A Pattern-Based Approach to Parametric Specification Mining

Giles Reger , Howard Barringer, David Rydeheard
University of Manchester

***Abstract*—This paper presents a technique for using execution traces to mine parametric temporal specifications in the form of quantified event automata (QEA) - previously introduced as an expressive and efficient formalism for runtime verification. We consider a pattern-based mining approach that uses a pattern library to generate and check potential properties over given traces, and then combines successful patterns. By using predefined models to measure the tool's precision and recall we demonstrate that our approach can effectively and efficiently extract specifications in realistic scenarios.**

I. Introduction

This paper considers the mining of quantified event automata (QEA) [2] from execution traces. QEA were previously introduced for runtime verification [5] and consist of first-order quantification over variables occurring in a form of finite automaton. This formulation allows us to describe properties not just in terms of what events occur in a trace, but also how data values associated with events are related within a trace.

We are concerned with temporal specifications over events parameterised with data values, i.e. *parametric specifications*. A specification may deal with data parameters in two orthogonal ways - in a *quantified* or *free* manner. An example of the former would be to say that a file should only be read when open by specifying that event $\texttt{open}(f)$ occurs before $\texttt{read}(f)$ for every file f; an example of the latter would be to say that a counter is strictly increasing by specifying that if $\texttt{count}(x)$ follows $\texttt{count}(y)$ then $x > y$. QEA addresses both approaches but this paper only considers the quantified approach, leaving methods for dealing with free variables to future work.

We consider whether QEAs can be used in specification mining ([9], [17]), i.e. given a set of traces of events from runs of a system, can we build QEA specifications of the system. If so, we can mine specifications which previous systems cannot, due to increased expressiveness. Previous parametric specification mining approaches have extended techniques such as *stage-merging* and *active learning* from the propositional to parametric setting, either treating parameters in a quantified or free manner. Here, we choose an approach particularly suited to QEA, namely pattern-based mining [18], [6], where a collection of patterns is checked against the traces. This is a verification process: if the patterns are expressed as QEA, the verification process is that described in [2].

Pattern-based specification mining usually considers only small and simple patterns, so that the checking of traces is sufficiently fast. However, we wish to mine specifications of some complexity. To bridge this gap, methods by which patterns may be combined to form complex specifications are introduced [6]. We consider methods for combining QEA using what we call "open automata" as our notion of pattern.

Our pattern-based approach to mining parametric specifications consists of three stages:

1) *Generating.* A pattern library and candidate alphabet are combined to produce a set of possible patterns.
2) *Checking.* These patterns are checked against a set of given traces.
3) *Combining.* The successful patterns are combined to form a specification.

This general approach is not new (for example [6], [7], [18]), however, we are the first to apply it in a *parametric* setting, although the work of [12] is comparable as discussed in Section VI. The part of this process that deals with quantified parameters is the checking stage - based on a runtime verification algorithm for QEA. This consists of projecting traces of events into subtraces for given assignments to quantified variables and detecting patterns in these. A similar projection-based approach was taken in [8] using an automata-learning instead of a pattern-based approach.

Contributions. We make two main contributions: firstly, adapting the generate-and-check style approach to the quantified parametric setting in a general way, i.e. not specific to any set of patterns; and secondly, introducing the concept of open automata as a method for soundly combining small patterns to build specifications. A more detailed presentation of this work is given in [16].

Outline. We first present *quantified event automata* (Sec. II) and then introduce our notion of pattern (Sec. III) and how this is used to mine QEA (Sec. IV). We then evaluate our approach (Sec. V) and discuss related work (Sec. VI).

II. Quantified event automata

We introduce QEA using an example we will use throughout the paper and refer the reader to [2] and [16] for further details.

Figure 1 presents an example QEA for the usage of files - they must be opened to be read or written to, should not be left open, and if deleted should not be used again. Shaded states represent final states and there is an implicit error state $q_\perp$ that is used if no transitions match. Consider the trace

$$\tau = \begin{array}{l}\texttt{open}(A).\texttt{open}(B).\texttt{read}(A).\texttt{write}(B).\texttt{close}(A).\\ \texttt{close}(B).\texttt{delete}(A)\end{array}$$

Part of this work was supported by the Engineering and Physical Sciences Research Council [grant number EP/P505208/1].

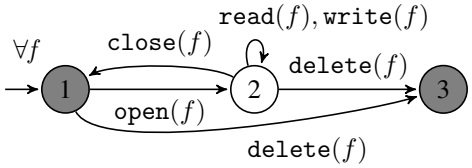

Fig. 1. A QEA for a resource-usage property.

