Mining Specifications from Documentation Using a Crowd

Peng Sun
Iowa State University
Ames, IA, USA
psun@iastate.edu

Chris Brown
North Carolina State University
Raleigh, NC, USA
dcbrow10@ncsu.edu

Ivan Beschastnikh
University of British Columbia
Vancouver, BC, Canada
bestchai@cs.ubc.ca

Kathryn T. Stolee
North Carolina State University
Raleigh, NC, USA
ktstolee@ncsu.edu

***Abstract*—Temporal API specifications are useful for many software engineering tasks, such as test case generation. In practice, however, APIs are rarely formally specified, inspiring researchers to develop tools that infer or mine specifications automatically.**

Traditional specification miners infer likely temporal properties by statically analyzing the source code or by analyzing program runtime traces. These approaches are frequently confounded by the complexity of modern software and by the unavailability of representative and correct traces. Formally specifying software is traditionally an expert task. We hypothesize that human *crowd intelligence* provides a scalable and high-quality alternative to experts, without compromising on quality.

In this work we present *CrowdSpec*, an approach to use collective intelligence of crowds to generate or improve automatically mined specifications. CrowdSpec uses the observation that APIs are often accompanied by natural language documentation, which is a more appropriate resource for humans to interpret and is a complementary source of information to what is used by most automated specification miners.

***Index Terms*—Specification mining, crowdsourcing, Java APIs**

I. Introduction

Software APIs are designed to be used (and abused) by countless client programs. It is critical to specify correct API behavior to capture features that are absent or difficult to infer from the underlying implementation. The resulting API specifications, or *specs* for short, can be used as inputs to a variety of tools, including model checkers [1], automatic test-case generators [2], and program repair tools [3].

Manually specifying an API's behavior is a difficult task, and unsurprisingly, most APIs do not have specs. One line of work that has approached the problem of missing specs proposes to automatically infer *likely* API specs from executions of API client programs (Robillard, et al. extensively survey this space [4]). This line of work has produced techniques that infer a variety of specifications, such as data specs [5], temporal specs [6], [7], [8], [9], some combination of the two [10], [11], and other spec varieties [12]. Many of these are dynamic analysis techniques and require (1) diverse traces of program behavior, and (2) make the assumption that the observed behavior is correct. However, all these miners are incomplete: they will miss true specs for methods that do not appear in the traces.

Although automated approaches may produce a rough initial spec, what they sorely need is human-level reflection and insight. Recent work by Legunsen et al. [13] evaluated 17 Java API specs mined by automated techniques [14] to evaluate their bug finding effectiveness. They found a false alarm rate of 96.69%. This indicates that before automatically-derived specs can be used by practitioners, these specs must be reviewed for accuracy by humans. As expert knowledge is expensive, a cheaper and more scalable way to obtain human insight is to use the crowd. Furthermore, the collective crowd wisdom can often outperform individuals, even on complex tasks [15].

In this paper we propose *CrowdSpec*, a methodology to link automated spec mining approaches with manual creation of high-quality specs. We demonstrate that human crowd intelligence in CrowdSpec improves existing automated spec miners; we illustrate this by applying CrowdSpec to a recent state-of-the-art automated approach called SpecForge [8]. We evaluate CrowdSpec using three real world and popular Java APIs: `HashSet`, `StringTokenizer`, and `StackAr`. Since the aim of prior techniques like SpecForge has been to forego humans altogether, CrowdSpec pushes back on this trend by bringing humans back into the spec creation loop.

CrowdSpec re-introduces humans into the spec mining process in a *lightweight* and *scalable* manner. CrowdSpec is lightweight because it uses the fact that APIs are often accompanied by natural language resources, such as documentation, which are easier for a non-specialist to read and understand. These resources are better-suited to crowd-based spec mining than resources like source code or execution traces. In CrowdSpec, human crowd workers check temporal API specs against API documentation.

CrowdSpec's second key feature is scalability. Rather than using a recruited pool of experts, CrowdSpec uses human intelligence in a crowd. We prototyped CrowdSpec using Amazon's Mechanical Turk (MTurk) [16], a popular crowdsourcing platform. Using a thorough automated screening process, CrowdSpec identifies workers who are technically competent enough to answer questions about Java APIs and linear temporal logic (LTL) property types. Furthermore, we find that this crowd is sufficiently large. The three APIs we studied contain 1,998 properties in total that require multiple crowd opinions each (for replication). In our studies, 198 participants commented on 13,944 properties in 2,324 tasks.

To summarize, our work evaluates whether or not a crowd can positively contribute to spec mining tasks. Our results show that the crowd can improve the accuracy of a state-of-the-art miner, SpecForge [8], with gains in recall of 13.9% to 26.8% on three Java APIs: `HashSet`, `StringTokenizer`, and `StackAr`. Furthermore, we find that the crowd can perform as well as experts on two of the three APIs. The contributions of this work are:

- Evidence that crowd intelligence, when applied to Java API documentation, improves the accuracy and recall of API specs generated by the spec miner SpecForge (RQ1).
- Evidence that screening through qualification tests can identify a large crowd (198 workers) that is technically competent enough to answer questions about LTL specifications (RQ1).
- Qualitative analysis of where and why the crowd made mistakes in extracting LTL properties from Java API documentation. We find that ambiguities in API documentation were not the dominant source of errors, suggesting that documentation can be a reliable source of information for composing LTL specifications for Java APIs (RQ2).
- Results indicating that a combination of SpecForge+CrowdSpec can perform as well as voting experts. This hints at the power of hybrid approaches that combine crowd and automated specification miners to derive software specifications (RQ3).

In the following section we provide essential background information for understanding the rest of the paper.

II. Background

The broad objective of this research is to determine whether human crowd intelligence can enhance, or even replace, automated specification miners.

Java APIs. We consider specs for three Java APIs: `HashSet`, `StringTokenizer`, and `StackAr`. We chose these libraries for two reasons.

1) They are used in prior work so we can *compare* CrowdSpec to prior mining approaches [8].
2) They have small enough APIs that we can manually derive a ground truth to use to evaluate crowd accuracy.

Both `HashSet` and `StringTokenizer` are widely used and are part of the standard Java library, which has high quality documentation in the form of publicly available JavaDoc pages. We use the Java 7 version of these libraries with the assumption that more participants would be familiar with this earlier version. The `StackAr` library is an external library that has several different implementations (we use [17]). The three APIs have between six and nine methods each and have multiple constructors; to limit the study costs without loss of generality, we only consider the primary constructor, e.g., `HashSet()`.

Temporal specification property types. We mine specs based on six common linear temporal logic (LTL) *property types* listed below. These have been studied by Dwyer, et al. [18] and have been used by prior spec miners such as SpecForge [8], InvariMint [19], Synoptic [20], and Perracotta [21]. For each of these six property types we consider *property instances*, or *properties* for short, formed by all possible combinations of the methods from each API[1].

