Synergizing Specification Miners through Model Fissions and Fusions

Tien-Duy B. Le[1], Xuan-Bach D. Le[1], David Lo[1], and Ivan Beschastnikh[2]
[1]School of Information Systems
Singapore Management University, Singapore
[2]Department of Computer Science
University of British Columbia, Canada
{btdle.2012,dxb.le.2013,davidlo}@smu.edu.sg, bestchai@cs.ubc.ca

***Abstract*—Software systems are often developed and released without formal specifications. For those systems that are formally specified, developers have to continuously maintain and update the specifications or have them fall out of date. To deal with the absence of formal specifications, researchers have proposed techniques to infer the missing specifications of an implementation in a variety of forms, such as finite state automaton (FSA). Despite the progress in this area, the efficacy of the proposed specification miners needs to improve if these miners are to be adopted.**

We propose *SpecForge*, a new specification mining approach that synergizes many existing specification miners. SpecForge decomposes FSAs that are inferred by existing miners into simple constraints, through a process we refer to as model fission. It then filters the outlier constraints and fuses the constraints back together into a single FSA (i.e., model fusion). We have evaluated SpecForge on execution traces of 10 programs, which includes 5 programs from DaCapo benchmark, to infer behavioral models of 13 library classes. Our results show that SpecForge achieves an average precision, recall and F-measure of 90.57%, 54.58%, and 64.21% respectively. SpecForge outperforms the best performing baseline by 13.75% in terms of F-measure.

Keywords*—*Specification Mining, Synergizing Miners, Model Fission, Model Fusion

I. Introduction

The short time-to-market and rapid evolution of software has led to software systems and libraries that are released without any documented specifications. Even when a system includes formal specifications, these specifications may become quickly out of date as the software evolves [37]. Finally, developers often lack the necessary skill and motivation to write formal specifications, as this takes significant time and manual effort [14]. The unavailability of specifications negatively impacts the maintainability and reliability of systems. Without specifications developers find code comprehension more difficult, and software becomes more error-prone as bugs are introduced due to mistaken assumptions. Furthermore, without a formal specification, developers cannot take advantage of some state-of-the-art bug finding and testing tools that require formal specifications as an input [5], [28].

To address the unavailability of formal specifications, researchers have proposed a number of specification mining techniques. In this work, we focus on techniques that extract a model in the form of a finite state automaton (FSA) by analyzing the execution traces of systems or libraries of interest. Krka et al. [15], Beschastnikh et al. [2], [1], Lo et al. [8], [23], Mariani et al. [27], and many others have proposed FSA-based specification mining algorithms. However, despite the many studies that have inferred FSAs from execution traces of systems, work in this area must improve for these techniques to be adopted in practice. For example, FSAs inferred using the *k-tail* algorithm are usually inaccurate for execution traces containing methods that frequently co-occur in particular orders, but are not required to occur exactly in these orders [15].

In this work we propose **SpecForge**, an automated approach to synergize the many existing FSA-based specification mining algorithms. SpecForge first uses existing specification miners to infer a set of FSAs. It then uses these to generate a superior FSA. SpecForge first performs *model fissions* to extract important constraints that are common across the mined FSAs. SpecForge then performs *model fusions* to combine the extracted constraints into one FSA model. Both model fission and model fusion processes are completely automated. In this work, we use a set of 6 constraint templates to generate constraints, some of which were proposed by Dwyer et al. [10] and Beschastnikh et al. [1]. SpecForge checks whether one or more instances of these constraint templates are observed in a mined model. Constraints corresponding to models generated by various specification miners are then merged together while the outlier constraints are identified and omitted.

We evaluated SpecForge on execution traces of 10 programs, including 5 programs from the DaCapo benchmark dataset. This benchmark includes programs and their test cases. We use the test cases to generate execution traces for our evaluation. Our experiments demonstrate that SpecForge can achieve an average precision, recall, and F-measure of 90.57%, 54.58%, and 64.21%, respectively. Compared to the constituent approaches that we combine together, our proposed approach improves on their F-measure, which is the harmonic mean of precision and recall, by at least 13.75%. We have also experimented with manual tuning of SpecForge parameters, which produced a best-case SpecForge precision, recall, and F-measure of 83.35%, 71.82%, and 72.82% respectively.

To summarize, we make the following three contributions:

1) We propose SpecForge, a new research direction to synergize multiple specification mining algorithms to mine better specifications. This meta-approach to model inference takes advantage of the extensive prior work on developing new model inference algorithms.
2) We propose a novel approach to synergize multiple spec-

ification mining algorithms by performing model fissions and fusions. Each FSA generated by the multiple specification miners is broken down into simple constraints that characterize the FSA. These sets of constraints are then analyzed to form the final set of constraints which are then fused together to create the final FSA.

3) We have evaluated the effectiveness of SpecForge to infer specifications of 13 library classes from execution traces of 10 different programs. Our experimental results show that SpecForge can achieve a reasonably high average precision, recall, and F-measure of 90.57%, 54.58%, and 64.21%. The F-measure of our approach outperforms the best-performing baselines by 13.75%.

The structure of the remainder of this paper is as follows. In Section II we describe a number of existing FSA-based specification mining algorithms that we synergize in this work. In Section III, we present an example that illustrates the power of SpecForge. In Section IV we describe SpecForge. The experimental settings and evaluation results are detailed in Section V. We discuss related work in Section VI and conclude and describe future work in Section VII.

II. Existing FSA-Based Specification Miners

This section briefly introduces a number of existing specification mining algorithms from prior work. SpecForge is built on top of the specification mining algorithms described below.

k-tails: k-tails is a classic algorithm proposed by Biermann and Feldman [3] to infer a FSA from execution traces. The algorithm takes as input a set of execution traces and a parameter k. To infer a FSA that describes the input execution traces, *k-tails* first builds a prefix tree acceptor (PTA) that accepts all of the input traces. A PTA is an automaton in the form of a tree, where every common prefix among the input traces corresponds to one state. Next, *k-tails* merges every two states of the PTA that have identical sequences of the next k method invocations (i.e., k-tails). The effectiveness of the *k-tails* algorithm depends the choice of k and the quality of its input traces. If the value of k is small, *k-tails* might lead to incorrect state merges. If the value of k is large, then there are fewer merges, which limits the generalization of the inferred specifications. Similarly, if the number of input traces is small, then the inferred FSA might not accurately capture the correct specifications. In this work, we are interested in two variants of *k-tails* with the value of $k \in \{1, 2\}$. We refer to these two variants as *traditional 1-tails* and *traditional 2-tails*.

CONTRACTOR++: CONTRACTOR++ is a recently proposed algorithm by Krka et al. [15] that uses inferred value-based program invariants to aid the construction of a FSA from execution traces. CONTRACTOR++ first runs Daikon [11] to infer several families of value-based program invariants; these include relational invariants (e.g., x $>$ 5), null invariants (e.g., x is null), and size invariants (e.g., x.size() $>$ 5). It then calls CONTRACTOR [9] which is able to construct a FSA from a set of invariants by running SMT solvers. CONTRACTOR characterizes each state in the constructed FSA by a set of methods that are enabled on that state. A legal state is a state in which the preconditions of the enabled methods are consistent with one another. CONTRACTOR creates a transition for a method from a source state to a target state if that method has both its precondition satisfied in the source state as well as its postcondition satisfied in the target state.

