Synergizing Specification Miners through Model Fissions and Fusions

Tien-Duy B. Le¹, Xuan-Bach D. Le¹, David Lo¹, Ivan Beschastnikh²

¹ Singapore Management University
² University of British Columbia
Synergizing Specification Miners through Model Fissions and Fusions

Tien-Duy B. Le¹, Xuan-Bach D. Le¹, David Lo¹, Ivan Beschastnikh²

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

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

Inferred with a spec miner

FSA model
Model Fission

Temporal constraint

Satisfied by the FSA model

FSA model

Temporal constraint
Model Fusion

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

FSA model'
SpecForge: Overall Framework

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

Legend:

- Process
- Data
Phase 2: Models Fission

• Decompose each $\text{FSA}_i$ 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

LTL Constraint Types

- **AF(a,b):** \textit{a} is always followed by \textit{b}
  - a b a b √
  - a b b a ×
  - c b b b √
  - c a a a ×

- **NF(a,b):** \textit{a} is never followed by \textit{b}
  - b b a a √
  - a b b a ×
  - a c a a √
  - c b a b ×

- **AP(a,b):** \textit{a} is always preceded by \textit{b}
  - b b a a √
  - a b b b ×
  - c b b b √
  - c a a a ×
LTL Constraint Types

• **AF(a,b):**  \( a \) is always followed by \( b \)
  
  \[
  \begin{array}{c}
  a \ b \ a \ b \ \checkmark \\
  a \ b \ b \ a \ \times
  \\
  c \ b \ b \ b \ \checkmark \\
  c \ a \ a \ a \ \times
  \end{array}
  \]

• **NF(a,b):**  \( a \) is never followed by \( b \)
  
  \[
  \begin{array}{c}
  b \ b \ a \ a \ \checkmark \\
  a \ b \ b \ a \ \times
  \\
  a \ c \ a \ a \ \checkmark \\
  c \ b \ a \ b \ \times
  \end{array}
  \]

• **AP(a,b):**  \( a \) is always preceded by \( b \)
  
  \[
  \begin{array}{c}
  b \ b \ a \ a \ \checkmark \\
  a \ b \ b \ b \ \times
  \\
  c \ b \ b \ b \ \checkmark \\
  c \ a \ a \ b \ \times
  \end{array}
  \]
LTL Constraint Types

• AF(a,b): \( a \) is always followed by \( b \)
  
  \[
  \begin{array}{ll}
  \text{a b a b } & \bigcirc \\
  \text{a b b a } & \bigotimes \\
  \text{c b b b } & \bigcirc \\
  \text{c a a a } & \bigotimes \\
  \end{array}
  \]

• NF(a,b): \( a \) is never followed by \( b \)
  
  \[
  \begin{array}{ll}
  \text{b b a a } & \bigcirc \\
  \text{a b b a } & \bigotimes \\
  \text{a c a a } & \bigcirc \\
  \text{c b a b } & \bigotimes \\
  \end{array}
  \]

• AP(a,b): \( a \) is always preceded by \( b \)
  
  \[
  \begin{array}{ll}
  \text{b b a a } & \bigcirc \\
  \text{a b b b } & \bigotimes \\
  \text{c b b b } & \bigcirc \\
  \text{c a a b } & \bigotimes \\
  \end{array}
  \]
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

Phase II: Model Fission

Legend

- Process
- Data

FSA\(_1\) → Constraint Candidates\(_1\) → LTL Constraints\(_1\)

FSA\(_2\) → Constraint Candidates\(_2\) → LTL Constraints\(_2\)

... → ...