We define a property to be true if (1) a program that uses an API and does not follow the property will trigger a Java exception, or (2) a violation of the property is impossible in the Java language (e.g., a constructor call must precede any other object method call, otherwise the code is not well-formed). Next, we explain each LTL property type in the context of the `HashSet` API:

- a is always followed (AF[2]) by b: an occurrence of event a must be eventually followed by an occurrence of event b (e.g., a false property example: *size() AF clone()*).
- a is never followed (NF[3]) by b: there are no occurrences of event b after an occurrence of event a (e.g., a true property example: *iterator() NF HashSet()*).
- a always precedes (AP[4]) b: an occurrence of event b must be preceded by event a (e.g., a true property example: *HashSet() AP size()*).
- a always immediately precedes (AIP[5]) b: an occurrence of event b must be immediately preceded by an occurrence of event a (e.g., a false property example: *size() AIP clear()*).
- a is always immediately followed (AIF[6]) by b: an occurrence of event a must be immediately followed by an occurrence of event b (e.g., a false property example: *contains(Object o) = false AIF clear()*).
- a is never immediately followed (NIF[7]) by b: there are no occurrences of event b immediately after any occurrence of event a (e.g., a true property example: *clear() NIF add(E e) = false*).

SpecForge. In this paper we use CrowdSpec to improve the SpecForge tool [8]. SpecForge is a specification mining approach that infers specifications using multiple state-of-the-art finite state automaton (FSA) miners: k-tails [23], CONTRACTOR++, SEKT, and TEMI [24], although SpecForge is not constrained to combining just these spec miners. SpecForge combines these algorithms in a way that allows it to outperform each of them individually, thus representing one of the most accurate automated LTL specification mining tools based on execution traces.

SpecForge uses a three-step algorithm:

1) Runs the FSA spec mining algorithms on the input traces to independently generate multiple FSA models, one model per spec miner.
2) Uses model checking to perform *model fissions* [8] to extract temporal constraints that are common across the mined FSAs.

[1]We follow the majority of prior work in scoping each property to a single object instance; we do not consider multi-object properties [22].
[2]LTL: $G(a \rightarrow XF\ b)$
[3]LTL: $G(a \rightarrow XG(\neg b))$.
[4]LTL: $\neg b$ W a
[5]LTL: $F(a) \rightarrow (\neg a\ U(b \wedge Xa))$.
[6]LTL: $G(a \rightarrow X\ b)$.
[7]LTL: $G(a \rightarrow X(\neg b))$.

3) Performs *model fusions* [8] to combine the common temporal constraints into a single FSA model that it then outputs.

To improve SpecForge with CrowdSpec, we introduce a crowdsourcing step after the fission step in SpecForge. In this new step (2.5) the crowd would further verify the properties derived during the fission step.

We selected SpecForge for two reasons. First, to our knowledge, it is one of the best performing FSA spec mining algorithms in the literature. Second, SpecForge uses simple and concise temporal properties – instantiations of exactly the six property types listed previously. These short property types are a good match to microtask crowdsourcing.

Crowdsourcing. In this work, we use microtask crowdsourcing to obtain human insights on API properties. Crowdsourcing obtains human intelligence on tasks by publishing those tasks to an unknown crowd of qualified workers, and it has been used extensively in software engineering [25]. *Microtask* crowdsourcing involves giving small tasks to the crowd, which can be accomplished in minutes rather than hours. In this work, we decompose the larger problem of obtaining insights on all properties in an API into smaller tasks. In our studies each *task* contains questions concerning *six property instances*, one per property type; all six instances relate the same *two* methods from an API under study.

Achieving high accuracy with MTurk participants requires quality-control mechanisms. Our CrowdSpec infrastructure uses best practices in crowd control to encourage high response quality (see Section IV-E). CrowdSpec uses gold standard questions [26], redundant question formats [27], notices about the value of their work [28], instructional tutors [29], conflict detection [30], and random click detection [31]. In concert these techniques train the crowd to deliver highly accurate results on the temporal spec mining task.

III. Research Questions

To determine whether crowd intelligence can improve (or replace) machine-generated specs, we designed and ran several studies. Our studies were designed to answer the following three research questions:

RQ 1. Can a crowd use documentation to improve machine-generated specs?

To answer RQ1 we ran experiments in which we used the crowd to *improve* specs mined by state-of-the-art temporal spec miner SpecForge [8]. Since SpecForge identifies true specs for pairs of methods it *observes* in the traces, we posit that there is an opportunity to improve SpecForge-mined specs, particularly the recall (on its own SpecForge has a recall of 46.43%[8] on the `HashSet` library).

We answer this question with two studies on `HashSet` (*HashSet_A* and a full replication, *HashSet_B*), one study on `StringTokenizer` (*StringToken*), and one study on `StackAr` (*StackAr*). In each study, the crowd was presented with tasks. In each task, two methods from the API were selected and the crowd was asked questions about six property instances considering those two methods, one for each of the six LTL property types. The answer from SpecForge (whether the property instance is true or false) was provided and the crowd was given the opportunity to agree or disagree; properties that SpecForge does not observe, it assumes to be false. We measured improvement in terms of overall accuracy, precision, and recall as compared to the original SpecForge answers and the ground truth.

SpecForge uses program traces while the crowd uses documentation, and both information sources are imperfect. When the crowd makes a mistake, it could be because of imperfect documentation, imperfect study infrastructure, an imperfect understanding of LTL properties, fatigue, or other reasons. Thus, we pose a second research question:

RQ 2. Why does the crowd make mistakes when identifying LTL specs based on JavaDoc documentation?

Using humans to specify library APIs can be expensive, and if these property instances need to be later re-checked by experts, perhaps it is best to use experts in the first place. However, when the errors are due to imperfect documentation, it might not matter whether we use the crowd or the experts. Our crowd studies asked each participant to provide a free-text rationale for their responses. We study these qualitative responses to determine when (and why) the crowd selects the wrong answer (see Section IV-D).

Provided the crowd can improve automated specification miner's inferred specifications, our third RQ is about comparing SpecForge+CrowdSpec against a group of experts:

RQ 3. Can SpecForge+CrowdSpec perform as well as a group of experts when formulating LTL specs for an API?

To form the ground truth, a group of three experts independently derived ground truth based on documentation and then discussed discrepancies to reach a consensus (see Section IV-A). In this research question, we remove the discussion component and compare SpecForge+CrowdSpec against each of the experts individually as well as against the three experts when voting on each property (instead of discussing). The goal is to highlight performance differences between using a crowd and using a single expert in isolation, or using three experts and combining their opinions by voting.

IV. Study

To run our studies, we need to collect a ground truth, design tasks that collect quantitative and qualitative feedback to answer the RQs, and define metrics for the analysis.

A. Obtaining ground truth

SpecForge computed precision/recall metrics using model checking over ground truth models. However, not all methods appear in these models: e.g., the model for `HashSet` is missing *clear()* and *clone()* methods, which are, for example, part of ten

[8]This is different from the 55.44% reported in SpecForge [8], which is due to what is considered to be the ground truth; see Section IV-A.

TABLE I
API GROUND TRUTH DETAILS. INSTANCES REFER TO PROPERTY INSTANCES, EXPERT AGREEMENT IS INTER-RATER FLEISS' KAPPA, AND % TRUE LISTS THE FRACTION OF TRUE INSTANCES IN THE API.

API	Instances	Agreement	% True
`HashSet`	1,014	0.82	6% (56)
`StringTokenizer`	384	0.76	9% (35)
`StackAr`	600	0.76	7% (43)

TABLE II
DISTRIBUTION OF TRUE PROPERTIES FOR EACH PROPERTY TYPE, PER API.