TABLE I: Execution Traces. "STN": StringTokenizer(). "HMTT": hasMoreTokens() = true. "HMTF": hasMoreTokens() = false. "NT": nextToken().

ID	Execution Trace
T1	STN - HMTT - NT - HMTF
T2	STN - HMTT - NT - HMTT - NT - HMTF
T3	STN - HMTT
T4	STN - HMTF
T5	STN - HMTF - HMTF
T6	STN - HMTT - HMTT - NT - HMTF
T7	STN - NT - HMTF
T8	STN - HMTT - NT - NT - HMTF

SEKT: State-enhanced k-tails (SEKT) is another recently proposed algorithm by Krka et al. [15]. Similar to CONTRACTOR++, it also makes use of inferred value-based invariants to construct a better FSA from execution traces. SEKT first runs Daikon to infer value-based invariants and then runs a variant of k-tails [3] that utilizes the inferred invariants. Similar to k-tails, SEKT also requires that every two states that are merged together have the same sequences of the next k invocations. However, different from k-tail, SEKT also requires that the merged states must share the same value-based invariants. This additional merging requirement allows SEKT to avoid problematic merges. In our study, we consider two variants of SEKT with its parameter k set to 1 and 2. The two variants are referred to as *SEKT 1-tails* and *SEKT 2-tails*.

TEMI: Trace-enhanced MTS[1] inference (TEMI) is yet another recently proposed algorithm by Krka et al. [15]. TEMI has two main phases. In the first phase, TEMI runs an algorithm similar to CONTRACTOR++ to build a FSA. It considers transitions in the FSA built in the first phase as *maybe* transitions. In the second phase, TEMI converts *maybe* transitions that are observed in the execution traces to *required* transitions. TEMI has two variants: optimistic (it outputs all maybe and required transitions) and pessimistic (it outputs only required transitions).

III. Motivating Example

This section provides an example that illustrates the power of SpecForge in improving the specifications learned by several other specification miners.

Figures 1 and 2 demonstrate the behavior models that traditional 2-tails [15] and CONTRACTOR++ [15] learn for java.util.StringTokenizer from execution traces that are collected during the execution of Dacapo batik [4]. There are only three methods of StringTokenizer that are invoked by Dacapo batik. They are StringTokenizer() (i.e., the constructor), hasMoreToken(), and nextToken(). Therefore, the output FSAs only include interactions among these three methods.

Table I shows a list of correct traces that should be accepted by correct models (T1-T7) and one trace that should be rejected by correct models (T8) for illustration purpose. Notice that the model inferred by traditional 2-tails (in Figure 1) accepts T1, T4, T7, but not T2, T3, T5, T6, and T8. This model is

[1]Modal Transition System

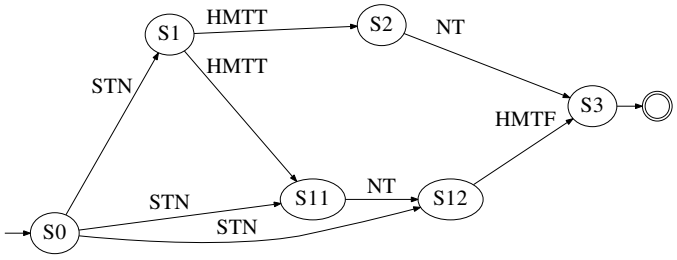

Fig. 1: `StringTokenizer`'s Traditional 2-tail Model. "STN": StringTokenizer(). "HMTT": hasMoreTokens() = true. "HMTF": hasMoreTokens() = false. "NT": nextToken().

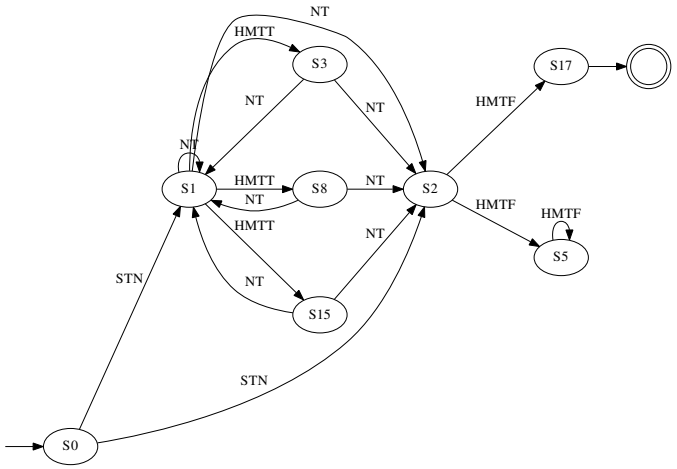

Fig. 2: `StringTokenizer`'s CONTRACTOR++ Model. "STN": StringTokenizer(). "HMTT": hasMoreTokens() = true. "HMTF": hasMoreTokens() = false. "NT": nextToken().

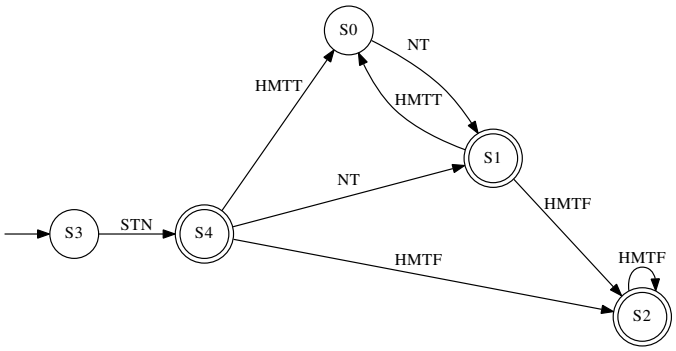

Fig. 3: `StringTokenizer`'s SpecForge Model. "STN": StringTokenizer(). "HMTT": hasMoreTokens() = true. "HMTF": hasMoreTokens() = false. "NT": nextToken().

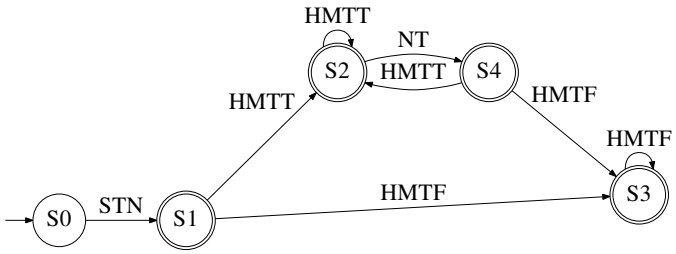

Fig. 4: `StringTokenizer`'s Ground Truth Model. "STN": StringTokenizer(). "HMTT": hasMoreTokens() = true. "HMTF": hasMoreTokens() = false. "NT": nextToken().

restricted to accepting traces that contain no repetitions of any methods. On the other hand, the CONTRACTOR++ model (in Figure 2) is less restrictive, as it accepts T1, T2, T4, T7, T8, but not T3, T5, T6. However, the CONTRACTOR++ model inaccurately specifies that execution traces must end with an invocation of hasMoreToken() that returns false. In practice, developers can stop using StringTokenizer objects anytime, even if there are still extractable tokens. Another issue with the CONTRACTOR++ model is that it allows nextToken() to be invoked consecutively, which may cause an exception.

To handle the inaccuracies of existing miners, our solution, SpecForge, forges many specification miners together. It first uses model checking to decompose FSAs learned by various miners into simple temporal constraints. For example, from the traditional 2-tails model in Figure 1, SpecForge discovers that nextToken() is *never immediately followed by* itself, and hasMoreToken() returning true is *never immediately followed by* hasMoreToken() returning false. By using appropriate constraint selection heuristics (see Section IV-C), SpecForge can omit wrong, or poorly supported, constraints and retain those that are correct.

