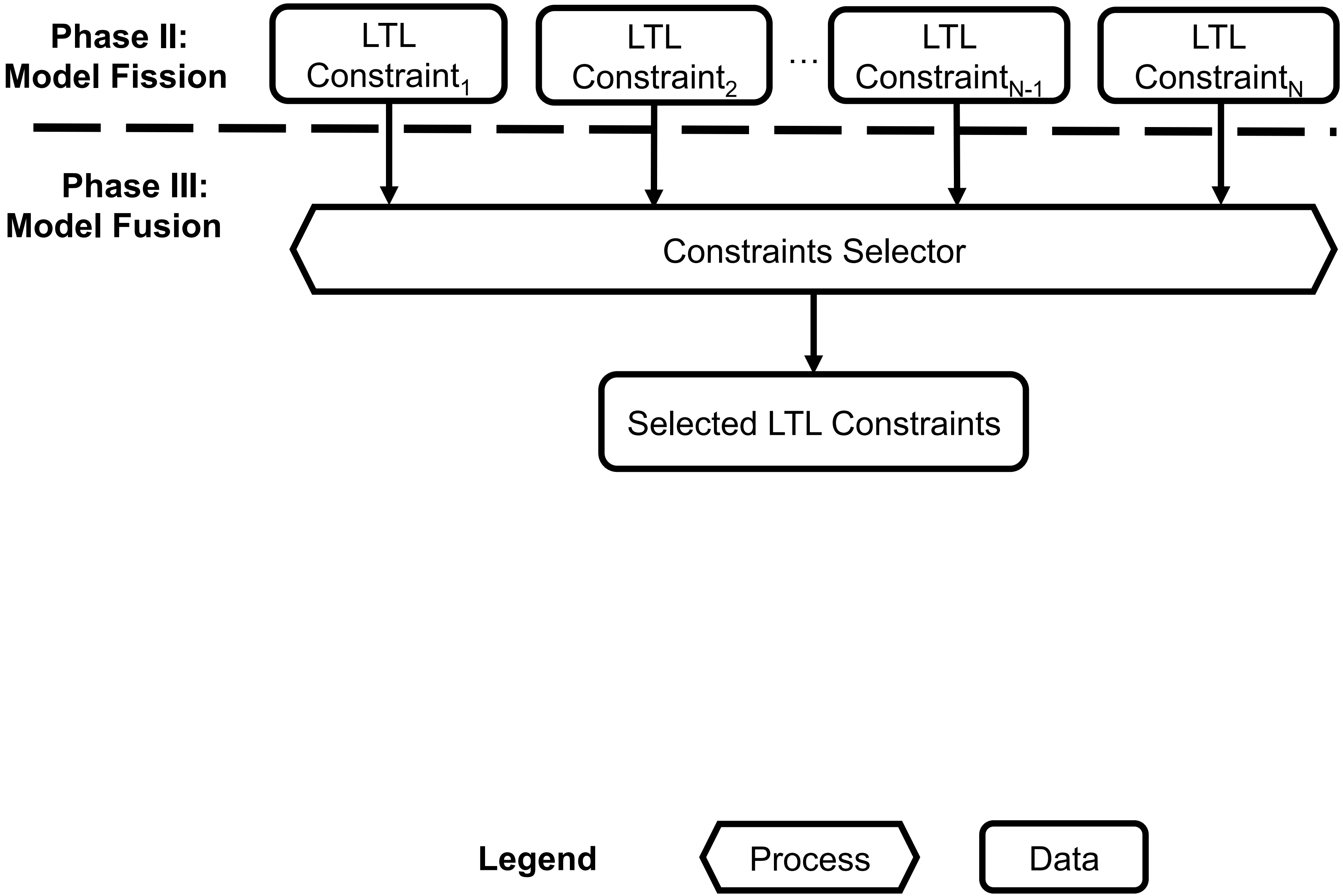
FSA → LTL Constraints

- Model checking is costly
- Define a time threshold when checking constraint candidates
  - Terminate SPIN if running time > threshold