Property	`HashSet`	`StringTokenizer`	`StackAr`
AF	0%(0)	0%(0)	0%(0)
NF	8%(13)	13%(8)	10%(10)
AP	8%(14)	11%(7)	11%(11)
AIP	0%(0)	0%(0)	0%(0)
AIF	0%(0)	0%(0)	0%(0)
NIF	17%(29)	31%(20)	22%(22)

true properties in our manually-derived ground truth. Further, we sought a more complete picture of each API spec, one that includes explicitly stated false properties. Thus, we created a ground truth specification dataset for each of the three APIs using the following process:

Three paper authors (hereafter referred to as *experts*[9]) manually labeled property instances for six property types (AF, AIF, NF, NIF, AP, AIP) across all possible pairs of methods in each API. We treated methods with a boolean return type as two entries, one for true and one for false return value. When there was not unanimous agreement, the authors discussed each property and came to a consensus. Typical disagreements were either oversights or miscommunication on the requirements of a true property, as defined in Section II. Table I lists the total number of property instances per API, along with an inter-expert agreement score (Fleiss' kappa).

We observed that some property types, specifically AF, AIF, and AIP, cannot be true for any pair of methods. This is because the client of the API can exit at any time. These results are presented in Table II. For liveness property types AF and AIF and all pairs of methods (a, b), we can always end the program right after calling a. Likewise, for *a AIP b*, it is always possible to call a method between a and b for all pairs of methods. Table I lists the fraction of true property instances per property type in each API.

We make the ground truth and all of our experimental data available for other researchers to review and to use [32].

B. Tasks

Each task in our studies was designed such that a participant explores the six property types between two methods in a single API. Each task contains the JavaDoc information for the two methods and the following materials for each of the six property types: 1) the SpecForge answer (referred to in the tasks as the "machine's answer"), 2) a question about whether the participant agrees or disagrees with the machine, 3) free-text space to provide an explanation, and 4) a 5-point Likert scale question about confidence. Figure 1 shows the question portion of the task for the `clear()` and `clone()` methods in the `HashSet API`, and the AF property type. Method descriptions from the API are provided, and an API documentation link leads to the library Java 7 JavaDoc (and to [17] for `StackAr`).

[9]Two of whom hold PhDs in CS and research program analysis.

TABLE III
MEASURES USED IN OUR EVALUATION.

		Ground Truth	
		True	False
Crowd Decision	True	True Positive (tp)	False Positive (fp)
	False	False Negative (fn)	True Negative (tn)

C. Metrics

We measure the crowd's accuracy against the ground truth to answer RQ1, and the crowd's and experts' accuracy to answer RQ3. Most spec miners that use dynamic analysis, such as SpecForge, only explicitly identify true properties [8], [21]. For the false properties, these techniques do not typically distinguish between a property that is not mined because a trace violates it, and a property that was not observed or for which the traces did not provide sufficient evidence. In the latter case, this leads to incompleteness. Our experiments, on the other hand, can identify properties explicitly as true or as false, allowing an *exhaustive* evaluation of API property instances. For this reason, we measure accuracy, which represents correctness compared to the ground truth, in addition to precision and recall. Table III summarizes our metrics notation.

In each experiment we assign multiple participants to each property. To extract a crowd consensus, we assign an odd number of participants to each property and use majority rule to determine the crowd's opinion.

Precision. (p) is the percentage of properties that are actually true, of those that are reported to be true: $p = \frac{tp}{tp+fp}$. For our experiments, precision is the percentage of the correctly labeled true properties from the crowd.

Recall. (r) represents the percentage of the true properties that are reported to be true: $r = \frac{tp}{tp+fn}$. For our experiments, recall is the percentage of true properties in an API that were identified as true by the crowd.

Accuracy. (a) is the percent of correctly mined properties, true and false, in the ground truth: $a = \frac{tp+tn}{tp+fp+fn+tn}$. Unlike precision and recall, accuracy includes tn properties since the crowd explicitly defines properties as true or false.

D. Qualitative Analysis

We explore why the crowd makes mistakes when identifying LTL specs with the three Java APIs for RQ2. To accomplish this, we identify all responses where participants' answers disagreed with the ground truth. Then, two authors independently coded user responses into categories that describe why they made mistakes using an open card sort. In formulating these categories the coders focused on capturing *why* each participant made a mistake, using the participant's response and the free-text explanations of their response for guidance.

The two coders first independently analyzed the replies for the `HashSet` API. Then, they discussed the initial groupings

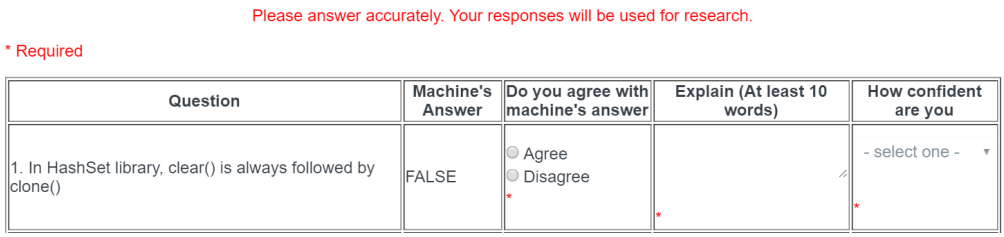

Fig. 1. HIT Design for CrowdSpec studies, with one property type (AF) shown for methods `clear()` and `clone()`. We set the *Machine's Answer* to TRUE or FALSE according to the SpecForge answers.

TABLE IV
STUDY AND PARTICIPANT CHARACTERISTICS.

Study Features	*HashSet_A*	*HashSet_B*	*StringToken*	*StackAr*
People per task	5	5	3/4/5	3/4/5
Payment	$0.40	$0.40	$0.40	$0.40
Total cost	$473.75	$473.73	$138.68	$218.05
Valid responses	845	845	246	388
Duration	2 days	4 days	30 days	17 days
Quality Control	*HashSet_A*	*HashSet_B*	*StringToken*	*StackAr*
Qualification test	yes	yes	yes	yes
# questions	7	7	7	7
Conflict detection	yes	yes	yes	yes
Gold standard	yes	yes	yes	yes
Random click	yes	yes	yes	yes
Participants	*HashSet_A*	*HashSet_B*	*StringToken*	*StackAr*
Total participants	39	38	66	55
Male/female/unk	30/9/0	28/8/2	51/15/0	32/23/0
Avg. age	30	31	33	34
% CS degree	74%	74%	68%	60%
Java familiarity	3.87	3.95	3.64	3.51

to distill 12 error categories. Next, they used these 12 categories to classify the responses in the `StringTokenizer` and `StackAr` API studies. The coders went through and compared their codings based on the error category each response was assigned. When there was disagreement, the coders discussed and came to an agreement. Finally, high-level error classes were identified to classify the participants' mistakes.

E. Implementation

To gain access to a crowd of participants, we used MTurk's microtask crowdsourcing platform. We created Human Intelligence Tasks (HITs) that are performed by MTurk workers, where each task described in Section IV-B was a HIT. We built CrowdSpec on our own server to afford us more control over the study context. This server interfaces with Amazon's MTurk, which was used to manage recruitment, advertisement, and payment. Our system is based on PHP and MySQL.

We ran four studies, *HashSet_A*, *HashSet_B*, *StringToken* and *StackAr* across three different API libraries. Each study had an independent sample of participants, including *HashSet_B*, which is a replication of *HashSet_A*. Table IV summarizes several features of each study: MTurk logistics (e.g., cost, workers per task, study duration), quality control mechanisms (e.g., conflict detection, random click detection), and participant characteristics (e.g., gender, age, experience). Next, we elaborate on each category.