After the constraints have been selected, SpecForge can re-construct a FSA that accepts behaviors that do not violate any selected constraints. A sample FSA that is inferred by an instance of SpecForge built on top of the 7 specification miners described in Section II, which uses one of the constraint selection heuristics, is shown in Figure 3. This FSA accepts trace T1, T2, T3, T4, T5, T6, T7, but not T8 (see Table I). Compared to the models produced by traditional 2-tails and CONTRACTOR++, SpecForge model is more accurate. The SpecForge model closely approximates the ground truth model shown in Figure 4.

IV. Proposed Approach: SpecForge

A. Overall Architecture

Figure 5 illustrates the architecture of SpecForge. SpecForge takes as input a set of execution traces of an API and outputs a finite state automaton (FSA). SpecForge has three steps: (1) model construction, (2) model fission, and (3) model fusion.

In the model construction step, the input traces are fed as inputs to N different FSA-based specification miners. Each miner infers a FSA according to its underlying mining algorithm: $FSA_1, \ldots, FSA_N$. Many different specification mining algorithms have been proposed in the literature and in this work we focus on the $N = 7$ algorithms described in Section II.

Once the specification miners infer their respective FSAs, SpecForge unifies these FSAs into one model. First, each inferred FSA is deconstructed into a set of constraints (*model fission*). Based on some criteria, the *strongly supported* constraints are selected from this set. Finally, the selected constraints are fused to form the final specification (*model fusion*).

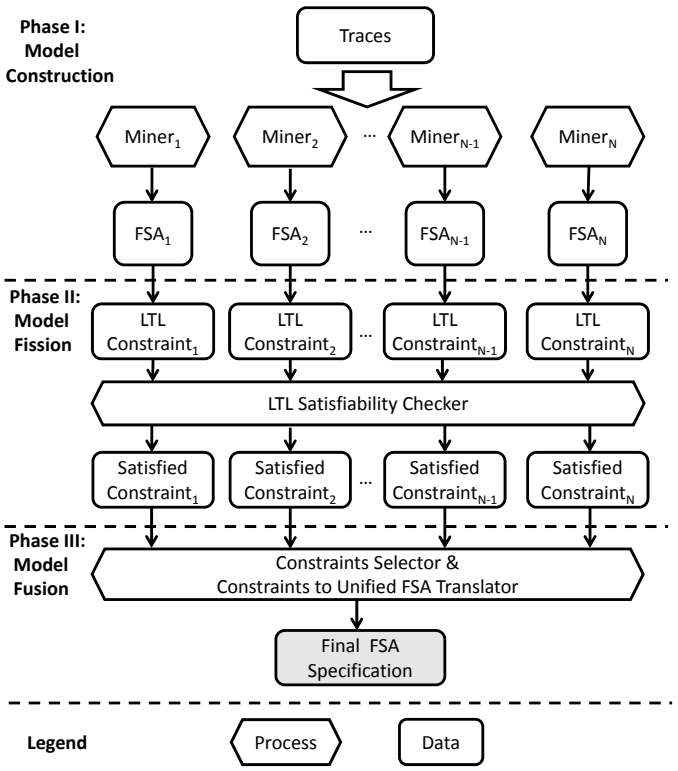

Fig. 5: SpecForge Overview

In the next two sections we further describe the model fission and model fusion steps.

B. Model Fission

The goal of this phase is to break a single FSA (e.g., FSA_i) into a set of basic building blocks that can be compared to blocks from other FSAs and used to build new FSAs. A key observation in our work is that a FSA can be thought of as a collection of *ordering constraints* among events. These ordering constraints can be shared by more than one FSA and are suitable building blocks for other FSAs. A classic formalism for specifying ordering constraints is Linear Temporal Logic (LTL) [31]. We use LTL in this work to specify ordering constraints between events.

The model fission process consists of two steps: constraint enumeration and constraint checking. In the first step, we generate a set of LTL constraint that may or may not be satisfied by the FSA. In the constraint checking step we filter out those LTL constraints that are not satisfied by the FSA.

Constraint Enumeration: It is impossible to check all possible LTL constraints. We therefore consider just the LTL constraints that fit the following six templates, each of which relates two events:

- a is **always followed by** b (denoted by AF(a, b): an occurrence of event a must be *eventually* followed by an occurrence of event b in the execution trace. In LTL, this rule is expressed as: $G(a \rightarrow XF\ b)$.
- a is **never followed by** b (denoted by NF(a, b)): there are *no* occurrences of event b after an occurrence of event a in the execution trace. In LTL, this rule is expressed as: $G(a \rightarrow XG(\neg b))$.
- a is **always preceded by** b (denoted by AP(a, b)): an occurrence of event a must be preceded by event b in the execution trace. In LTL, this rule is expressed as: $\neg a$ W b.
- a is **always immediately followed by** b (denoted by AIF(a, b)): an occurrence of event a must be *immediately* followed by an occurrence of event b in the execution trace. In LTL, this rule is expressed as: $G(a \rightarrow X\ b)$.
- a is **never immediately followed by** b (denoted by NIF(a, b)): there are no occurrences of event b immediately after any occurrence of event a in the execution trace. In LTL, this rule is expressed as: $G(a \rightarrow X(\neg b))$.
- a is **always immediately preceded by** b (denoted by AIP(a, b)): an occurrence of event a must be *immediately* preceded by an occurrence of event b in the execution trace. In LTL, this rule is expressed as: $F(a) \rightarrow (\neg a\ U(b \wedge Xa))$

Two of the six templates (i.e., always followed by, and always preceded by) correspond to two of the most commonly used LTL constraints (i.e., response and precedence) based on the survey by Dwyer et al. [10]. Another two templates (i.e., never followed by, and never immediately followed by) were introduced by Beschastnikh et al. [1] and have been demonstrated to be useful for describing FSA mining algorithms. The last two templates (i.e., always immediately followed by, and always immediately precedes) are newly introduced in this work. As a result, the bottom three templates are variations of the first three templates with the additional "immediately" requirement.

Given a set of execution traces, SpecForge enumerates all possible event pairs that appear in the traces. For each pair of events, e.g., a and b, we construct six possible LTL constraints corresponding to AF(a,b), NF(a,b), AP(a,b), AIF(a,b), NIF(a,b), and AIP(a,b). These constraints form the input to the constraint checking step.

Constraint Checking: This step checks the satisfiability of each of the generated LTL constraints in the enumeration step in the FSA model. For this checking we use the SPIN model checker [13], converting the FSA model into SPIN's Promela language. This process filters out those LTL constraints that are not satisfied by the FSA. At the end of this step the FSA is decomposed into a set of LTL constraints based on the six templates listed above; each constraint is satisfied by the FSA.

C. Model Fusion

The model fusion phase in SpecForge takes as input the sets of LTL constraints for each of the inferred FSAs. For the inferred FSAs $FSA_1, \ldots, FSA_N$, we denote the corresponding sets of LTL constraints as $C_1, \ldots, C_N$. That is, C_i is a set of constraints $\{C_{i1}, \ldots C_{iM_i}\}$ such that C_{ij} is based on one of the templates above and is satisfied by FSA_i.