☞ potentially miss important LTL constraints ☹



Phase 3: Model Fusion

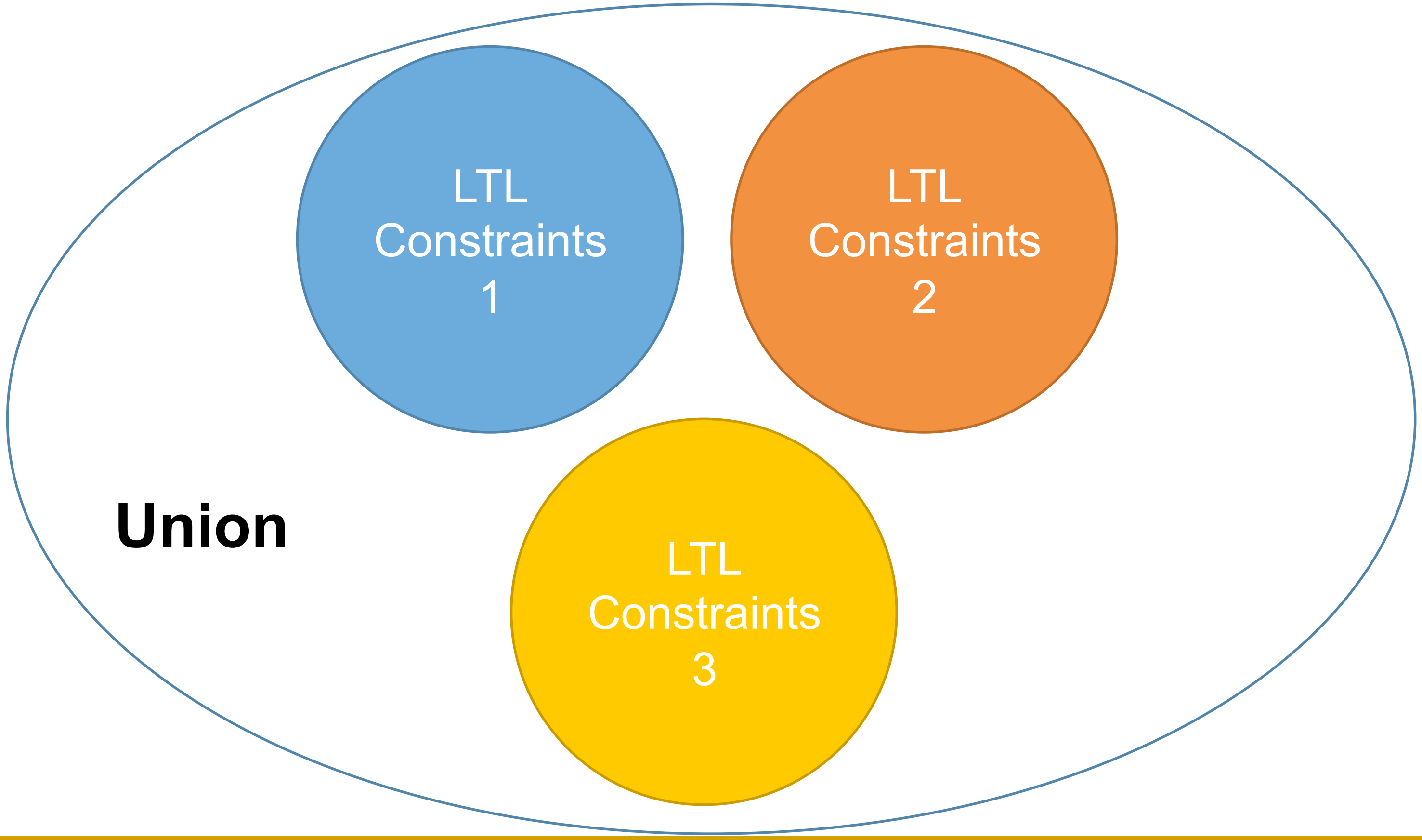


Selecting Constraints to Fuse

- Select subset of LTL constraints
  - These determine the final SpecForge model
- Unclear which constraints work best
- We propose 4 heuristics
  - union
  - majority
  - satisfied by ≥ x
  - intersection

Constraint Selection

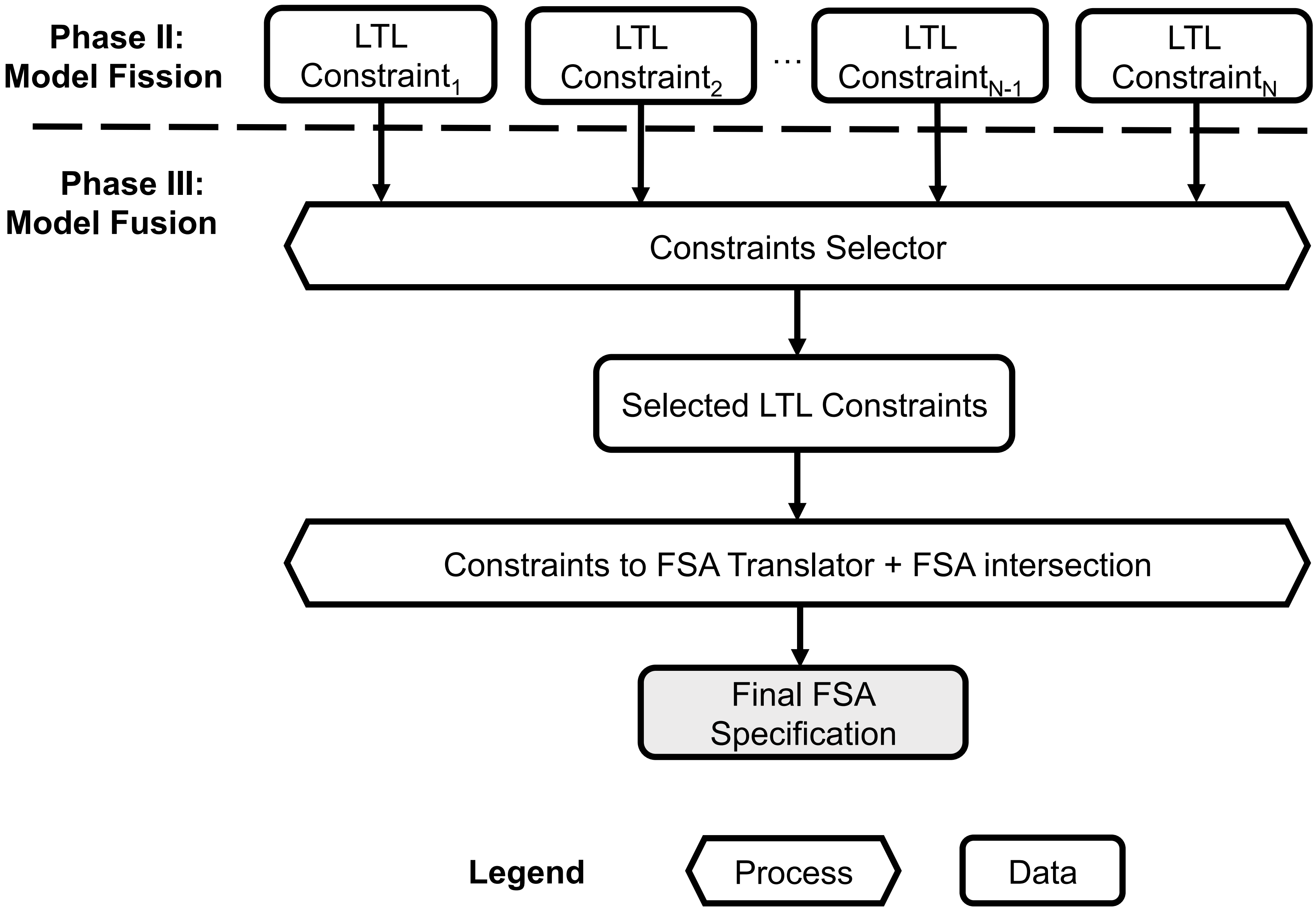
- Union
  - Assume all LTL constraints are correct
  - Returns all LTL constraints of all miners



Constraint Selection

- Satisfied by ≥ `x`
  - Select LTL constraints that satisfy at least `x` FSAs inferred by `x` miners.
- Majority
  - Assume correct LTL constraints satisfy *majority* of FSAs
  - ~ Satisfied by
- Intersection
  - Assume correct LTL constraints satisfy *all* of FSAs
  - ~ Satisfied by `N`