1) **MTurk Study Logistics:** For consistency in exposure to potential participant populations, each study was deployed at 10pm Eastern Standard Time on a Sunday evening. An upper bound of five workers were assigned to each HIT. In the *StringToken* and *StackAr* studies, we removed HITs if consensus was reached on all six property types by three or four people as a cost-saving technique. Participants were paid $0.40 per HIT[10] and the studies ran between two and 30 days. Submitted HITs can be approved or rejected, and workers, or participants, are only paid for approved HITs. Approval was granted if all questions were answered, gold standard questions were answered correctly, and no conflicts were detected (explained next). Participants had 20 minutes to complete each HIT.

2) **Quality Control:** We used best practices from the crowdsourcing literature to ensure high quality responses. Two general active quality control strategies were employed. First, we used a qualification test to screen the participants. Second, we used within-study checks on their work, including conflict detection and gold standard questions. All of our quality control techniques were entirely automated and did not increase the cost of our studies.

a) Qualification Pretest: The workers are qualified to perform and submit our HITs if they pass a qualification test. For workers unfamiliar with LTL properties, we incorporated training materials into the qualification test [29]. We showed examples and explanations of each of the six LTL property types using the Java `HashMap` API. Then, we used the `ArrayList` library in the qualification test, where the questions were identical to those in the HITs. Participants passed if they answered at least 5 out of 7 questions correctly.

b) Within-study Controls: Despite being qualified to perform the tasks, we ensured high quality results by employing within-study quality checks, specifically conflict detection, lightweight random click detection, gold standard questions, redundant question formats, and indications that their responses are important for research. In concert, these controls ensure the crowd delivers highly accurate results on the temporal specification mining tasks.

To ensure comprehension on each HIT submission we used *in vivo* conflict checking [30] to determine when participants submitted conflicting responses for related properties, such as (AF and NF) or (AIF and NIF). A conflict happens when a participant responds true on both properties in either pair

[10]Each HIT had 6 property instances; per-property instance cost was $0.07.

mentioned above. If a conflict is detected, the HIT is rejected. For example, indicating that `clone()` AF `clear()` is true *and* that `clone()` NF `clear()` is true, is a conflict. Workers were alerted of conflicts at the time of submission and given the opportunity to modify their response. If a conflict was still submitted, the HIT was rejected.

To combat workers who were gaming the system, we included lightweight random click detection [31]. If a worker spent less than one minute on a HIT, a warning was given upon clicking the submission button: *"It seems you are randomly clicking through, this may cause your submission to be rejected."* Participants were able to revise their answer after getting the warning. If a participant spent more time reviewing their answer (beyond the one minute threshold), then the automatic time check detection was canceled. However, if the participant still submitted the HIT within the one minute threshold, they were marked as a random clicker and gold standard questions appeared on their next HIT. Workers marked as random clickers were not blocked.

When a random clicker is detected, that participant's next HIT is augmented with two gold standard questions [26], in addition to the six LTL questions. We directly choose the gold questions from the set of questions the participant answered correctly in the qualification test under the assumption that participants understand the meaning of questions they have correctly submitted in qualification test. Workers are required to provide correct answers to both gold standard questions. If either gold standard question is incorrect, this is logged on the workers' profile on our server and the HIT is rejected. If a worker answers at least one gold standard question incorrectly in two different HITs, the worker is blocked.

To combat listlessness, we used redundant question formats [27] to make sure workers continued to pay attention after the qualification test. This also facilitates the qualitative analysis to support RQ2 (Section V-B). To combat worker apathy, we indicated that their work is important [28], as illustrated by the red text in Figure 1.

In our experience, these quality control mechanisms are essential. We initially ran the same study with only the qualification test for quality control, omitting the within-study checks. Accuracy on the `HashSet`, `StringTokenizer`, and `StackAr` APIs was 52%, 42%, and 50%, respectively, with precision scores under 14% for all three APIs. These poor results drove us to investigate best practices in eliciting quality results from the crowd. As shown in Section V, accuracy rose to above 93% for all the APIs; precision increased to 67% at worst and 100% at best. For the rest of the paper, we refer only to the studies that had all the quality control mechanisms.

3) **Participants:** To control for between-study learning effects, each study had an independent group of participants.[11] In total, 198 participants performed the tasks, with 60% - 74% of them having a CS degree. Using a 5-point Likert scale with 1 being "not familiar at all" and 5 being "very familiar", most participants were at least somewhat familiar with Java at the start of the study.

[11]The initial study, not reported here, also had an independent group of 26.

TABLE V
COMPARISON OF THE CROWD AND SPECFORGE AGAINST GROUND TRUTH; 56 OF 1,014 PROPERTIES ARE TRUE IN `HashSet` LIBRARY, 35 OF 384 PROPERTIES ARE TRUE IN `StringTokenizer` LIBRARY, 43 OF 600 THE PROPERTIES ARE TRUE IN `StackAr`

Study	**Accuracy**	fp	fn	p	r
HashSet_A	98.03%	0.00%	1.97%	100.00%	64.29%
HashSet_B	98.03%	0.49%	1.48%	89.13%	73.21%
SpecForge_HS	97.04%	0.00%	2.96%	100.00%	46.43%
StringToken	93.49%	2.34%	4.17%	67.86%	54.29%
SpecForge_ST	91.15%	3.39%	5.47%	51.85%	40.00%
StackAr	98.50%	1.00%	0.50%	86.96%	93.02%
SpecForge_SA	98.50%	0.00%	1.50%	100.00%	79.07%

V. RESULTS

A. RQ1: Can a crowd improve SpecForge?

The high-level results appear in Table V. The columns represent the metrics from Section IV-C. The rows are split by API: the first three rows are for `HashSet`; the next two for `StringTokenizer`; the last two for `StackAr`. If the study name has the prefix `SpecForge`, this means it is the results from SpecForge compared to the ground truth. The remaining rows are the results from the four SpecForge+CrowdSpec studies.

Regarding `HashSet`, *HashSet_A* and *HashSet_B* both show an improvement in accuracy over SpecForge by nearly 1%. We note that a 1% improvement is substantial because 1) from a statistical point of view, the difference between 97% and 98% in our study shows a P-value of 0.002 (McNemar's test), which is significant; 2) the error rate (2%) of our studies reduces the error rate (3%) of SpecForge by one-third.

Regarding `StringTokenizer`, *StringToken* shows an improvement in accuracy over SpecForge by 2%. Regarding `StackAr`, *StackAr* shows no improvement in accuracy over SpecForge, but we note that SpecForge performs the best on this API out of the three. For recall, the gains are 14% (*StringToken*) to 27% (*HashSet_B*). In concert, these results demonstrate that the crowd is indeed capable of improving machine-mined specs using documentation.

We also note that all the properties on which the crowd incorrectly disagreed with SpecForge were of NF or NIF property types. Table VI presents results comparing the crowd's accuracy on different property types, separated by API (the *HashSet_A* and *HashSet_B* studies are combined for `HashSet`). The crowd was the most accurate on the AF, AIF, and AIP property types; the least accurate property type is NIF. The crowd's consistent accuracy for these property types, whether it is high or low, indicates that these types are particularly easier/harder to understand for participants. The poor performance on NIF and NF property types may also indicate that the documentation is not sufficiently detailed to evaluate them.

One factor that could impact accuracy is Java familiarity. To determine this we partitioned the participants into three groups by their stated familiarity with Java on the qualification

TABLE VI
CROWD'S ACCURACY ON EACH PROPERTY TYPE AND EACH API.