The fusion process first selects LTL constraints from these input sets and then fuses the selected constraints into a new FSA. The fusion process contains the following three steps: (1)

constraint selection, (2) constraint to model translation, and (3) unified model construction. We described each of these steps below.

Constraint Selection: The goal of this step is to select a subset of LTL constraints from the sets of all input constraints. In this work, we consider the following four heuristics for selecting constraints:

- **Union**: This heuristic assumes that all of the generated constraints are correct and any one set is incomplete. It returns the union of all the constraint sets: $\cup_{1 \leq i \leq N} C_i$.
- **Majority**: Unlike the Union heuristic this heuristics assumes that some of the constraints are incorrect, but it assumes that those constraints that are in common across a majority of the constraint sets are correct. This heuristic returns the union of all constraints that are satisfied by the majority of the FSAs. Let num-containing(C_{ij}) be the number of input constraint sets containing C_{ij}. This heuristic returns the set $\{C_{ij} | \text{num-containing}(C_{ij}) \geq N/2\}$.
- **Satisfied By $\geq$ x**: This heuristics generalizes the above heuristics. We deem a constraint as correct if it is satisfied by at least x FSAs. For $x > N/2$ this heuristics is at least as strict as the Majority heuristic. Otherwise, it is more lenient. This heuristic returns the set $\{C_{ij} | \text{num-containing}(C_{ij}) \geq x\}$.
- **Intersection**: The final heuristic is the most conservative. It assumes that a correct constraint must have been satisfied by all inferred FSAs. It returns the set $\{C_{ij} | \text{num-containing}(C_{ij}) = N\}$.

Constraint to Model Translation: At the end of the previous step we have a set of selected constraints. In this step, we convert each constraint into a simple FSA (see Figure 7). Each simple FSA involves two distinct events in a given alphabet (e.g., a and b). Note that in Figure 7 not all rejecting states are shown for each FSA.

For example, Figure 7 (a) represents the FSA corresponding to the LTL constraint AF(a, b). In Figure 7 (a), accepting state is represented by a double circle and rejecting state is represented by a single circle. The initial state is $S1$ and whenever the event a happens, state $S2$ is entered. Next, whenever event b happens from state $S2$, state $S1$ is entered again. For example, this FSA accepts the sentence $aabab$.

In addition to the six simple FSAs in Figure 7, we consider one special FSA which describes the rule "a is never immediately followed by a" (the two events are the same event). In this case, we construct a FSA (see Figure 6) which is slightly different from Figure 7 (d).

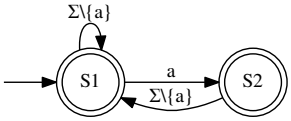

Fig. 6: FSA for "a is never immediately followed by a."

Unified Model Construction: In this step, SpecForge combines the constraint FSA models generated in the previous step into a unified FSA. Each model specifies a language or a set of execution traces that it accepts. We want the unified FSA to accept an intersection of these languages (i.e., a set of sentences in which each is accepted by *all* of the simple models). To construct such a unified FSA, SpecForge performs intersection over the FSAs corresponding to the selected constraints using the dk.bricks.automaton library [29]. The unified FSA will always be a connected FSA since it is always possible to represent a set of sentences as one connected FSA.

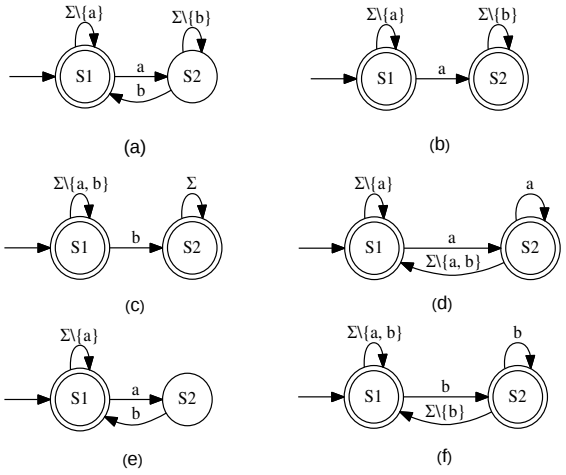

Fig. 7: Translations of LTL expressions to FSAs: (a) a is always followed by b, (b) a is never followed by b, (c) a is always preceded by b, (d) a is never immediately followed by b, (e) a is always immediately followed by b, (f) a is always immediately preceded by b.

TABLE II: List of Target Library Classes and Analyzed Programs.

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

V. Empirical Evaluation

In this section, we first describe our methodology in evaluating SpecForge against a number of baselines. We then describe four research questions and present our experimental results that answer these questions. We finish the section by discussing remaining, untapped potentials, of our approach and threats to validity.

A. Methodology

Target Library Classes: In this work, we evaluate the effectiveness of the SpecForge specification miner in generating behavioral models of 13 library classes. The list of 13 library classes is listed in Table II. The 7 underlying specification miners on top of which SpecForge is built require a set of execution traces to infer FSAs. These traces were obtained by running a set of test cases. We use traces from passing test cases as they are likely to capture correct program behaviour. We make use of a number of execution traces made available by Krka et al. and generate additional execution traces by running programs in the DaCapo benchmark. The list of client programs that had been run to generate the traces are also listed in Table II.

Ground Truth Models: To evaluate the quality of a generated FSA, we need a ground truth FSA. Our ground truth FSAs are taken from those that were created by Krka et al. [15] and Pradel et al. [32]. Following Krka et al., we remove edges and nodes from the FSAs that do not appear in the execution traces that we use to mine the model. Also, since the models were not created by library creators, to ensure correct ground truths, we check the ground truth models against documented specifications of library usage. We corrected a few errors in the ground truth FSAs that were manually created by Krka et al. and Pradel et al. that do not follow the documented specifications. We exclude net.sf.jftp.net.wrappers.SftpConnection from our experiments (which was considered by Krka et al.) due to the lack of documentation, which prevented us from verifying its ground truth model. We exclude some library classes whose models are made available by Pradel et al. due to difficulties in running Daikon to collect execution traces from some of the JDK libraries[2]. The corrected ground-truth models are publicly available: https://github.com/ModelInference/SpecForge

Evaluation Metrics: To measure the effectiveness of SpecForge, we use precision and recall introduced by Lo and Khoo [8]. These have been previously used to evaluate many different specification mining algorithms, e.g., [15], [8]. Precision and recall are computed by comparing the language that is accepted by an inferred FSA with the language that is accepted by a ground truth FSA. Precision refers to the proportion of sentences that are accepted by the inferred model that is also accepted by the ground truth model. Recall refers to the proportion of sentences that are accepted by the ground truth model that is also accepted by the inferred model. We also compute F-measure, which is the harmonic mean of precision and recall and is defined as:

$$\textit{F-measure} = 2 \times \frac{\textit{Precision} \times \textit{Recall}}{\textit{Precision} + \textit{Recall}}$$

There is often a tradeoff between precision and recall: one can gain precision by sacrificing recall (and vice versa). For example, SpecForge we omit more constraints in the model fusion phase (resulting in a more general model that accepts more traces) to achieve higher recall. However, this *may* reduce precision (if correct constraints are removed in the process)[3]. In the extreme, if only one *correct* rule is selected, the precision will be 100%, but recall will be low. F-measure is often used as a summary measure that evaluates if an increase in precision outweighs a reduction in recall (and vice versa). Thus, we use F-measure as a final yardstick to evaluate the effectiveness of specification mining algorithms.