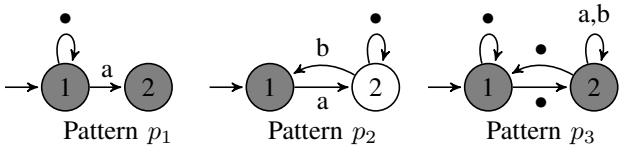

Fig. 2. A very small pattern library.

To interpret this trace we find all values that f can take - here A and B - and project the trace for each value:

$$[f \mapsto A]: \quad \sigma_1 = \texttt{open}(A).\texttt{read}(A).\texttt{close}(A).\texttt{delete}(A)$$
$$[f \mapsto B]: \quad \sigma_2 = \texttt{open}(B).\texttt{write}(B).\texttt{close}(B)$$

τ is accepted as both projections reach a final state in the automaton of Figure 1 with f appropriately replaced. Note how the domain of f is extracted from the trace. In general, QEAs can have multiple universal and existential quantifications.

III. A NOTION OF PATTERN

In this section we introduce a notion of pattern and pattern libraries. Patterns are combinable predicates on traces. An initial suggestion might be to use standard finite state automata, but it was shown [7] that these have an inadequate form of combination and we address this with a new form of automata.

A. Open Automata

We introduce *open automata* as a formalism for patterns.

An *open automaton* is a non-deterministic finite automaton that includes a special $\bullet$ symbol in its alphabet. Given a pattern p and a mapping from symbols to events φ we write $p(\varphi)$ for the *instantiated* pattern obtained by applying φ to symbols in p. The special hole symbol $\bullet$ can match any symbol not in the alphabet - for example the pattern p_1 in Fig. 2 with a replaced by $\texttt{delete}(A)$ accepts the projected traces σ_1 and σ_2 in Sec. II as all symbols that are not $\texttt{delete}(A)$ match $\bullet$.

We can combine open automata by synchronizing on hole symbols - this is equivalent to expanding each open automata by adding transitions for symbols not in its alphabet where there are $\bullet$ transitions and then performing standard intersection. This is sound and complete i.e. for any two open automata p_1 and p_2 we have $\mathcal{L}(p_1) \cap \mathcal{L}(p_2) = \mathcal{L}(p_1 \cap p_2)$ (see [16] for a proof).

B. Pattern libraries

Our approach uses a predefined pattern library, which will determine the specifications that may be mined. A k-pattern library is a list of patterns over a fixed set of k symbols, which we will take as the first k letters of the alphabet, i.e. $\{a, b, c, \ldots\}$. A pattern library is a set of k-pattern libraries. We have built an initial library of (around 150) useful patterns. This library is large as we need to capture many variations of simple patterns - with holes in different positions.

C. Pattern Checkers

To allow us to check many patterns at once we introduce the related concepts of *pattern checkers* and *event pattern checkers*. We present these ideas informally here and give a full description in [16]. We will demonstrate these constructs and their use using the pattern library given in Fig. 2.

A pattern checker is a structure that we construct from a set of k-patterns that realises a function from traces of symbols to sets of patterns. As p_2 and p_3 have the same size alphabet we can use them to construct the pattern checker given in Fig. 3.

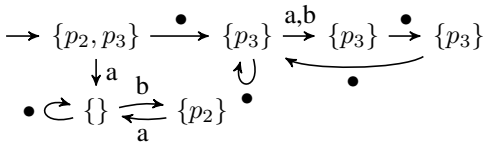

Fig. 3. A pattern checker for patterns p_2 and p_3.

An event pattern checker combines a pattern checker with a map from symbols to events and realises a function from traces of events to sets of instantiated patterns. Pattern checkers can be precompiled from pattern libraries and then combined with alphabets to produce event pattern checkers later.

Let the generation function G be a *generation function* that takes a pattern library and alphabet and produces a set of event pattern checkers for all values of k and mappings of symbols to events in the alphabet. For example, given the pattern library of Fig. 2 and alphabet $\{\texttt{open}(f), \texttt{read}(f), \texttt{write}(f), \texttt{close}(f), \texttt{delete}(f)\}$ the generation function G generates 25 event pattern checkers - 5 for p_1 and 20 for the p_2 and p_3 pattern checker.