	HashSet			StringTokenizer			StackAr		
	Accuracy	Precision	Recall	Accuracy	Precision	Recall	Accuracy	Precision	Recall
AF	100.00%	0.00%	0.00%	98.44%	0.00%	0.00%	100.00%	0.00%	0.00%
NF	97.63%	95.46%	73.08%	85.94%	44.44%	50.00%	98.00%	90.00%	90.00%
AP	98.82%	100.00%	85.71%	93.75%	80.00%	57.14%	98.00%	100.00%	81.82%
AIP	100.00%	0.00%	0.00%	100.00%	0.00%	0.00%	100.00%	0.00%	0.00%
AIF	100.00%	0.00%	0.00%	100.00%	0.00%	0.00%	100.00%	0.00%	0.00%
NIF	91.72%	91.30%	58.62%	82.81%	84.62%	55.00%	95.00%	81.48%	100.00%

test. Participants were categorized as *Familiar* if they stated they were *familiar* or *very familiar* in the survey following the qualification test. The *Unfamiliar* category represents *unfamiliar* and *very unfamiliar* responses. Overall, participants' accuracy across the *familiar* and *unfamiliar* groups did not differ significantly across all the libraries (α = 0.05, Mann-Whitney U test). One potential implication of this result is that workers who were unfamiliar with Java were sufficiently trained at the start of the study, such that they performed as well as workers who were familiar with Java.

Summary: We answer RQ1 in the affirmative: the crowd is able to improve machine-mined LTL-specifications, even when that crowd is only somewhat familiar with Java at the start. It also shows the value of combining trace-based analysis (SpecForge) with documentation-based analysis (CrowdSpec). Future work in this space should consider other types of specifications, such as data specs [5].

B. RQ2: Why does the crowd make mistakes?

To gain a deeper understanding of why the crowd answers incorrectly on LTL specification questions, we analyzed the crowd's explanations of their incorrect answers. We analyzed a total of 582 mistakes made by participants in the study, 183 on `HashSet`, 180 on `StringTokenizer`, and 219 on `StackAr`. As `HashSet` is a much larger API than the others, we randomly sampled approximately 20% of the errors to analyze a data set similar in size to the others.

As discussed in Section IV-D, two coders used the `HashSet` dataset to distill common error categories. We calculated the inter-rater agreement for `StringTokenizer` (κ = 0.26) and `StackAr` (κ = 0.49) using Cohen's Kappa. Using Landis' measurement of observer agreement for categorical data, our agreement for `StackAr` had a moderate strength of agreement and `StringTokenizer` had a fair strength of agreement [33]. Table VII details the 12 error categories, grouped into four high-level error classes. Table VIII lists the distribution of wrong responses across the error categories for each API and in aggregate across the APIs.

a) **API documentation errors:** Approximately 22% of the incorrect responses had a textual explanation that indicated a misunderstanding of the libraries (*API Error* row in Table VIII); 43% of the participants had at least one wrong property response in this category. This category includes misunderstanding constructor usage (APIb) and method relationships (APIa), confusion about parameters (APIe) and method return values (APId), or overlooking other methods in the API (APIc).

The documentation for `HashSet` had the highest error rate due to API issues (29%), while the `StringTokenizer` documentation was the least confusing (18%). Method relation errors were the most prevalent mistakes made in this category, implying that the crowd was confused about the interactions between methods. For example, in Table VII, the example for APIa demonstrates that a worker indicated `push(Object o)` and `pop()` are unrelated, when in fact both impact the program state. The frequency of this category highlights the importance of improving API documentation quality, with special attention paid to method relations.

b) **True spec errors:** Approximately 22% of the error responses indicated workers struggled with understanding the definition of LTL specs; 36% of participants wrote responses in this category. This includes misunderstanding LTL definitions (TSa), overlooking the requirement for a single instance (TSc), and making judgments based on bad practice (TSb).

Participants tend to have trouble with understanding the definition of a true spec and LTL properties (15%). For example, some participants reversed the method order when answering a question, as was the case for TSa in Table VII. This suggests that participants are confused about temporal API specifications. Two approaches might reduce this type of error: better education about LTL specs and the inclusion of temporal constraint information in JavaDocs.

c) **Study design errors:** Approximately 19% of the error responses indicated an issue with the study design or UI; this was observed for 34% of the participants. Within this category, the mistake most responses had was providing a correct explanation but selecting the wrong choice (SDa). One possible reason was that participants misunderstood the purpose of the task, which asked them to agree/disagree with the SpecForge answer instead of agree/disagree with the given statement. Another potential reason was that they just clicked the wrong button accidentally. The other category, incorrect knowledge transfer (SDb), meant the workers used their incorrect answers on some property instances to justify their answers on others.

That the dominant category demonstrated correct understanding but a wrong click implies the need for an improved study design to make the tasks clearer to workers. It also implies that the use of natural language processing to evaluate the congruence between quantitative and qualitative responses could serve as a useful quality control mechanism.

d) **Unclear errors:** The dominant error class includes nonsense or confusion in the response; this represents 37% of the error responses and 15% of the participants. Many of

TABLE VII
CROWD ERROR CATEGORIES FOR OUR STUDIES AND AN EXAMPLE ERROR FOR EACH CATEGORY.

Class	Code	Category	Example
API Doc. Error	APIa	Method relation	*"These are opposite, unrelated operations."*- Misunderstood relationship between `StackAR` methods in property [push(Object o) *AP* pop()].
	APIb	Constructor usage	*"In HashSet libray, when using ADD, it is acceptable to use HASHSET IMMEDIATELY afterward."*
	APIc	Overlooked certain method	*"[A] stack cannot be full after its been made logically empty."*- For the property [makeEmpty() *AF* isFull() = true], user overlooks that elements can be added between these calls.
	APId	Method return value	*"Returns the same value as the hasMoreTokens method."*- Confusion about return value in the property [hasMoreTokens() = true *NF* countTokens()].
	APIe	Parameter	*"if remove(Object o) returns false it means that o is not contained into the set, and an immediate call to remove(Object o) will return false not true."*
True Spec Error	TSa	LTL/True spec definition	*"Once all elements are cleared [then] the set is empty."*- Misunderstood method order in property [isEmpty() = true *AIF* clear()].
	TSb	Bad practice	*"Bad programming practice, but you can still do it."*
	TSc	Single instance requirement	*"Well if you wanted to create a second token for a different sting you might call it again."*- Confused about task that specifies one object instance.
Study Design Error	SDa	Misunderstanding what to agree/disagree or wrong click	*"I see no reason why you could not use counttokens right after setting up the tokens."*- Machine's answer for [StringTokenizer(String str) *NIF* countTokens()] is false. User correct reasoning, but user's property response indicates the opposite.
	SDb	Incorrect knowledge transfer	*"No, based on response on 1 and 2, it is not recommended to to so."* User explanation based on previous questions.
Unclear	Ua	Nonsense response	*"I THINK THIS IS THE CORRECT ANSWER."*
	Ub	Unsure	*"there may be changes made in between the two calls though I do not see a way to make these changes within StringTokenizer so I am quite unsure but am guessing that this is not [false] because a false measurement means there is nothing left to return a true."*

TABLE VIII
DISTRIBUTION OF PERCENT AND NUMBER OF INCORRECT RESPONSES ACROSS THE ERROR CATEGORIES FROM TABLE VII.