In the evaluation of precision and recall, we need to generate a set of sentences that characterize the language that is accepted by a FSA. To generate sufficient number of sentences that characterize the language that is accepted by an FSA, we follow the procedure described by Lo et al. in their recent empirical study paper that compares the effectiveness of various existing specification mining algorithms [24]. We generate a set of sentences that are accepted by the FSA such that each edge of the FSA is covered at least 10 times (10-transition-coverage). Since some of the inferred automata is an NFA (Non-deterministic Finite Automata) and are very large, to keep the number and length of traces manageable, we limit the number of traces to 10,000 and the length of traces to 100.

Default Configuration: Our proposed approach has a number of parameters that can be tuned. The first parameter is the selection of constraint templates that are used in the constraint enumeration step. The second parameter is the selection of heuristic used in the constraint selection step. By default, we make use of *all* the six constraint templates for the constraint enumeration step, and the Intersection heuristic for the constraint selection step.

B. Research Questions

RQ1: How effective is the SpecForge specification mining approach?

The effectiveness of a specification mining approach affects the usefulness of the mined specification for program comprehension and for automated program analysis. To answer this research questions, we measure the precision, recall, and F-measure of SpecForge in inferring behavioral models of the 13 library classes described in Section V-A.

RQ2: How much does SpecForge improve over existing specification mining approaches?

Many specification mining approaches that analyze execution traces and output a finite state automaton have been proposed in the literature. SpecForge is built on top of 7 existing approaches. To answer this research question, we compare the precision, recall, and F-measure of SpecForge with those of the 7 existing approaches for each of the 13 library classes that we investigate in this work.

RQ3: What is the impact of changing the constraint templates used in the constraint enumeration step?

In this paper, we consider six different constraint templates. Some constraints may capture important properties in a FSA that a miner successfully "generalizes" from a set of execution traces, while others may capture the idiosyncrasies of a FSA miner which "overfit" the execution traces. In this research question, we investigate if some constraint templates are prone to capturing incorrect constraints. To answer this question, we evaluate the effectiveness of SpecForge when only some of the constraint templates are considered. We then highlight the

[2]We have checked with Daikon developers who responded: "there is not an easy way to generate invariants within the JDK; we assume that the libraries are correct" [33].

[3]If only bad constraints are removed, then precision will also improve.

TABLE III: Precision, Recall, and F-measure: Our Approach with Default Configuration.

Target Library Classes	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%

effect of adding and omitting some constraint templates.

RQ4: What is the impact of changing the heuristics used in the constraint selection step?

In this work, we propose a number of heuristics that we can adopt when selecting constraints that have been extracted from the input FSAs. These heuristics, presented in Section IV-C, include: union, intersection, majority, and satisfied by $\geq$ N. For this research question, we evaluate the effectiveness of our approach when using each of these heuristics. To simplify analysis, we do not vary the constraint templates and use the templates from the default configuration.

C. Results

In the following subsections, we detail the experimental results that answer each of the four research questions.

1) RQ1: Effectiveness of SpecForge: We execute SpecForge on an Intel(R) E5-2667 2.9 GHz processor server with 189 GB RAM running Linux 2.6; on average, SpecForge takes less than one second to infer a specification for each input class. Table III shows the precision, recall, and F-measure of SpecForge for the different target library classes. We note that precision ranges from 54.62% to 100%, and that recall ranges from 24.32% to 100%, while F-measure ranges from 39.13% to 100%. Noticeably, the precision is higher than the recall in most of the cases (11 out of 13 classes). Thus, most of the behaviors captured in the inferred models are correct but some correct behaviors are not captured successfully. Furthermore, there are 8 library classes for which our approach achieves a precision of 100%, and 1 library class for which its recall is 100%. Our approach achieves the best F-measure for `LinkedList` (i.e., 90.59%) and it achieves the worst F-measure for `Signature` (i.e., 39.13%). Overall, SpecForge achieves an average precision, recall, and F-measure of 90.57%, 54.58%, and 64.21%, respectively.

We have manually investigated the inaccuracies of models generated with SpecForge. We found that the main cause of the low F-measures is due to wrong temporal rules being selected and fused into the overall model. Sections V-C3 and V-D describe how we can further improve the F-measure.

2) RQ2: Our Approach vs. Baselines: In this research, we compare the effectiveness of SpecForge against the 7 existing baselines described in Section II. These baselines are: traditional 1-tails, traditional 2-tails, CONTRACTOR++, SEKT 1-tails, SEKT 2-tails, optimistic TEMI, and pessimistic TEMI. Tables IV and V show the precision, recall, and F-measure of the baselines.[4] Table IV shows the precision, recall, and F-measure of SEKT 1-tails, SEKT 2-tails, Optimistic TEMI, and Pessimistic TEMI, while Table V shows the precision, recall, and F-measure of traditional 1-tails, traditional 2-tails, and CONTRACTOR++. From the tables, we find that CONTRACTOR++ has the best average F-measure of 56.45%, and SEKT 2-tails has the least average F-measure of 23.18%.

TABLE VI: Average Precision, Recall, and F-measure: Our Approach with Different Constraint Templates.

Constraint Templates	Average		
	Precision	Recall	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%

Comparing Table III with Tables IV and V, our approach outperforms the average F-measures of all the baselines by 13.75% to 177.02%. The average precision of our approach is slightly lower than those of the baselines (by up to 8.11%), however its recall is substantially higher than those of the baselines (by up to 296.37%). Overall, the above statistics show that our approach is more effective than all of the baselines.

3) RQ3: Different Constraint Templates: Table VI compares the effectiveness of SpecForge when different constraint templates are used in the constraint enumeration step. Due to space constraints, we do not show all possible combinations. From the table, we note that there are several combinations of constraint templates that result in higher average precision, recall and F-measure compared to the default setting. Among the combinations shown in Table VI, AF + NF + AP and AF+NF+AP+AIF have the highest average precision, recall, and F-measure respectively, which are 83.35%, 71.82% and 72.82%.

On the other hand, ALL − AP and AF + NF + AP have the least average F-measure of approximately 22%. The decrease in F-measure shows that the absence of the *always preceded by* constraints has a significant impact on the effectiveness of SpecForge. Overall, choosing a suitable combinations of constraint templates is important for improving effectiveness.

4) RQ4: Different Constraint Selection Heuristics: Table VII compares SpecForge's performance for different constraint selection heuristics. The table lists the selection heuristics in increasing order of strictness in selecting constraints.

[4]Krka et al. have also estimated the precision and recall of these approaches [15]. We use a more rigorous evaluation setting to generate sentences from inferred and ground truth models, i.e., 10-transition coverage (see Section V-A). Therefore, the results shown in Tables V and IV are different from the ones calculated by Krka et al. [15].

TABLE IV: Precision, Recall, and F-measure: Traditional 1-tail, Traditional 2-tail, and CONTRACTOR++. "P" = Precision, "R" = Recall, and "F" = F-measure.