IV. MINING PATTERNS

In this section we describe how we mine QEA using open automata as patterns. Our approach is split into three main stages, as outlined in Figure 4 and described in the following. The *mining process* takes as input a pattern library P, an alphabet of events $\mathcal{A}$, a list of quantifications Λ over variables X, a set of positive traces T_+, and a (possible empty) set of negative traces T_-.

If Λ is not given we can explore all possible combinations efficiently after the trace has been traversed, giving a separate specification per quantification list. It is generally difficult to generate negative traces and the monitoring process does not require them. We include negative traces as they serve as a an additional source of external knowledge that can contribute towards the final specification.

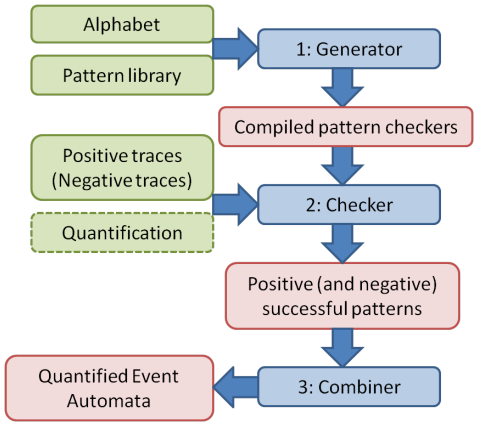

Fig. 4. An overview of the tool.

The generation stage is performed by the generation function G introduced in the previous section.

A. Checker

The checker uses the notion of projection and acceptance from QEA to produce sets of positive and negative successful patterns for each binding of quantified variables. A *scorecard* is a map from bindings to sets of patterns. We generate a scorecard $\mathsf{S}(\tau)$ for a trace τ as follows:

Definition 1 (Scoring)**.** *Given pattern library* P*, alphabet* $\mathcal{A}$*, set of quantified variables* X *and trace* τ*, let* $\mathsf{S}(\tau)$ *be*

$$\langle \theta \mapsto \mathsf{G}(\mathsf{P}, \mathcal{A}(\theta))(\tau \downarrow_{\mathcal{A}(\theta)}) \mid \mathsf{dom}(\theta) = X \wedge \tau \downarrow_{\mathcal{A}(\theta)} \neq \epsilon \rangle$$

where $\mathcal{A}(\theta)$ *is the alphabet instantiated with* θ *and* $\tau \downarrow_{\mathcal{A}(\theta)}$ *is* τ *with all events not in* $\mathcal{A}(\theta)$ *removed.*

This can be implemented directly to produce scorecards. However, it requires passing over each trace multiple times. In *Runtime Verification*, techniques have been introduced which may be adapted here to produce more efficient implementations. We refer the reader to work discussed in [2].

Once scorecards have been constructed the list of quantifications is used to extract successful patterns.

Definition 2 (Successful patterns)**.** *Given a scorecard* S*, the successful patterns for a list of quantifications* Λ *are given by* $\mathsf{pat}(\mathsf{S}, \langle\,\rangle, \Lambda)$*, defined as*

$$\begin{array}{lcl} \mathsf{pat}(\mathsf{S}, \theta, \forall x \Lambda') & = & \bigcap_{d \text{ in } \mathsf{dom}(\tau)(x)} \mathsf{pat}(\mathsf{S}, \theta \dagger \langle x \mapsto d \rangle, \Lambda') \\ \mathsf{pat}(\mathsf{S}, \theta, \exists x \Lambda') & = & \bigcup_{d \text{ in } \mathsf{dom}(\tau)(x)} \mathsf{pat}(\mathsf{S}, \theta \dagger \langle x \mapsto d \rangle, \Lambda') \\ \mathsf{pat}(\mathsf{S}, \theta, \epsilon) & = & \mathsf{S}(\theta) \end{array}$$

where $\mathsf{dom}(\tau)(x)$ *gives the values for* x *derived from* τ*.*

The positive and negative successful patterns are therefore

$$\mathsf{suc}_\alpha = \bigcap_{\tau \in T_\alpha} \mathsf{pat}(\mathsf{S}(\tau), \langle\,\rangle, \Lambda) \qquad \text{for } \alpha \in \{-, +\}$$

Consider the projected traces σ_1 and σ_2 in Sec. II. The set of positive successful instantiated patterns are:

$$p_1(\langle a \mapsto \texttt{delete} \rangle),\ p_2(\langle a \mapsto \texttt{open}, b \mapsto \texttt{close} \rangle),$$
$$p_3(\langle a \mapsto \texttt{read}, b \mapsto \texttt{write} \rangle),\ p_3(\langle a \mapsto \texttt{write}, b \mapsto \texttt{read} \rangle),$$

B. Combiner

The notion of combination for open automata is used to produce an instantiated open automaton that, with the quantification list, forms a QEA.