Code	`HashSet`	`StTokenizer`	`StackAr`	**Total**
API Error	**29%(53)**	**18%(32)**	**19%(42)**	**22%(127)**
APIa	13%(23)	3%(5)	10%(22)	9%(50)
APIb	3%(5)	9%(17)	3.00%(6)	5%(28)
APIc	3%(6)	4%(7)	5%(11)	4%(24)
APId	4%(8)	1%(2)	1%(3)	2%(13)
APIe	6%(11)	1%(1)	0%(0)	2%(12)
True Spec	**19%(35)**	**38%(68)**	**11%(24)**	**22%(127)**
TSa	16%(29)	30%(54)	3%(7)	15%(90)
TSb	1%(2)	4%(8)	6%(14)	4%(24)
TSc	2%(4)	3%(6)	1%(3)	2%(13)
Design	**29%(53)**	**22%(39)**	**10%(21)**	**19%(113)**
SDa	28%(52)	19%(35)	9%(20)	18%(107)
SDb	1%(1)	2%(4)	0%(1)	1%(6)
Unclear	**23%(42)**	**23%(41)**	**60%(132)**	**37%(215)**
Ua	23%(42)	21%(38)	59%(129)	36%(209)
Ub	0%(0)	2%(3)	1%(3)	1%(6)
Total	100% (183)	100% (180)	100% (219)	100% (582)

these responses simply repeated the given temporal statement or copied text from the API docs (Ua). Others explicitly stated that they were not sure about their answer (Ub).

This result indicates a need for better quality control mechanisms to improve the response quality, such as natural language processing as previously mentioned. We also notice that these responses are disproportionally present in the `StackAr` API, which is also the API on which the experts had the most disagreement, pointing to a possible documentation issue.

Summary: The dominant category for the crowd's mistakes was *unclear*, indicating a need for more research and improved quality control mechanisms. However, the fact that ambiguities in API documentation were not the dominant source of errors may suggest documentation can be a reliable source for composing LTL specifications for Java APIs.

C. RQ3: Why not just use experts?

Section V-A (RQ1) has shown that the crowd can improve machine-mined specs. To gain a deeper insight into the crowd's performance, we compared the accuracy of SpecForge+CrowdSpec, to experts individually, experts who vote using a majority rule, and experts who discuss the properties (equivalent to the ground truth).

Table IX lists the results. For `HashSet` the SpecForge accuracy is 97.04%. Adding the crowd (CrowdSpec), this increases to 98.03%. Independently, the three experts achieve accuracies better than SpecForge and SpecForge+CrowdSpec, ranging from 98.22% - 99.61%. When using voting to identify the winner, the expert accuracy is 98.42%. After discussion, the experts' accuracy is 100% (by definition). For `StringTokenizer`, SpecForge and SpecForge+CrowdSpec underperform compared to the experts individually and the experts combined (by voting and by discussion). For `StackAr`, SpecForge and SpecForge+CrowdSpec outperform Expert 1 and Expert 2 but not Expert 3, experts voting, or experts discussing.

To determine if SpecForge+CrowdSpec can significantly outperform experts, we performed a pairwise accuracy comparison using McNemar's test on SpecForge+CrowdSpec, experts voting, and experts discussion. Table X presents the p-value for each pair of techniques to infer specs. We find that the accuracy of SpecForge+CrowdSpec and experts voting are similar: there is no significant difference in two libraries, `HashSet` and `StackAr`. Between SpecForge+CrowdSpec and discussing experts, we found that discussing experts

TABLE IX
COMPARISON OF ACCURACY BETWEEN SPECFORGE, CROWDSPEC, AND EXPERTS FOR EACH LIBRARY

API	SpecForge	SF+ CrowdSpec	Expert1	Expert2	Expert3	Experts Voting	Experts Discussing
HashSet	97.04%	98.03%	99.61%	98.32%	98.22%	98.42%	100%
StTokenizer	91.15%	93.49%	97.14%	97.92%	98.44%	100.00%	100%
StackAr	98.50%	98.50%	98.17%	96.50%	98.67%	98.67%	100%

TABLE X
ACCURACY COMPARISON OF CROWDSPEC, EXPERTSVOTING, EXPERTSDISCUSSION FOR EACH LIBRARY

	SpecForge+CrowdSpec versus	
	ExpertsVoting	ExpertsDiscussion
HashSet	0.618	***<0.001
StringTokenizer	***<0.001	***<0.001
StackAr	1.000	**0.004

*α=0.1, **α=0.01, *** α=0.001

consistently outperformed CrowdSpec for all three libraries. This implies that the value of using experts is realized during discussion.

Summary: There is no substantial difference in accuracy between experts voting and the SpecForge+CrowdSpec (which is essentially the crowd voting). Instead of combining independent expert opinions with voting, experts are much more useful when they can discuss their disagreements. However, if discussion is not an option, for two of the three APIs, SpecForge+CrowdSpec performs statistically as well as three voting experts.

VI. RELATED WORK

Crowd-sourcing for software engineering. Crowdsourcing, and specifically microtask crowdsourcing [25], has been shown to be effective for tasks related to software engineering, such as building software [34], testing [35], [36], determining the impact of code smells [37], evaluating website usability [38], verifying software [39], and program synthesis [15]. In particular, Amazon's Mechanical Turk has been used for several software engineering tasks (e.g., [40], [38], [35]), with varied success [39].

CrowdMine (proposed [41] and detailed in Chapter 6 of Li's thesis [40]) is a closely related work in the context of digital design rather than software APIs. CrowdMine, like CrowdSpec, uses MTurk to mine temporal specifications. There are three key differences: (1) CrowdMine presents workers with traces instead of documentation, which we believe is a poor match for human workers, (2) it relies on gamification for quality control, and (3) it does *not* use the crowd to augment existing techniques as we do with SpecForge.

Maintaining quality in crowdsourcing. Quality control is important in crowdsourcing as the crowd is an unknown population and gaming can severely impact result quality. The success of crowdsourcing on MTurk relies on finding qualified participants. The Pew Research Center has found that MTurk workers are well-educated [42], which echoes the characteristics of our study participants, where 67% have a college degree. Pastore et al. report experiments with MTurk where qualified programmers were found to be six times better at spotting bad program assertions than open call crowdsourcing with a general population [43]. They also noted the complexity of training the crowd to achieve useful results.

Researchers performing web studies have explored different ways to improve response quality. One effective strategy to filter out random clickers is to identify when responses are uniformly distributed and likely to be made by bots [31]. Other work uses clickstream data to cluster similar users and identify fraudulent users as outliers [44]. CrowdSpec uses a timer to identify random clickers, which is more rudimentary, but does identify workers who answer HITs haphazardly. CrowdSpec also uses gold standard questions, which have been shown to identify workers who may not be paying attention [45], [26]. Finally, research on survey design has found that participants are more likely to be careful when they perceive they are contributing to research [28]; CrowdSpec also uses this strategy.

Mining specifications from documentation. Text mining and NLP techniques have been applied to API documentation for the purpose of supporting migration between APIs [46], inferring parameter constraints from method descriptions [47], and to infer resource specifications [48]. The most related approach to our work is ICON [49].

ICON is a machine learning and NLP technique that infers temporal constraints from API documentation with precision and recall of 79% and 60%, respectively, using three APIs different from the ones we studied. ICON considers four temporal properties: followed by, preceded by and their negations. In contrast, we consider a super-set of these properties. The precision and recall of SpecForge+CrowdSpec is higher than that of ICON, but it is not immediately clear why. We offer three possible explanations: (1) the crowd is more accurate than NLP techniques, (2) the API libraries we use are easier than those used in the ICON evaluation, or (3) starting the crowd with a preconception of true specs based on SpecForge leads to better results and ICON would see similar improvements if it also used this information.