Target Library Class	Traditional 1-tails			Traditional 2-tails			CONTRACTOR++		
	P	R	F	P	R	F	P	R	F
ArrayList	100.00%	11.15%	20.06%	100.00%	10.45%	18.92%	100.00%	46.15%	63.15%
HashMap	100.00%	22.68%	36.97%	100.00%	19.08%	32.05%	100.00%	4.32%	8.28%
HashSet	100.00%	20.76%	34.38%	100.00%	13.50%	23.79%	100.00%	100.00%	100.00%
Hashtable	100.00%	30.23%	46.43%	100.00%	21.89%	35.92%	100.00%	1.55%	3.05%
LinkedList	100.00%	29.49%	45.55%	100.00%	26.49%	41.88%	100.00%	79.39%	88.51%
StringTokenizer	68.18%	39.46%	49.99%	71.21%	21.77%	33.34%	71.38%	16.33%	26.57%
NFST	100.00%	1.80%	3.54%	100.00%	0.90%	1.79%	87.27%	40.54%	55.36%
SMTPProtocol	100.00%	17.50%	29.79%	100.00%	17.50%	29.79%	85.71%	50.00%	63.16%
Signature	100.00%	8.11%	15.00%	100.00%	8.11%	15.00%	100.00%	75.68%	86.15%
Socket	97.15%	10.18%	18.43%	98.69%	8.86%	16.26%	100.00%	0.22%	0.44%
StackAr	34.04%	14.52%	20.36%	51.69%	14.52%	22.67%	98.35%	100.00%	99.17%
ToHTMLStream	100.00%	20.00%	33.33%	100.00%	20.00%	33.33%	100.00%	100.00%	100.00%
ZipOutputStream	100.00%	0.00%	0.00%	95.00%	0.00%	0.00%	100.00%	25.00%	40.00%
Average	92.26%	17.38%	27.22%	93.58%	14.08%	23.44%	95.59%	49.17%	56.45%

TABLE V: Precision, Recall, and F-measure: SEKT 1-tail, SEKT 2-tail, Optimistic TEMI, and Pessimistic TEMI. "P" = Precision, "R" = Recall, and "F" = F-measure.

Target Library Class	SEKT 1-tails			SEKT 2-tails			Optimistic TEMI			Pessimistic TEMI		
	P	R	F	P	R	F	P	R	F	P	R	F
ArrayList	100.00%	10.50%	19.00%	100.00%	10.28%	18.64%	100.00%	31.03%	47.36%	100.00%	18.92%	31.82%
HashMap	100.00%	21.75%	35.73%	100.00%	19.02%	31.96%	100.00%	4.32%	8.28%	100.00%	1.71%	3.36%
HashSet	100.00%	20.76%	34.38%	100.00%	13.50%	23.79%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
Hashtable	100.00%	27.44%	43.06%	100.00%	20.79%	34.42%	100.00%	0.16%	0.32%	100.00%	1.58%	3.11%
LinkedList	100.00%	28.45%	44.30%	100.00%	25.96%	41.22%	100.00%	79.39%	88.51%	100.00%	34.51%	51.31%
StringTokenizer	63.64%	21.77%	32.44%	75.94%	20.41%	32.17%	52.89%	14.29%	22.50%	78.99%	17.01%	27.99%
NFST	100.00%	0.00%	0.00%	100.00%	0.00%	0.00%	89.61%	40.54%	55.83%	94.00%	30.63%	46.21%
SMTPProtocol	100.00%	17.50%	29.79%	100.00%	17.50%	29.79%	94.81%	50.00%	65.47%	100.00%	5.00%	9.52%
Signature	100.00%	8.11%	15.00%	100.00%	8.11%	15.00%	100.00%	75.68%	86.15%	100.00%	75.68%	86.15%
Socket	100.00%	10.11%	18.36%	100.00%	8.86%	16.28%	100.00%	0.22%	0.44%	100.00%	18.00%	30.51%
StackAr	95.55%	14.52%	25.21%	84.85%	14.52%	24.80%	98.55%	100.00%	99.27%	100.00%	1.79%	3.53%
ToHTMLStream	100.00%	20.00%	33.33%	100.00%	20.00%	33.33%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
ZipOutputStream	100.00%	0.00%	0.00%	100.00%	0.00%	0.00%	100.00%	25.00%	40.00%	100.00%	6.82%	12.77%
Average	96.86%	15.45%	25.43%	96.98%	13.77%	23.18%	95.07%	47.74%	54.93%	97.92%	31.67%	38.94%

TABLE VII: Average Precision, Recall, and F-measure: Our Approach with Different Constraint Selection Heuristics.

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

From the table, we notice that *intersection* is the most effective selection heuristic with an F-measure of 64.21%, while union is the least effective heuristic with an F-measure of 15.40%. The results also show that stricter selection heuristics tend to improve SpecForge's F-measure. Note that with a stricter heuristic, SpecForge only selects a subset of constraints which enlarges the language accepted by the final inferred FSA. This potentially increases recall (if more correct sentences are accepted by the inferred FSA) – in the worst case recall will remain the same (if no additional correct sentences are accepted). Selecting fewer constraints may reduce precision (if many incorrect sentences are accepted by the inferred FSA). However, in the results, we note that precision, in general, increases with stricter heuristics – this shows that the additional sentences in the accepted languages of inferred FSAs include no or few incorrect sentences.

D. Discussion

Untapped Potentials. In this paper we evaluated SpecForge based on the 7 specification mining techniques discussed in Section II. Currently, our meta-approach works better than existing baselines on average, however, it does not perform the best in all cases. In our future work, we plan to improve SpecForge's performance in two concrete ways:

1. We plan to extend SpecForge with other specification mining techniques, such as Synoptic [2] and Perfume [30]. This can help SpecForge exclude more incorrect constraints, which is the main reason why currently SpecForge performs poorly on some target classes. As just one example, if we drop 4 incorrect constraints that are used to construct the FSA for java.util.Hashtable we can boost the F-measure of the inferred model from 61.22% to 72.30%.

2. We also plan to investigate alternative selection heuristics. Currently SpecForge applies the same constraint selection heuristic across all the constraint templates. However, it is possible to apply a different selection heuristic to each template. This will allow us to vary the strictness of the heuristics used for the constraint selection process. One promising direction is to use machine learning approaches to identify the best heuristics to apply for each template by learning over a large training dataset.

Threats to Validity: There are several threats that potentially affect the validity of our study: threats to internal validity, threats to external validity, and threats to construct validity.

Threats to internal validity relates to experimental errors and biases. We have checked our code several times to find errors and have fixed those that we have found. However, there could be errors that we did not notice. We make use of ground truth models that were created by Krka et al. [15] and Pradel et al. [32]. We have checked their models against actual specifications published by the library authors (e.g., the Javadocs of the library classes) and fixed errors in the ground truth models that we have found. Still, it is possible that the API documentation against which we checked the ground truth models is incorrect.

Threats to external validity relates to the generalizability of our findings. In this work, we have analyzed 13 different target classes. This is larger than the number of target classes used to evaluate many prior studies, e.g., [15], [24], [23]. In the future, we plan to add more target classes to further reduce this threat to external validity.

Threats to construct validity relates to the suitability of our evaluation metrics. We have followed a popular approach to measure the effectiveness of a specification mining algorithm [20] that were followed in many prior works, e.g., [1], [2], [15], [8], [19], [23]. In the calculation of precision and recall, following a recent empirical study that compares the performance of various specification mining algorithm [24], we ensure that a sufficient number of sentences were generated to characterize the language that is accepted by a FSA. We do so by continuing to generate sentences from a FSA until 10-transition-coverage is satisfied or until the number of generated sentences reaches 10,000. In this work, we use F-measure (i.e., F1) as a summary yardstick. F1 gives equal weight to precision and to recall, considering them equally important. There are other alternatives, such as F2 (gives more weight to recall) and F0.5 (gives more weight to precision). When a mined specification is used as a formal specification in bug finding or testing tools, both low recall and low precision lead to uncaught errors and false alarms (depending on how the specifications are used). As we do not have a specific application in this paper, we have no reason to weigh precision and recall differently. We therefore use F1 as our summary measure.