Model Fusion

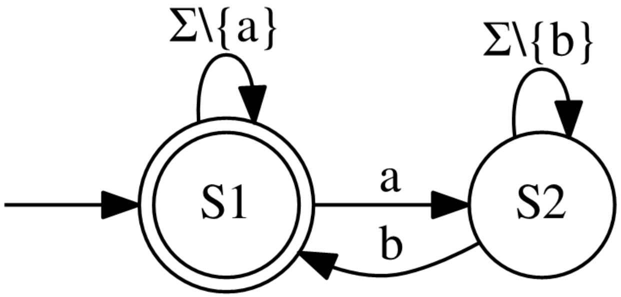


LTL Constraints → FSA

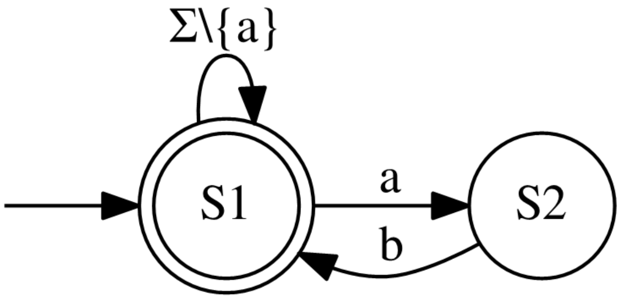
- Convert each constraint into an FSA
  - Each FSA has two events (e.g., `a` and `b`) in a given alphabet $\sum$
  - Each constraint type has its own way to construct the FSA

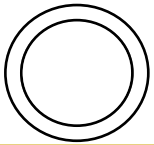
LTL Constraints → FSA

- `AF(a,b)`: `a` is always followed by `b`



- `AIF(a,b)`: `a` is always immediately followed by `b`



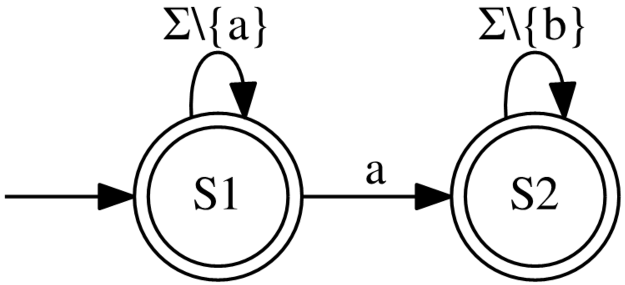


Final state

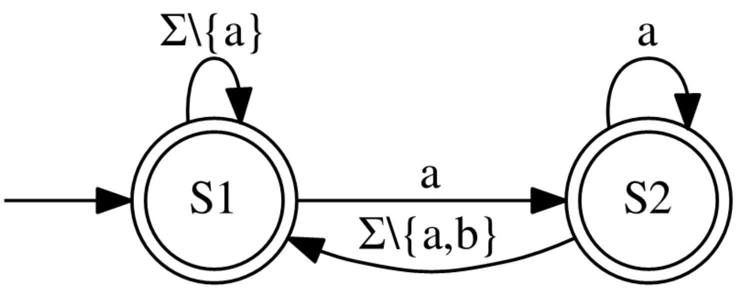
∑: alphabet (i.e., set of method calls might occur in execution traces)

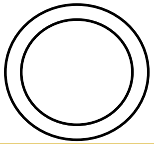
LTL Constraints → FSA

- `NF(a,b)`: `a` is never followed by `b`



- `NIF(a,b)`: `a` is never immediately followed by `b`



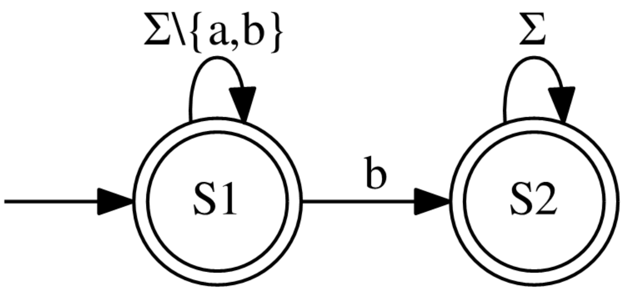


Final state

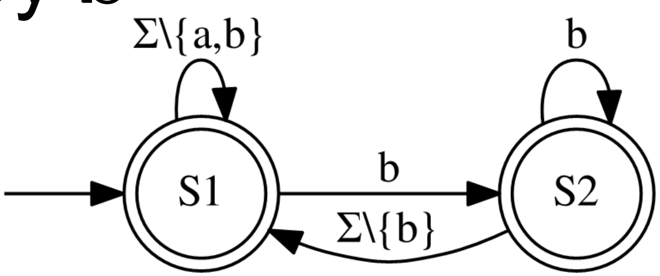
∑: alphabet (i.e., set of method calls might occur in execution traces)

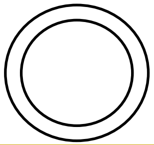
LTL Constraints → FSA

- `AP(a,b)`: `a` is always preceded by `b`



- `AIP(a,b)`: `a` is always immediately preceded by `b`





Final state

∑: alphabet (i.e., set of method calls might occur in execution traces)

LTL Constraints → FSA

- LTL Constraints → constraint FSAs
- Final model = intersection of constraint FSAs
  - Final FSA satisfies all of the selected LTL constraints

SpecForge summary:

1. Run each spec miner on traces
2. Decompose generated models with fission
3. Build new model using fusion

Evaluation Research Questions

1. How effective is SpecForge?
2. Does SpecForge improve over existing spec miners?
3. What is the impact of constraint templates on model quality?
4. What is the impact of constraint selection heuristic on model quality?