Characterizing Software APIs. Our work assumes that the API documentation is of high quality. However, Uddin and Robillard found that ambiguity, incompleteness, and inaccuracy are typical issues in API documentation [50]. One way to cope with these issues is to use prior techniques to improve the API documentation first, before applying CrowdSpec. For example, we can use work by Treude et al. [51] who used Stack Overflow as a source of crowd knowledge about an API to improve API documentation. Alternatively, we could

generates code examples to enhances API documentation [52], or enrich documentations with API patterns [53].

API documentation contains several knowledge types, as captured in the work of Maalej and Robillard [54]. The *directives* type is the one that we believe to be the most valuable in our work, as it is more complementary to the types used by SpecForge (*control-flow* and *patterns*).

VII. Discussion

Scalability is an important consideration in this work. For each pair of methods in an API we created one task. Thus, the number of tasks grows quadratically with the size of the API. The experiments in this paper consider all six property types and all pairs of methods for three APIs. Moving forward, techniques will be needed to improve the scalability and we have evidence that this is possible. For example, since the AF, AIP, and AIF properties will always be false, we could reduce the size of the tasks or the number of tasks by 50%.

a) Cost effectiveness: Although a crowd has a higher availability and a lower hourly rate than experts, our experiments do not address the cost comparison between the two groups. For example, in our study the experts were unpaid, and unlike the crowd workers, were not recruited for the study. Future work should explore the cost trade-off directly.

b) Implications for Spec Mining: Specification mining based on traces is inexpensive, but is ultimately an incomplete approach. Using other sources of information, such as documentation, can help with this. We showed that CrowdSpec successfully elicits highly accurate specs from the crowd by presenting workers with information from JavaDoc pages.

Our RQ1 results show that there is no significant difference in accuracy between *familiar* and *unfamiliar* crowd groups. This result decreases the participant qualifications bar for the design of effective crowd-based spec miners, and thereby indicates that such miners can be highly scalable. Moreover, the impact of Java familiarity on accuracy shows that a crowd's performance is not affected by their prior knowledge of the API. This hints that if crowd is properly trained before the study, they are likely to be competent for an API they do not know.

Some participant responses (4%) noted *"bad programming practice"* as a rationale for incorrectly labeled property instance. Access to knowledge about best practice is a unique advantage of a crowd-based approach. We think that spec miners can take advantage of this by using the crowd to mine not just true/false, or hard, specs; but, also soft specs, such as specs indicative of good/bad programming practice.

RQ3 indicates that experts gain significant value from discussion. This raises the question of whether a crowd's performance would likewise improve through discussion. In our future work we plan to answer this question by experiments with different approaches to stimulate effective crowd discussion.

c) Implications for Crowdsourcing Software Engineering: Our success at getting the crowd to create accurate specs is in contrast with prior work that found MTurk unsuitable for verification tasks [39]. This perhaps emphasizes the need of crowd control strategies, like those employed by CrowdSpec, and appropriate qualification tests, to get access to a crowd qualified to perform complex software engineering tasks. We posit that such a crowd exists on the MTurk marketplace, so the challenges are participant screening and quality control.

d) Applicability to other APIs: We chose well-known Java APIs so we can compare CrowdSpec to prior mining approaches and because they are real. As designed, our study presents a greater challenge to the crowd: improving more accurate specifications. The three APIs considered in this paper are all well-documented APIs used in thousands of real Java projects. Beyond these APIs, our approach is applicable as long as the API is accompanied by documentation. The process of decomposing an API into tasks would be the same.

e) Applicability to other properties: Would crowdsourcing other property types yield the same results? We think the results will differ. For example, in our studies the crowd did poorly on the never-immediately-followed-by (NIF) property instances as compared to the other properties. We think the difference lies in the essential complexity of properties because even experts found NIF to be the most difficult property type.

VIII. Threats to validity

Conclusion. For each study, we collected at most five responses and used a majority rule method for selecting the crowd opinion. This approach may have low statistical power. The impact may be low crowd accuracy; future work will include statistical tests to determine when the crowd has reached consensus, as is done with Automan [55].

Internal. The results are subject to self-selection bias. The participants chose which, and in what order, to complete the tasks. The study results may be subject to history effects due to the study context on MTurk. The studies were deployed sequentially with at least one week in between each study, so the study circumstances were different.

External. We evaluated CrowdSpec using three Java APIs, `HashSet`, `StringTokenizer`, and `StackAr`. Our results may not generalize to other APIs. Although there is some structural overlap with other Java APIs in that all APIs have a constructor that must precede all other API methods.

IX. Conclusion

This paper proposes that human intelligence can play a role in specification mining, and that humans can be brought back into the spec creation loop in the form of a crowd that performs specification mining micro-tasks based on API documentation.

We show that the crowd achieves an accuracy of 93.5% – 98.5% when composed with SpecForge, with substantial gains in recall. We consider why the crowd makes mistakes and explore the potential of the crowd to correct experts, noting that the value of the experts over crowds is only realized when they are allowed to discuss specs.

Acknowledgments

This work is supported in part by NSF #1446932 and #1645136, and the Harpole-Pentair endowment to Iowa State University.

REFERENCES

[1] F. Song and T. Touili, "Model-checking software library api usage rules," in *International Conference on Integrated Formal Methods*. Springer, 2013, pp. 192–207.

[2] T. Fertig and P. Braun, "Model-driven testing of restful apis," in *Proceedings of the 24th International Conference on World Wide Web*. ACM, 2015, pp. 1497–1502.

[3] D. Gopinath, M. Z. Malik, and S. Khurshid, "Specification-based program repair using sat," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2011, pp. 173–188.

[4] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford, "Automated api property inference techniques," *IEEE Trans. Softw. Eng.*, vol. 39, no. 5, pp. 613–637, May 2013.

[5] 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.

[6] T.-D. B. Le and D. Lo, "Deep Specification Mining," in *ISSTA*, 2018.

[7] S. S. Emam and J. Miller, "Inferring Extended Probabilistic Finite-State Automaton Models from Software Executions," *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, vol. 27, no. 1, pp. 4:1–4:39, Jun. 2018.

[8] T.-D. B. Le, X.-B. D. Le, D. Lo, and I. Beschastnikh, "Synergizing specification miners through model fissions and fusions (t)," in *ASE*, 2015.

[9] C. Lemieux, D. Park, and I. Beschastnikh, "General LTL Specification Mining," in *ASE*, 2015.

[10] D. Lo, L. Mariani, and M. Pezzè, "Automatic steering of behavioral model inference," in *ESEC/FSE*, 2009.

[11] D. Lo and S. Maoz, "Scenario-based and Value-based Specification Mining: Better Together," in *ASE*, 2010.

[12] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy, "Inferring models of concurrent systems from logs of their behavior with csight," in *ICSE*, 2014.

[13] O. Legunsen, W. U. Hassan, X. Xu, G. Rosu, and D. Marinov, "How good are the specs? a study of the bug-finding effectiveness of existing java api specifications," in *ASE*, 2016.

[14] M. Pradel and T. R. Gross, "Automatic generation of object usage specifications from large method traces," in *ASE*, 2009.

[15] R. A. Cochran, L. D'Antoni, B. Livshits, D. Molnar, and M. Veanes, "Program boosting: Program synthesis via crowd-sourcing," *SIGPLAN Not.*, vol. 50, no. 1, pp. 677–688, Jan. 2015. [Online]. Available: http://doi.acm.org/10.1145/2775051.2676973

[16] P. Sun and K. T. Stolee, "Exploring crowd consistency in a mechanical turk survey," in *Proceedings of the 3rd International Workshop on CrowdSourcing in Software Engineering*. ACM, 2016, pp. 8–14.