VI. Related Work

In this section, we describe related work on specification mining. We start by describing existing studies that mine finite state automaton from execution traces in Section VI-A. We then describe existing studies that mine specifications in other formalisms, such as temporal properties and sequence diagrams, in Section VI-B.

A. Mining Finite State Automaton

Specification Miners:

Krka et al. propose several specification mining algorithms that can infer a FSA from execution traces by leveraging value-based invariants that are inferred by Daikon [15]. They propose a number of algorithms namely CONTRACTOR++, state-enhanced k-tails (SEKT), and trace-enhanced MTS inference (TEMI). A brief descriptions of these algorithms are presented in Section II. SpecForge builds on these algorithms and the two variants of k-tails. Our experiments have compared the effectiveness of SpecForge and these algorithms and demonstrated that SpecForge outperforms all of them.

Our work has been partly inspired by work of Beschastnikh et al. who have proposed an approach to specify FSA inference algorithms declaratively [1]. A specification consists of a set of property types (variable-labeled FSAs) that resemble our constraint templates and a composition function. Property instances matching the property type are mined from the traces (resulting in event-labeled FSAs) and these are then composed using the composition function into one FSA. The composition function resembles our model fusion step. However, we have a fundamentally different goal — to synergize existing model inference algorithms, rather than to describe existing or new inference algorithms. As a result, in our work we mine property instances that match the prescribed templates from the inferred models, rather than from the input traces. Additionally, our templates and fusion step are less generic than their property types and composition function as their aim is to express a variety of FSA inference algorithms.

Lo et al. propose an approach named SMArTIC that infers a finite state automaton from a set of execution traces [8]. This approach is built on a variant of k-tails automaton learning method that infers a probabilistic FSA and employs trace filtering and clustering. Erroneous traces are removed from the input execution traces and rather than learning a model from all the traces, the traces are clustered into groups and a separate FSA is learned from each group. These FSAs are later combined together into one FSA by identifying equivalent transitions – the goal is to get a larger FSA that accepts all the sentences accepted by the smaller FSAs. This is similar to the model fusion step of SpecForge. However, the aim of our work is to combine models from multiple specification mining algorithms.

Walkinshaw and Bogdanov propose an approach that allows users to manually input temporal properties to guide a specification mining algorithm in the inference of a FSA from execution traces [34]. This work was extended by Lo et al. who proposed an approach to automatically mine temporal properties from execution traces, and use these mined properties to automatically guide or steer a specification mining algorithm in its inference process [23]. As one step of SpecForge, we also infer temporal properties as constraints. However, rather than inferring them from execution traces, we infer them from FSAs that are generated by the underlying FSA mining algorithms on top of which SpecForge is built on. We do not use a data mining process to infer these properties, but rather a model checking algorithm.

Lorenzoli et al. [25], and Mariani and Pastore [26] propose two approaches named gkTail and KLFA to mine extended FSAs that incorporate data flow information. gkTail is able to infer algebraic constraints which specify restrictions on the values of some variables/arguments in the transitions of the FSAs. KLFA includes universally quantified constraints in the transitions of the FSAs to specify the re-occurrence of data values. Walkinshaw et al. have recently proposed an approach to generate algebraic constraints for transitions in a FSA by leveraging a classification algorithm [35]. In this work, we

focus on the generation of simple FSAs without algebraic constraints and quantified constraints.

Key Differences: None of the above-mentioned approaches are able to combine multiple FSAs mined by different algorithms together. To our knowledge, we are the first to propose this research direction. Our work can also integrate the above-mentioned miners: they can be used to learn new FSAs which can then be used as input to SpecForge. The goal of our work is to synergize many existing miners to build a more effective miner.

Miner Assessments:

A number of papers have also proposed methodologies to evaluate the quality of a specification mining algorithm that generates FSAs. Lo and Khoo propose a framework named QUARK to measure the quality of an inferred FSA in terms of precision and recall [20]. Pradel et al. perform an empirical study to evaluate the performance of specification mining algorithms by manually constructing a number of ground truth models for many target library classes [32]. Lo et al. extend the work by Lo and Khoo, and Pradel et al. by evaluating many more specification mining algorithms producing FSAs and extended FSAs [24]. They also define the notion of *n-transition coverage* in the generation of sentences from inferred and ground-truth automata to estimate precision and recall. In this work, we also make use of the notion of precision and recall to measure the quality of a specification mining algorithm. We also generate sentences from inferred and ground-truth automata following n-transition coverage. We also make use of models that were generated manually by Pradel et al. [32].

B. Mining Other Formalisms

Yang et al. extract two event temporal properties of a form similar to our always-followed-by constraints from execution traces [36]. Lo et al. extend Yang et al.'s work to mine temporal properties of arbitrary lengths that specify that "whenever a series of events occur, eventually another series of events will occur" [21]. Lo et al. also extend Yang et al.'s work by mining quantified temporal properties where data flow relationships among events in a rule is specified [7]. Recent work by Lemieux et al.[18] presents a tool for general LTL specification mining. Recently, Le and Lo investigate the effectiveness of various interestingness measures proposed in the data mining community (in addition to the commonly used support and confidence measures) to mine correct temporal properties from execution traces [16]. Fahland et al. mine modal sequence diagrams in the form of Live Sequence Charts from execution traces of systems [12]. Ernst et al. propose a well-known tool named Daikon to mine algebraic invariants (e.g., constraints on the values of arguments and global variables) from execution traces [11].

Different from these studies, we mine specifications in the form of FSA. SpecForge can potentially be integrated with the above mentioned studies, especially studies that mine temporal properties. These temporal properties can be combined with constraints that we infer from the mined FSAs and can be used to mine a more accurate FSA.

Recently, a number of more advanced techniques have been proposed to mine more complex rules. Lo et al. mine rules enriched with quantification [7]. With quantification, users can mine rules that specify data flow constraints between two method invocations, e.g., the output of one method invocation is the x^{th} input of another method invocation. Lo et al. also mine rules following the semantics of Live Sequence Charts (LSCs) which are enriched with Daikon-style constraints to serve as guards [22]. For all of the above studies, support and confidence have been used as the interestingness measures.

VII. Conclusion and Future Work

Software specifications are tremendously useful, yet composing them is labor-intensive and many software developers forego this process. In response, the research community has proposed numerous specification miners to extract likely software specifications. Unfortunately, due to the complexity of the specification mining problem, many existing miners employ techniques that impose certain trade-offs. For example, some miners rely on temporal properties, others employ statistical techniques, while still others utilize data invariants. In this work we propose SpecForge, a framework to synergize across previously proposed FSA specification miners by utilizing model fission and model fusion. SpecForge uses FSA specifications inferred by other specification miners to build a single superior FSA.

We have evaluated SpecForge by inferring specifications of 13 target library classes from the execution traces of their client applications. We demonstrated that the FSAs constructed with SpecForge are superior to those inferred by any one specification mining approach. Our experiments show that SpecForge (with default configuration) can achieve an average precision, recall, and F-measure of 90.57%, 54.58%, and 64.21% respectively. Although the average precision of SpecForge is slightly lower than the baselines (by up to 8.11%), its average recall is significantly better (by up to 296.37%). In terms of average F-measure, the harmonic mean of precision and recall, SpecForge improves over the best performing baseline by 13.75%. We have also tried to adjust the configuration of SpecForge, and the best configuration can achieve an average precision, recall, and F-measure of 83.35%, 71.82%, and 72.82% respectively. We believe that SpecForge generalizes and can easily include, and build on, other FSA specification mining approaches.