FSA\(_{N-1}\) → Constraint Candidates\(_{N-1}\) → LTL Constraints\(_{N-1}\)

FSA\(_N\) → Constraint Candidates\(_N\) → LTL Constraints\(_N\)

Model Fission
For each constraint type
- Enumerate constraint candidates (e.g., possible method call combinations)
- Verify each candidate on \( \text{FSA}_i \) with a model checker
- Retain just the constraints that hold in \( \text{FSA}_i \)
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 ☹

![Diagram showing FSA, Constraint Candidates, Model Checker, and LTL Constraints with Process and Data Legend]
Phase 3: Model Fusion

Phase II: Model Fission

Phase III: Model Fusion

Legend

- Process
- Data
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 $\geq x$
  – intersection
Constraint Selection

- **Union**
  - Assume all LTL constraints are correct
  - Returns all LTL constraints of all miners
Constraint Selection

• Satisfied by $\geq x$
  – Select LTL constraints that satisfy at least $x$ FSAs inferred by $x$ miners.

• Majority
  – Assume correct LTL constraints satisfy majority of FSAs
  – $\sim$ Satisfied by

• Intersection
  – Assume correct LTL constraints satisfy all of FSAs
  – $\sim$ Satisfied by $N$
Model Fusion

Phase II: Model Fission
- LTL Constraint_1
- LTL Constraint_2
- \ldots
- LTL Constraint_{N-1}
- LTL Constraint_N

Phase III: Model Fusion
- Constraints Selector
- Selected LTL Constraints
- Constraints to FSA Translator + FSA intersection
- Final FSA Specification

Legend
- Process
- Data
LTL Constraints $\rightarrow$ 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 $\rightarrow$ FSA

- $\text{AF}(a, b): a$ is always followed by $b$

- $\text{AIF}(a, b): a$ is always immediately followed by $b$

$\Sigma$: alphabet (i.e., set of method calls might occur in execution traces)
LTL Constraints $\rightarrow$ FSA

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

  ![Diagram of NF](image)

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

  ![Diagram of NIF](image)

\[ \Sigma: \text{alphabet (i.e., set of method calls that might occur in execution traces)} \]

Final state
LTL Constraints $\rightarrow$ FSA

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

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

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

Final state
LTL Constraints $\rightarrow$ FSA

- LTL Constraints $\rightarrow$ 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]

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


## 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**: $2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$

<table>
<thead>
<tr>
<th>Inferred FSA traces</th>
<th>Ground truth traces</th>
<th>Precision</th>
<th>Recall</th>
<th>F-measure</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Inferred Traces" /></td>
<td><img src="image2" alt="Ground Truth Traces" /></td>
<td>$\frac{2}{4}$</td>
<td>$\frac{2}{2}$</td>
<td>$\frac{2}{3}$</td>
</tr>
<tr>
<td><img src="image3" alt="Inferred Traces" /></td>
<td><img src="image4" alt="Ground Truth Traces" /></td>
<td>$\frac{2}{2}$</td>
<td>$\frac{2}{4}$</td>
<td>$\frac{2}{3}$</td>
</tr>
</tbody>
</table>

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]


# RQ1: SpecForge’s Effectiveness

<table>
<thead>
<tr>
<th>Target Class Library</th>
<th>Precision</th>
<th>Recall</th>
<th>F-measure</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArrayList</td>
<td>100.00%</td>
<td>65.08%</td>
<td>78.85%</td>
</tr>
<tr>
<td>HashMap</td>
<td>100.00%</td>
<td>44.02%</td>
<td>61.13%</td>
</tr>
<tr>
<td>HashSet</td>
<td>100.00%</td>
<td>55.44%</td>
<td>71.33%</td>
</tr>
<tr>
<td>Hashtable</td>
<td>100.00%</td>
<td>44.11%</td>
<td>61.22%</td>
</tr>
<tr>
<td>LinkedList</td>
<td>100.00%</td>
<td>82.80%</td>
<td>90.59%</td>
</tr>
<tr>
<td>StringTokenizer</td>
<td>60.00%</td>
<td>74.15%</td>
<td>66.33%</td>
</tr>
<tr>
<td>NFST</td>
<td>92.00%</td>
<td>30.63%</td>
<td>45.96%</td>
</tr>
<tr>
<td>SMTPProtocol</td>
<td>93.73%</td>
<td>45.00%</td>
<td>60.81%</td>
</tr>
<tr>
<td>Signature</td>
<td>100.00%</td>
<td>24.32%</td>
<td>39.13%</td>
</tr>
<tr>
<td>Socket</td>
<td>77.07%</td>
<td>40.86%</td>
<td>53.41%</td>
</tr>
<tr>
<td>StackAr</td>
<td>54.62%</td>
<td>100.00%</td>
<td>70.65%</td>
</tr>
<tr>
<td>ToHTMLStream</td>
<td>100.00%</td>
<td>60.00%</td>
<td>75.00%</td>
</tr>
<tr>
<td>ZipOutputStream</td>
<td>100.00%</td>
<td>43.18%</td>
<td>60.32%</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>90.57%</strong></td>
<td><strong>54.58%</strong></td>
<td><strong>64.21%</strong></td>
</tr>
</tbody>
</table>
RQ2: SpecForge vs. Baselines