[17] "Class StackAr," http://www.cs.umd.edu/class/fall2004/cmsc433/projects/p2/javadoc/StackAr.html, accessed: February 24, 2017.

[18] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in Property Specifications for Finite-state Verification," in *ICSE*, 1999.

[19] 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 Transactions on Software Engineering (TSE)*, vol. 41, no. 4, pp. 408–428, April 2015.

[20] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst, "Leveraging existing instrumentation to automatically infer invariant-constrained models," in *FSE*, 2011.

[21] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das, "Perracotta: Mining temporal API rules from imperfect traces," in *ICSE*, 2006.

[22] M. Pradel, C. Jaspan, J. Aldrich, and T. R. Gross, "Statically checking api protocol conformance with mined multi-object specifications," in *ICSE*, 2012.

[23] 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.

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

[25] K. Mao, L. Capra, M. Harman, and Y. Jia, "A survey of the use of crowdsourcing in software engineering," *RN*, vol. 15, no. 01, 2015.

[26] R. Snow, B. O'Connor, D. Jurafsky, and A. Y. Ng, "Cheap and fast—but is it good?: evaluating non-expert annotations for natural language tasks," in *Proceedings of the conference on empirical methods in natural language processing*. Association for Computational Linguistics, 2008, pp. 254–263.

[27] K. T. Stolee, J. Saylor, and T. Lund, "Exploring the benefits of using redundant responses in crowdsourced evaluations," in *Proceedings of the Second International Workshop on CrowdSourcing in Software Engineering*. IEEE Press, 2015, pp. 38–44.

[28] J. A. Krosnick, "Response strategies for coping with the cognitive demands of attitude measures in surveys," *Applied cognitive psychology*, vol. 5, no. 3, pp. 213–236, 1991.

[29] M. Chi, K. VanLehn, D. Litman, and P. Jordan, "An evaluation of pedagogical tutorial tactics for a natural language tutoring system: A reinforcement learning approach," *International Journal of Artificial Intelligence in Education*, vol. 21, no. 1-2, pp. 83–113, 2011.

[30] A. Shiel, "Conflict crowdsourcing: Harnessing the power of crowdsourcing for organizations working in conflict," 2013.

[31] S.-H. Kim, H. Yun, and J. S. Yi, "How to filter out random clickers in a crowdsourcing-based study?" in *Proceedings of the 2012 BELIV Workshop: Beyond Time and Errors - Novel Evaluation Methods for Visualization*, ser. BELIV '12. New York, NY, USA: ACM, 2012, pp. 15:1–15:7. [Online]. Available: http://doi.acm.org/10.1145/2442576.2442591

[32] "CrowdSpecMine: Supporting materials for submission," https://bestchai.bitbucket.io/crowdspecmine-eval/.

[33] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *Biometrics*, vol. 33, no. 1, pp. 159–174, 1977. [Online]. Available: http://www.jstor.org/stable/2529310

[34] T. D. LaToza, W. B. Towne, C. M. Adriano, and A. van der Hoek, "Microtask programming: Building software with a crowd," in *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '14. New York, NY, USA: ACM, 2014, pp. 43–54. [Online]. Available: http://doi.acm.org/10.1145/2642918.2647349

[35] E. Dolstra, R. Vliegendhart, and J. Pouwelse, "Crowdsourcing gui tests," in *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, March 2013, pp. 332–341.

[36] M. Nebeling, M. Speicher, and M. C. Norrie, "Crowdstudy: General toolkit for crowdsourced evaluation of web interfaces," in *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, ser. EICS '13. New York, NY, USA: ACM, 2013, pp. 255–264. [Online]. Available: http://doi.acm.org/10.1145/2494603.2480303

[37] K. T. Stolee and S. Elbaum, "Exploring the use of crowdsourcing to support empirical studies in software engineering," in *International Symposium on Empirical Software Engineering and Measurement*, 2010.

[38] D. Liu, R. G. Bias, M. Lease, and R. Kuipers, "Crowdsourcing for usability testing," *Proceedings of the American Society for Information Science and Technology*, vol. 49, no. 1, pp. 1–10, 2012.

[39] T. W. Schiller and M. D. Ernst, "Reducing the barriers to writing verified specifications," *SIGPLAN Not.*, vol. 47, no. 10, pp. 95–112, Oct. 2012. [Online]. Available: http://doi.acm.org/10.1145/2398857.2384624

[40] W. Li, "Specification mining: New formalisms, algorithms and applications," Ph.D. dissertation, EECS Department, University of California, Berkeley, Mar 2014. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-20.html

[41] W. Li, S. A. Seshia, and S. Jha, "Crowdmine: Towards crowdsourced human-assisted verification," in *DAC*, 2012.

[42] P. Hitlin, "Research in the crowdsourcing age, a case study," http://www.pewinternet.org/2016/07/11/research-in-the-crowdsourcing-age-a-case-study/, 7 2016.

[43] F. Pastore, L. Mariani, and G. Fraser, "Crowdoracles: Can the crowd solve the oracle problem?" in *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, March 2013, pp. 342–351.

[44] G. Wang, T. Konolige, C. Wilson, X. Wang, H. Zheng, and B. Y. Zhao, "You are how you click: Clickstream analysis for sybil detection," in *Proceedings of the 22Nd USENIX Conference on Security*, ser. SEC'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 241–256. [Online]. Available: http://dl.acm.org/citation.cfm?id=2534766.2534788

[45] C. Callison-Burch, "Fast, cheap, and creative: evaluating translation quality using amazon's mechanical turk," in *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing: Volume 1-Volume 1*. Association for Computational Linguistics, 2009, pp. 286–295.

[46] R. Pandita, R. P. Jetley, S. D. Sudarsan, and L. Williams, "Discovering likely mappings between apis using text mining," in *Source Code Analysis*

and Manipulation (SCAM), 2015 IEEE 15th International Working Conference on. IEEE, 2015, pp. 231–240.

[47] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar, "Inferring method specifications from natural language API descriptions," in *ICSE*, 2012.

[48] H. Zhong, L. Zhang, T. Xie, and H. Mei, "Inferring resource specifications from natural language API documentation," in *ASE*, 2009.

[49] R. Pandita, K. Taneja, L. Williams, and T. Tung, "ICON: Inferring temporal constraints from natural language api descriptions," in *Proc. 32nd ICSME*, 2016.

[50] G. Uddin and M. P. Robillard, "How api documentation fails," *IEEE Software*, vol. 32, no. 4, pp. 68–75, July 2015.

[51] C. Treude and M. P. Robillard, "Augmenting API Documentation with Insights from Stack Overflow," in *ICSE*, 2016.

[52] R. P. L. Buse and W. Weimer, "Synthesizing api usage examples," in *ICSE*, 2012.

[53] J. Fowkes and C. Sutton, "Parameter-free probabilistic api mining across github," in *FSE*, 2016.

[54] W. Maalej and M. P. Robillard, "Patterns of Knowledge in API Reference Documentation," *IEEE Transactions on Software Engineering*, vol. 39, no. 9, pp. 1264–1282, Sept 2013.

[55] D. W. Barowy, C. Curtsinger, E. D. Berger, and A. McGregor, "Automan: A platform for integrating human-based and digital computation," *ACM SIGPLAN Notices*, vol. 47, no. 10, pp. 639–654, 2012.