Dataset [13 library classes]

Target Library Classes	Client Programs
`java.util.ArrayList`	Dacapo fop
`java.util.HashMap`	Dacapo h2
`java.util.HashSet`	Dacapo h2
`java.util.Hashtable`	Dacapo xalan
`java.util.LinkedList`	Dacapo avrora
`java.util.StringTokenizer`	Dacapo batik
`org.apache.xalan.templates.ElemNumber $NumberFormatStringTokenizer`	Dacapo xalan
`DataStructures.StackAr`	StackArTester
`java.security.Signature`	Columba, jFTP
`org.apache.xml.serializer.ToHTMLStream`	Dacapo xalan
`java.util.zip.ZipOutputStream`	JarInstaller
`org.columba.ristretto.smtp.SMTPProtocol`	Columba
`java.net.Socket`	Voldemort

Dataset

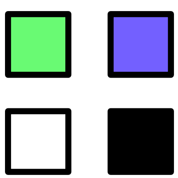
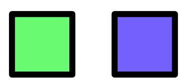
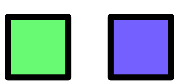
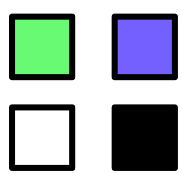
- Execution traces generated by client program tests, paired with Daikon invariants
- Ground-truth models
  - Krka et al. [1]
  - Pradel et al. [2]
  - ☞ Manually improved ground-truth models

[1] Krka, Y. Brun, and N. Medvidovic, "Automatic mining of specifications from invocation traces and method invariants,"FSE 2014

[2] M. Pradel, P. Bichsel, and T. R. Gross, "A framework for the evaluation of specification miners based on finite state machines," ICSM 2010

Evaluation Metrics

- **Precision**: fraction of *inferred model* traces that are accepted by *the ground truth model*
- **Recall**: fraction of *ground truth* traces that are accepted by *the inferred model*
- **F-measure**: 2 x (Precision x Recall) / (Precision + Recall)

Inferred FSA traces	Ground truth traces	Precision	Recall	F-mesure
		$\frac{2}{4}$	$\frac{2}{2}$	$\frac{2}{3}$
		$\frac{2}{2}$	$\frac{2}{4}$	$\frac{2}{3}$

[1] David Lo and Siau-Cheng Khoo, "Smartic: towards building an accurate, robust and scalable specification miner", FSE 2006

Default Configuration

- We use all of the 6 constraint types
  - AF, AIF, NF, NIF, AP, and AIP
- *Intersection* heuristic for constraint selection
- Trace generation
  - Each FSA edge covered by at least 10 traces
  - Limit number of traces to 10K per library
  - Limit trace length to 100 transitions

Baseline Specification Miners

- Traces-only
  - Traditional 1-tails & Traditional 2-tails [1]
- Invariants-only
  - CONTRACTOR++ [2]
- Invariant-Enhanced-Traces
  - SEKT 1-tails & SEKT 2-tails [2]
- Trace-Enhanced-Invariants
  - Optimistic TEMI & Pessimistic TEMI [2]

[1] A. W. Biermann and J. A. Feldman, “On the synthesis of finite- state machines from samples of their behavior,” IEEE Transactions on Computers,1972

[2] Krka, Y. Brun, and N. Medvidovic, “Automatic mining of specifications from invocation traces and method invariants,”FSE 2014

RQ1: SpecForge's Effectiveness

Target Class Library	Precision	Recall	F-measure
ArrayList	100.00%	65.08%	78.85%
HashMap	100.00%	44.02%	61.13%
HashSet	100.00%	55.44%	71.33%
Hashtable	100.00%	44.11%	61.22%
LinkedList	100.00%	82.80%	90.59%
StringTokenizer	60.00%	74.15%	66.33%
NFST	92.00%	30.63%	45.96%
SMTPProtocol	93.73%	45.00%	60.81%
Signature	100.00%	24.32%	39.13%
Socket	77.07%	40.86%	53.41%
StackAr	54.62%	100.00%	70.65%
ToHTMLStream	100.00%	60.00%	75.00%
ZipOutputStream	100.00%	43.18%	60.32%
Average	**90.57%**	**54.58%**	**64.21%**

RQ2: SpecForge vs. Baselines

Approach	Avg. Precision	Avg. Recall	Avg. F-measure
Traditional 1-tails	92.26%	17.38%	27.22%
Traditional 2-tails	93.58%	14.08%	23.44%
CONTRACTOR++	**95.59%**	**49.17%**	**56.45%**
SEKT 1-tails	96.86%	15.45%	25.43%
SEKT 2-tails	96.98%	13.77%	23.18%
Optimistic TEMI	95.07%	47.74%	54.93%
Pessimistic TEMI	97.92%	31.67%	38.94%
SpecForge	90.57%	54.58%	64.21%

- Hints at the underlying trade-offs between spec miners
- SpecForge has the best recall and F-measure

RQ3: Different LTL Constraints

Constraint	Avg. Precision	Avg. Recall	Avg. F-measure
ALL(default)	90.57%	54.58%	64.21%
ALL - AF	87.58%	60.52%	68.21%
ALL - NF	90.68%	54.98%	64.83%
ALL - AP	15.01%	54.58%	21.36%
ALL - AIF	90.73%	54.58%	64.33%
ALL - NIF	86.60%	62.62%	66.71%
ALL - AIP	89.85%	63.22%	70.75%
AF + NF + AP	83.35%	71.82%	72.82%
AF + NF + AP + AIP	86.57%	62.62%	66.70%
AF + NF + AP + NIF	89.85%	63.22%	70.75%
AF + NF + AP + AIF	83.35%	71.82%	72.82%
AIF + NIF + AIP	14.44%	60.92%	21.94%

- Constraint types really matter

RQ4: Different Constraint Selection Heuristics

Selection Heuristic	Precision	Recall	F-measure
Union	56.19%	10.26%	15.40%
Satisfied by x>= 2	78.51%	12.01%	18.36%
Satisfied by x>= 3	83.62%	17.81%	25.36%
Majority	93.00%	20.24%	28.98%
Satisfied by x>= 5	89.80%	34.98%	45.34%
Satisfied by x>= 6	88.82%	48.56%	59.48%
Intersection (default)	90.57%	54.58%	64.21%

- Union is too permissive (terrible Recall)
- Intersection is most constraining (best Recall and F-measure)
    - Conservative: do not admit a property from one spec miner unless it is validates by others

Advantages

- Transparently combines FSA spec miners
- Trivial to extend with new spec miners, LTL constraints and selection heuristics

Limitations

- Deals with the end-result; does not reason about internals of the spec miners
- Complex to tune
  - Spec miners
  - LTL constraint types
  - selection heuristic