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

- Hints at the underlying trade-offs between spec miners
- SpecForge has the best recall and F-measure
### RQ3: Different LTL Constraints

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

- Constraint types really matter
RQ4: Different Constraint Selection Heuristics

<table>
<thead>
<tr>
<th>Selection Heuristic</th>
<th>Precision</th>
<th>Recall</th>
<th>F-measure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Union</td>
<td>56.19%</td>
<td>10.26%</td>
<td>15.40%</td>
</tr>
<tr>
<td>Satisfied by $x \geq 2$</td>
<td>78.51%</td>
<td>12.01%</td>
<td>18.36%</td>
</tr>
<tr>
<td>Satisfied by $x \geq 3$</td>
<td>83.62%</td>
<td>17.81%</td>
<td>25.36%</td>
</tr>
<tr>
<td>Majority</td>
<td>93.00%</td>
<td>20.24%</td>
<td>28.98%</td>
</tr>
<tr>
<td>Satisfied by $x \geq 5$</td>
<td>89.80%</td>
<td>34.98%</td>
<td>45.34%</td>
</tr>
<tr>
<td>Satisfied by $x \geq 6$</td>
<td>88.82%</td>
<td>48.56%</td>
<td>59.48%</td>
</tr>
<tr>
<td>Intersection (default)</td>
<td>90.57%</td>
<td>54.58%</td>
<td>64.21%</td>
</tr>
</tbody>
</table>

- 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 validated 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

- 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 ✓

<table>
<thead>
<tr>
<th>STN: StringTokenizer()</th>
<th>NT: nextToken()</th>
</tr>
</thead>
<tbody>
<tr>
<td>HMTT: hasMoreTokens() = true</td>
<td>HMTF: hasMoreTokens() = false</td>
</tr>
</tbody>
</table>
- **StringTokenizer's CONTRACTOR++ model**
  - accepts traces that must end with HMTF
  - allows `nextToken()` methods executed consecutively
  - allows repetitions of methods

<table>
<thead>
<tr>
<th>STN: StringTokenizer()</th>
<th>NT: nextToken()</th>
</tr>
</thead>
<tbody>
<tr>
<td>HMTT:.hasMoreTokens() = true</td>
<td>HMTF:.hasMoreTokens() = false</td>
</tr>
</tbody>
</table>
Motivating Example

• SpecForge
  – Model Fission
• Inferred Temporal Constraints

✓ `nextToken()` is never immediately followed by itself

✓ `hasMoreToken() = true` is never immediately followed by `hasMoreToken() = false`

✓ ...

\[ \text{STN: StringTokenizer()} \] \hspace{2cm} \text{NT: nextToken()}
\[ \text{HMTT: hasMoreTokens() = true} \] \hspace{2cm} \text{HMTF: hasMoreTokens() = false} \]
• Inferred Temporal Constraints

\( \text{hasMoreTokens()} = \text{true} \) must be immediately followed by \( \text{nextToken()} \)

\( \ldots \)

\textbf{STN}: StringTokenizer() \quad \textbf{NT}: \text{nextToken()}

\textbf{HMTT}: hasMoreTokens() = \text{true} \quad \textbf{HMTF}: hasMoreTokens() = \text{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
STN: StringTokenizer()
HMTT: hasMoreTokens() = true
NT: nextToken()
HMTF: hasMoreTokens() = false