In the future, we plan to improve the effectiveness of SpecForge further by increasing the number of underlying FSA miners synergized together, increasing the number of constraint templates, and developing an approach that can infer good configurations of constraint templates and selection heuristics based on training data. We also plan to reduce the threats to external validity further by experimenting with additional target library classes and execution traces. Moreover, we plan to extend SpecForge to mine parametric specification following the work by Lee et al. [17] and Lo et al. [6], [7].

Acknowledgements

We would like to thank Ivo Krka and Michael Pradel for sharing their manually constructed ground truth models for a number of library classes online. We would also like to thank Ivo Krka for publicly releasing the implementations of the 7 specification mining algorithms that we forge together in this work.

REFERENCES

[1] 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," *IEEE Trans. Software Eng.*, vol. 41, no. 4, pp. 408–428, 2015.

[2] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst, "Leveraging existing instrumentation to automatically infer invariant-constrained models," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 267–277.

[3] A. W. Biermann and J. A. Feldman, "On the synthesis of finite-state machines from samples of their behavior," *IEEE Transactions on Computers*, vol. 100, no. 6, pp. 592–597, 1972.

[4] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*. New York, NY, USA: ACM Press, Oct. 2006, pp. 169–190.

[5] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, MA, USA: MIT Press, 1999.

[6] David Lo, Ganesan Ramalingam, Venkatesh Prasad Ranganath, and Kapil Vaswani, "Mining quantified temporal rules: Formalism, algorithms, and evaluation," in *16th Working Conference on Reverse Engineering, WCRE 2009, 13-16 October 2009, Lille, France*, 2009, pp. 62–71.

[7] David Lo, G. Ramalingam, V. P. Ranganath, and K. Vaswani, "Mining quantified temporal rules: Formalism, algorithms, and evaluation," *Sci. Comput. Program.*, vol. 77, no. 6, pp. 743–759, 2012.

[8] David Lo and Siau-Cheng Khoo, "Smartic: towards building an accurate, robust and scalable specification miner," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2006, Portland, Oregon, USA, November 5-11, 2006*, 2006, pp. 265–275.

[9] G. de Caso, V. A. Braberman, D. Garbervetsky, and S. Uchitel, "Automated abstractions for contract validation," *IEEE Trans. Software Eng.*, vol. 38, no. 1, pp. 141–162, 2012.

[10] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *Proceedings of the 1999 International Conference on Software Engineering, ICSE' 99, Los Angeles, CA, USA, May 16-22, 1999.*, 1999, pp. 411–420.

[11] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Sci. Comput. Program.*, vol. 69, no. 1-3, pp. 35–45, 2007.

[12] D. Fahland, D. Lo, and S. Maoz, "Mining branching-time scenarios," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, 2013, pp. 443–453.

[13] G. J. Holzmann, "The Model Checker SPIN," *IEEE Transactions on Software Engineering (TSE)*, vol. 23, no. 5, pp. 279–295, May 1997. [Online]. Available: http://dx.doi.org/10.1109/32.588521

[14] J. C. Knight, C. L. DeJong, M. S. Gibble, and L. G. Nakano, "Why are formal methods not used more widely?" in *Fourth NASA Formal Methods Workshop*, 1997, pp. 1–12.

[15] I. Krka, Y. Brun, and N. Medvidovic, "Automatic mining of specifications from invocation traces and method invariants," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, 2014, pp. 178–189.

[16] T. B. Le and D. Lo, "Beyond support and confidence: Exploring interestingness measures for rule-based specification mining," in *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015*, 2015, pp. 331–340.

[17] C. Lee, F. Chen, and G. Rosu, "Mining parametric specifications," in *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, 2011, pp. 591–600.

[18] C. Lemieux, D. Park, and I. Beschastnikh, "General LTL Specification Mining," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015.

[19] D. Lo, H. Cheng, J. Han, S.-C. Khoo, and C. Sun, "Classification of software behaviors for failure detection: a discriminative pattern mining approach," in *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2009, pp. 557–566.

[20] D. Lo and S. Khoo, "QUARK: empirical assessment of automaton-based specification miners," in *13th Working Conference on Reverse Engineering (WCRE 2006), 23-27 October 2006, Benevento, Italy*, 2006, pp. 51–60.

[21] D. Lo, S. Khoo, and C. Liu, "Mining temporal rules for software maintenance," *Journal of Software Maintenance*, vol. 20, no. 4, pp. 227–247, 2008.

[22] D. Lo and S. Maoz, "Scenario-based and Value-based Specification Mining: Better Together," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '10. New York, NY, USA: ACM, 2010, pp. 387–396.

[23] D. Lo, L. Mariani, and M. Pezzè, "Automatic steering of behavioral model inference," in *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009*, 2009, pp. 345–354.

[24] D. Lo, L. Mariani, and M. Santoro, "Learning extended FSA from software: An empirical assessment," *Journal of Systems and Software*, vol. 85, no. 9, pp. 2063–2076, 2012.

[25] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic generation of software behavioral models," in *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, 2008, pp. 501–510.

[26] L. Mariani and F. Pastore, "Automated identification of failure causes in system logs," in *19th International Symposium on Software Reliability Engineering (ISSRE 2008), 11-14 November 2008, Seattle/Redmond, WA, USA*, 2008, pp. 117–126.

[27] L. Mariani, F. Pastore, and M. Pezzè, "Dynamic analysis for diagnosing integration faults," *IEEE Trans. Software Eng.*, vol. 37, no. 4, pp. 486–508, 2011.

[28] W. Miao and S. Liu, "A formal specification-based integration testing approach," in *SOFL*, 2012, pp. 26–43.

[29] A. Møller, "dk.brics.automaton — Finite-state automata and regular expressions for Java," http://www.brics.dk/automaton/, 2010.

[30] T. Ohmann, M. Herzberg, S. Fiss, A. Halbert, M. Palyart, I. Beschastnikh, and Y. Brun, "Behavioral resource-aware model inference," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014, pp. 19–30.

[31] A. Pnueli, "The temporal logic of programs," in *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, Providence, RI, USA, 1977, DOI: 10.1109/SFCS.1977.32.

[32] M. Pradel, P. Bichsel, and T. R. Gross, "A framework for the evaluation of specification miners based on finite state machines," in *26th IEEE International Conference on Software Maintenance (ICSM 2010), September 12-18, 2010, Timisoara, Romania*, 2010, pp. 1–10.

[33] M. Roberts and M. Ernst, Personal Emails.

[34] N. Walkinshaw and K. Bogdanov, "Inferring finite-state models with temporal constraints," in *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy*, 2008, pp. 248–257.

[35] N. Walkinshaw, R. Taylor, and J. Derrick, "Inferring extended finite state machine models from software executions," in *20th Working Conference on Reverse Engineering, WCRE 2013, Koblenz, Germany, October 14-17, 2013*, 2013, pp. 301–310.

[36] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das, "Perracotta: mining temporal API rules from imperfect traces," in *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, 2006, pp. 282–291.

[37] H. Zhong and Z. Su, "Detecting API documentation errors," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, 2013, pp. 803–816.