Contributions

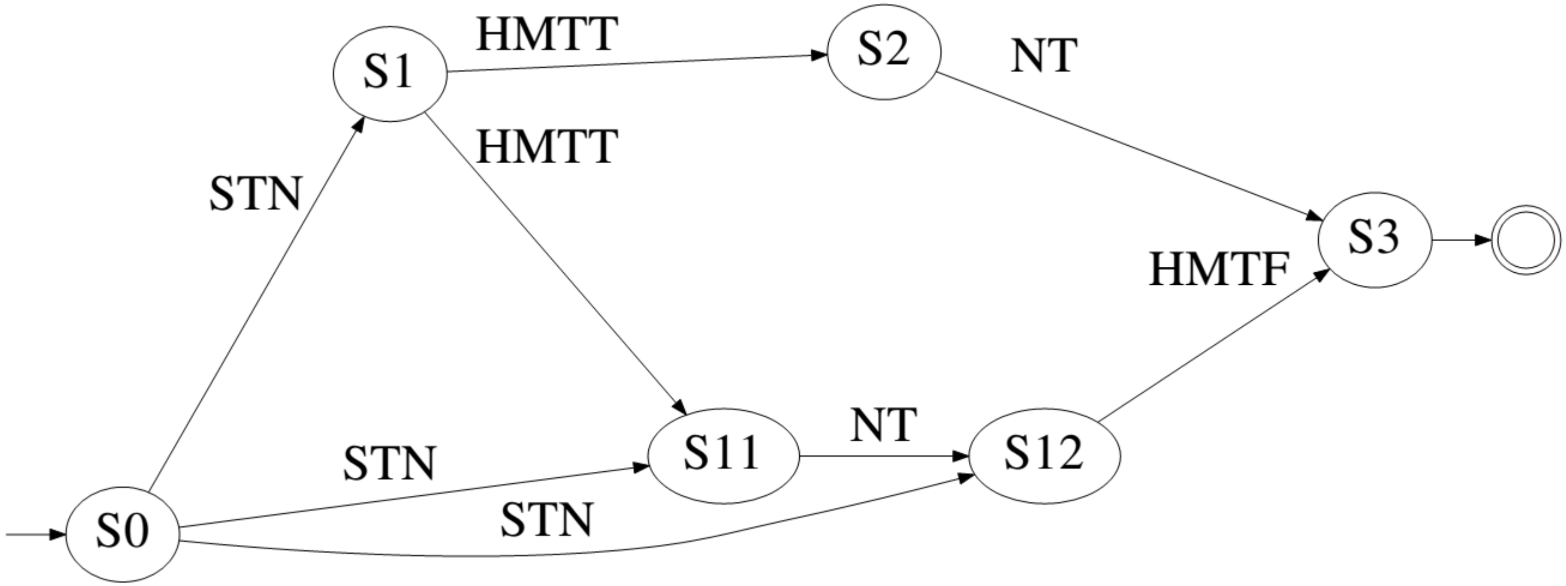
Proliferation of specification miners

SpecForge: a hybrid miner

- Introduced SpecForge to combine strengths of existing FSA specification miners
  - *Key techniques: model fission* and *fusion*
- Applied SpecForge to 13 lib classes and 7 spec miners
- SpecForge outperforms the best baseline by 14%

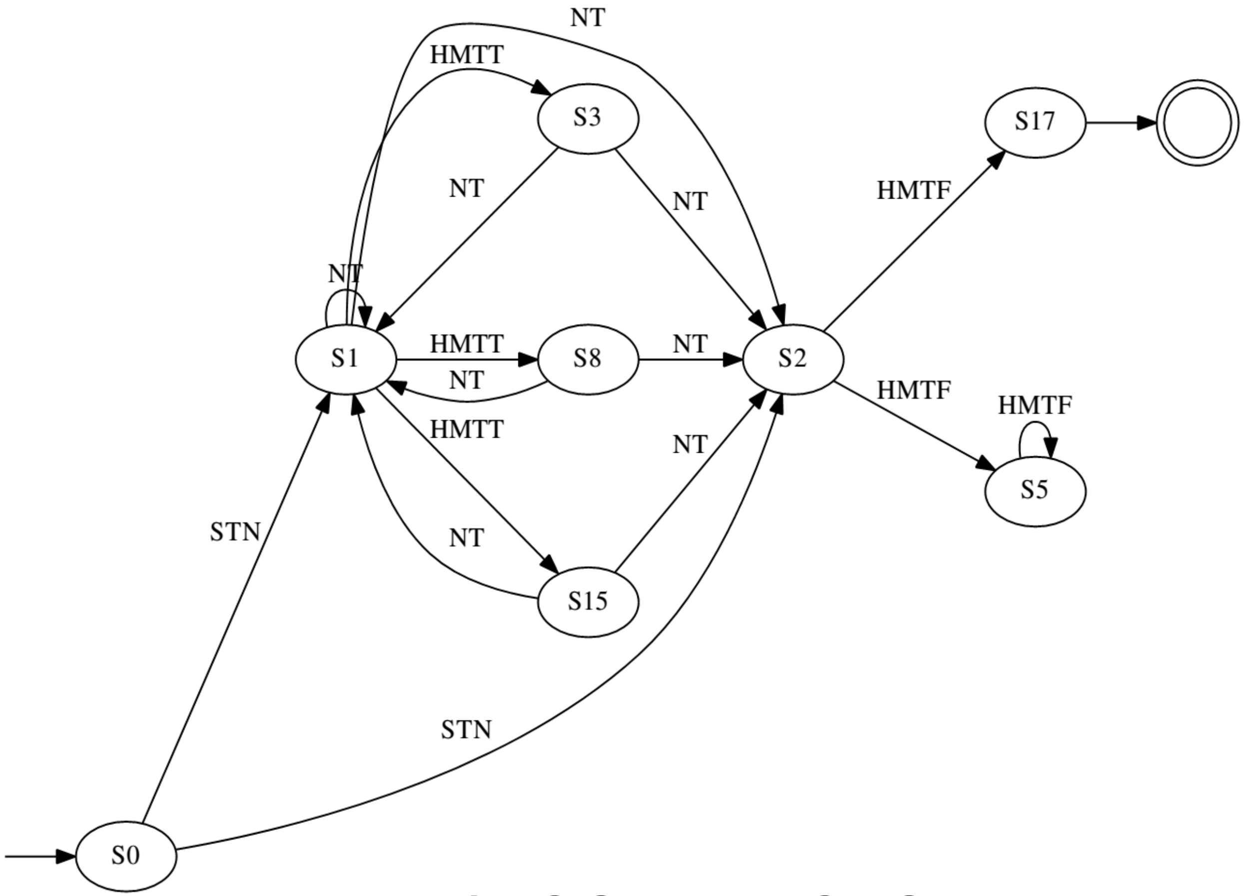
Motivating Example

- `java.util.StringTokenizer`
- k-tail (k=2)
- CONTRACTOR++



- `StringTokenizer`'s 2-tail model accepts execution traces that have
  - No repetitions of any methods ☒
  - No **NT** methods executed consecutively ☑

STN: `StringTokenizer()`	**NT**: `nextToken()`
HMTT: `hasMoreTokens() = true`	**HMTF**: `hasMoreTokens() = false`

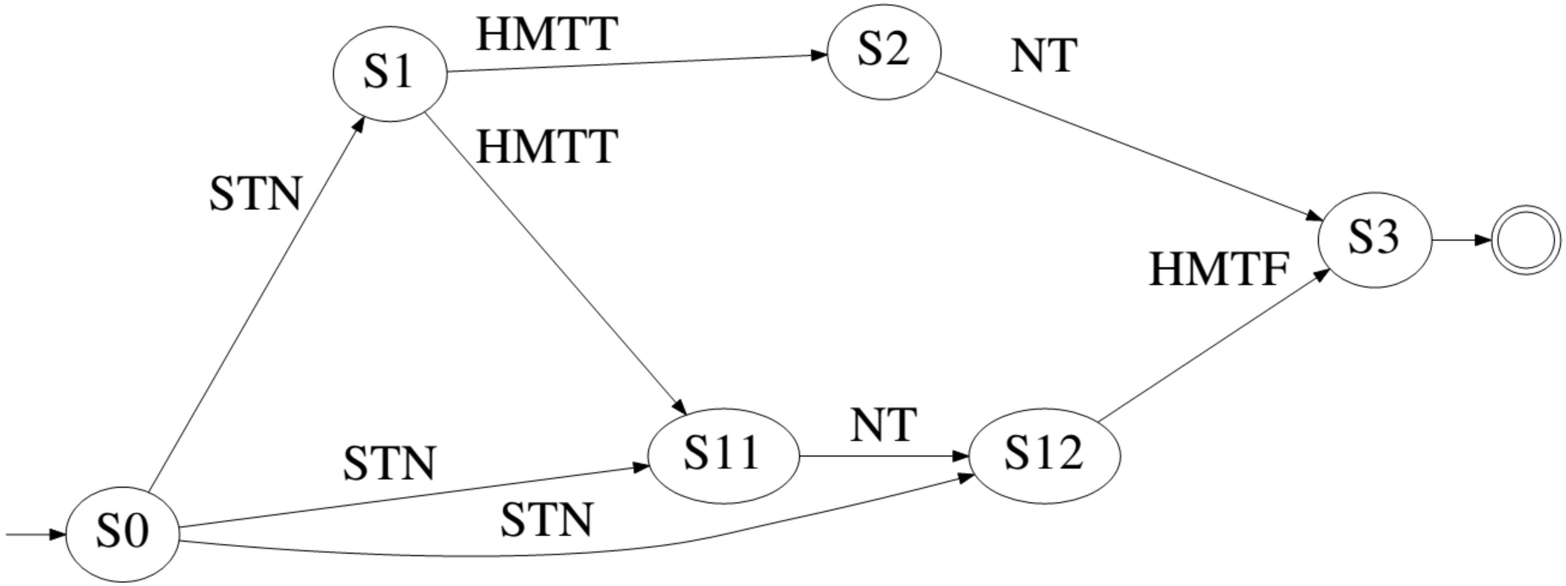


- `StringTokenizer`'s CONTRACTOR++ model
  - accepts traces that must end with **HMTF** ☒
  - allows `nextToken()` methods executed consecutively ☒
  - allows repetitions of methods ☑

STN: `StringTokenizer()`	**NT**: `nextToken()`
HMTT: `hasMoreTokens() = true`	**HMTF**: `hasMoreTokens() = false`

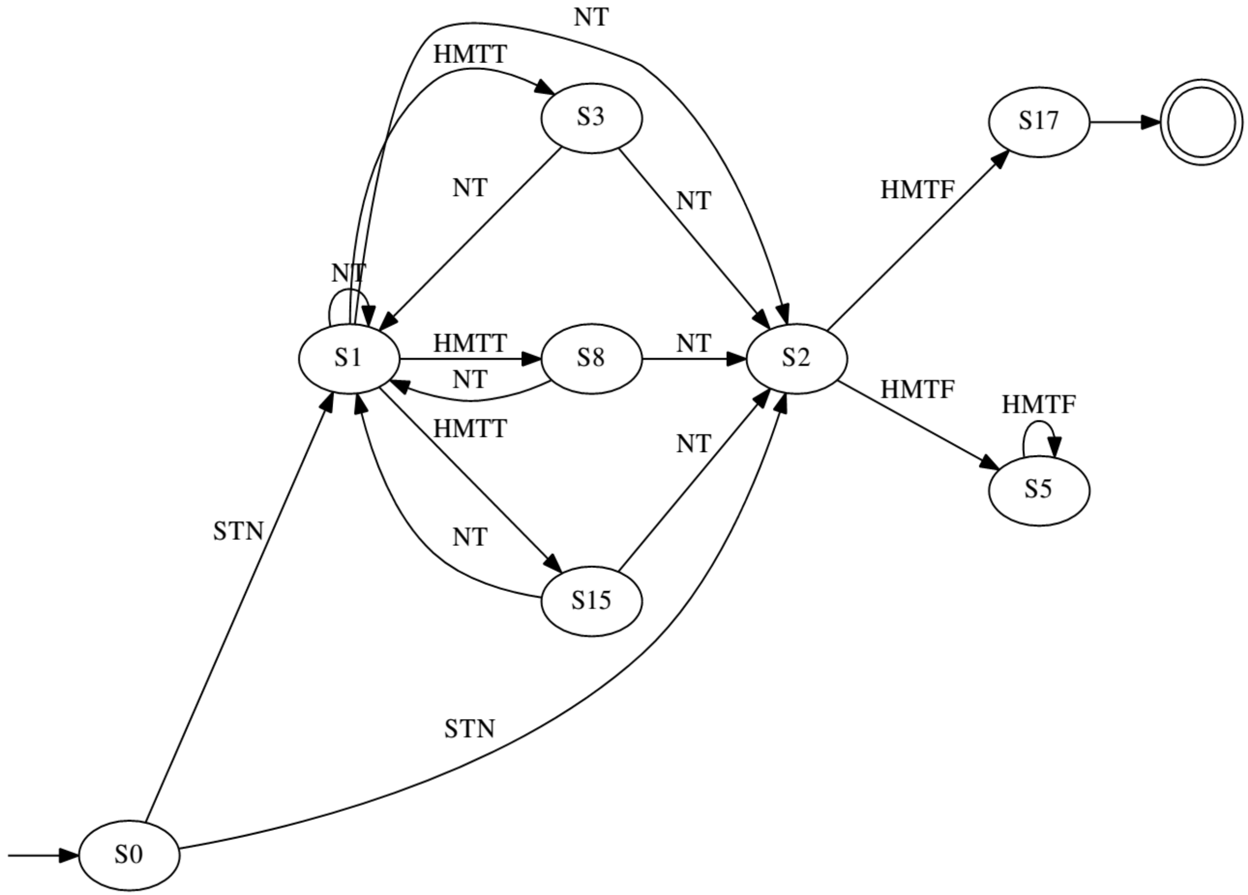
Motivating Example

- SpecForge
  - Model Fission



- Inferred Temporal Constraints
  - ☞`nextToken()` is never immediately followed by itself
  - ☞`hasMoreToken() = true` is never immediately followed by `hasMoreToken() = false`
  - ☞…

STN: `StringTokenizer()`
HMTT: `hasMoreTokens() = true`
NT: `nextToken()`
HMTF: `hasMoreTokens() = false`



- Inferred Temporal Constraints
  - ☞ `hasMoreTokens() = true` must be immediately followed by `nextToken()`
  - ☞ ...

STN: `StringTokenizer()` **NT**: `nextToken()`
HMTT: `hasMoreTokens() = true` **HMTF**: `hasMoreTokens() = false`

Motivating Example

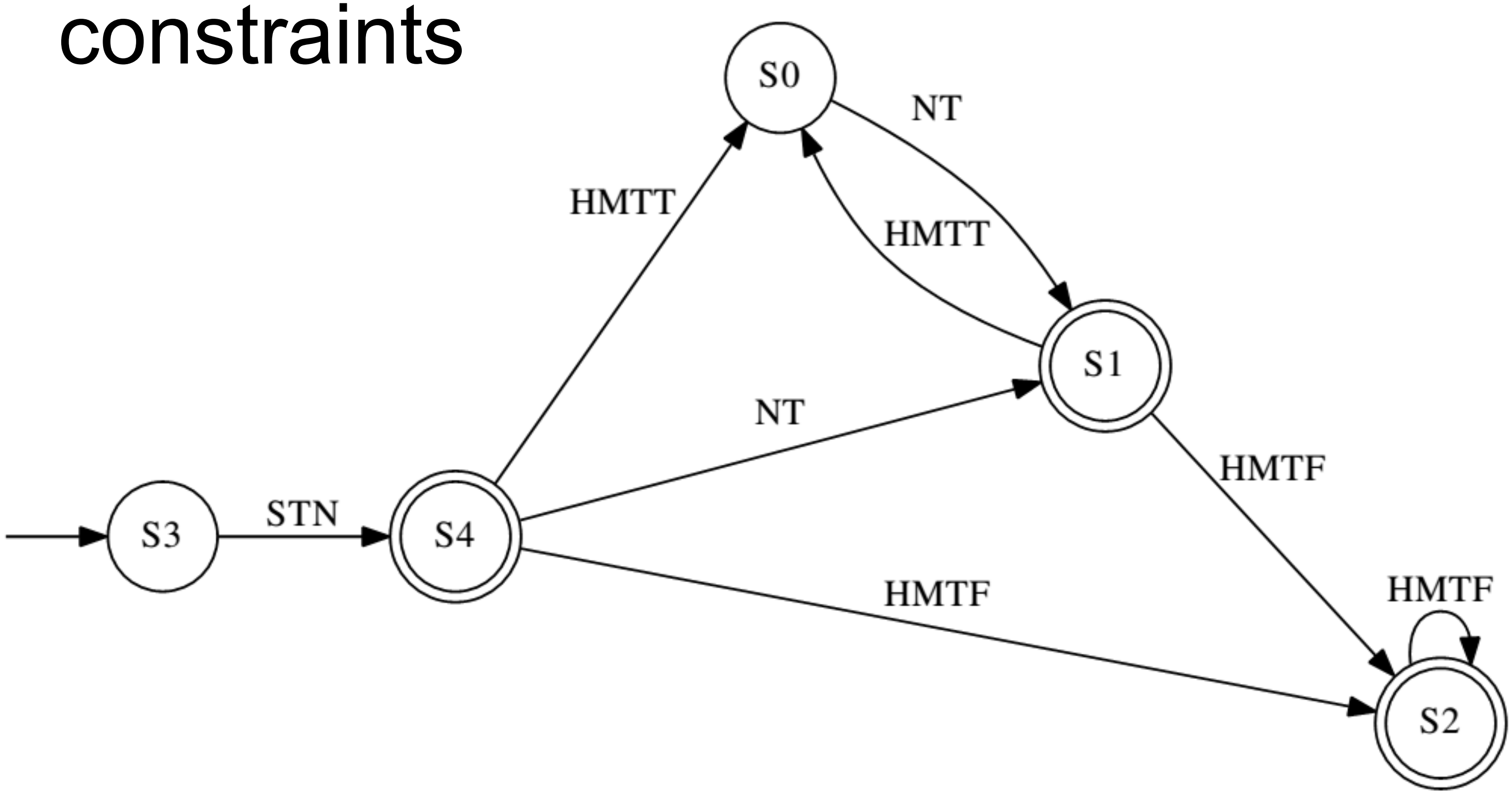
- SpecForge
  - Model Fission
  - Model Fusion

Motivating Example

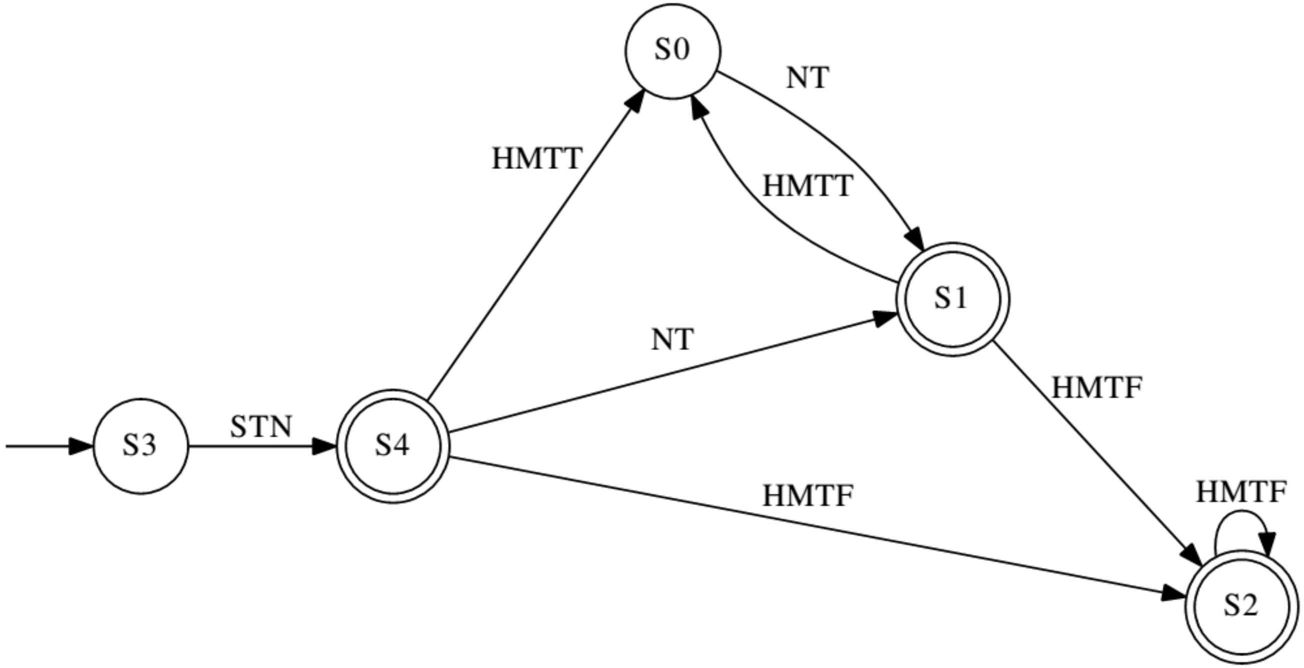
- Use a heuristic to select temporal constraints
  - **C1:** `nextToken()` is never immediately followed by itself
  - **C2:** `hasMoreToken() = true` is never immediately followed by `hasMoreToken() = false`
  - **C3:** `hasMoreTokens() = true` must be immediately followed by `nextToken()`
  - …
- `C1,C2` from 2-tail model improves limitations of CONTRACT++'s model
- `C3` from CONTRACT++'s model improves 2-tail model

Motivating Example

- Construct a FSA satisfies the selected constraints

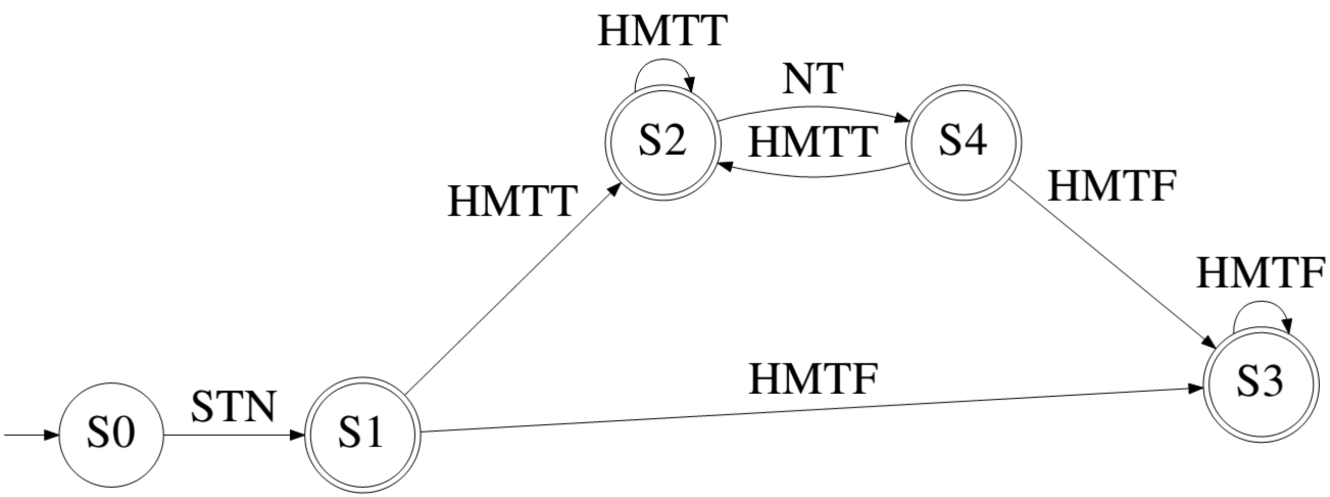


STN: `StringTokenizer()`
HMTT: `hasMoreTokens() = true`
NT: `nextToken()`
HMTF: `hasMoreTokens() = false`



SpecForge model

vs.



Ground-truth model

STN: `StringTokenizer()`
HMTT: `hasMoreTokens() = true`
NT: `nextToken()`
HMTF: `hasMoreTokens() = false`