Definition 3 (Combination)**.** *Given a set of positive successful patterns* suc_+ *and a set of negative successful patterns* suc_-*. Their combination is given as*

$$\left(\bigcap_{p \in \mathsf{suc}_+} p \right) \cap \overline{\left(\bigcap_{p \in \mathsf{suc}_-} p \right)}$$

where the complement $\overline{q}$ *of an open automaton* q *is given by inverting accepting states.*

The result is an open automaton with events as symbols - the translation to QEA via removing holes is straightforward. The size of the pattern library can affect combination time as it will limit the number of successful patterns.

In our running example the combination of the successful patterns given above gives the QEA in Fig. 1

C. Limitations of the mining technique

There are three limitations to our approach. Firstly, we assume that all given traces are correct, therefore if a pattern is satisfied by 99% of a trace it will not be recorded. There are a number of statistical techniques that can be used to deal with this issue. Secondly, we must provide a pattern library - if the library is too general we may fail to extract a specification but if the library is too specific we may fail to generalise from the given traces. Thirdly, an alphabet must be supplied, requiring some prior knowledge of the system under inspection. This may be addressed with additional computation, as shown in [8]. This need not be a restriction, as in many applications we know the events whose behaviour we wish to mine.

Finally, [16] analyses the complexity of our approach. In summary, this grows quickly with the size of the alphabet and, to a lesser extent, the number of quantified variables - therefore these are the two limiting factors for performance.

V. Evaluation

In this section we establish the viability of our framework (implemented in `Scala`) by measuring its ability to extract specifications from traces generated from a range of models.

Experiments were carried out on an Apple Mac Pro with two 2.26GHz quad-core Intel Xeon processors and 16GB of memory. The pattern library used was built using a mixture of automatic generation and intuition and was developed independently from the chosen models and prior to evaluation.

A. Evaluation setup

Our evaluation will follow four steps:

1) Select a set of realistic reference models capturing a range of different kinds of specifications.
2) Randomly generate training and test traces, with different categories of training traces and both positive and negative test traces.

3) Extract a specification for each category.
4) Use the test traces to compute the precision and recall of the extracted specification.

We discuss these stages further below.

B. The models

We use ten models to represent realistic specifications we might expect to extract, these are described in Table I and can be grouped into the following domains:

- **Communication.** *James* is taken from [8] as a specification extracted from the Apache James mail server program. *Satellites* and *Commands* are (simple) specifications of correct communication between planetary rovers, inspired by the work of [3].
- **Java API.** *SocketOutput* and *ColIter* describe the correct usage of data structures in the Java API.
- **Concurrency.** *MutualExcl* and *LockOrder* describe common desired concurrent behaviour.
- **Drivers.** The last three models are based on rules described in the SDV tool [1] and were taken from [12] as interesting specifications to address.

As well as covering a variety of domains, these ten models also capture different levels of complexity in number of states, size of alphabet and number of quantified variables. Satellites is the only model that uses existential quantification.

C. Generating traces

We used the predefined models to generate traces for both training (i.e. mining) and testing. To explore how the mining process is effected by the quality of the data provided we consider two coverage criteria for traces. This is similar to the approach taken by [11], however is updated for our scenario with quantifications and non prefix-closed automata.

- State coverage - All non-ultimately failing states are visited at least once by a projection of the trace
- Path coverage - All paths to non-ultimately failing states are visited at least once by a projection of the trace

As we might expect traces containing more information to lead to more accurate specifications, we randomly generate short, medium and long traces for each coverage level. The sizes are chosen so that the short traces are minimal and long traces represent realistic usage. We generate 10 traces for training and 100 traces for testing. Table II gives the average length of the generated traces - the dashes indicate that no traces were produced as the coverage criteria could not be met.

D. Measurements

We evaluate our technique for accuracy and efficiency.

1) Accuracy: To measure the accuracy of the mining process we use the common precision-recall evaluation measures taken from the field of information retrieval. *Recall* measures the mined specification's ability to identify correct behaviours and is defined as the fraction of positive traces that are correctly accepted. *Precision* measures the extent to which incorrect traces are rejected by the mined specification and is defined as the fraction of accepted traces that were correctly accepted.

2) Efficiency: It is necessary to demonstrate that the technique can be applied to realistic traces, therefore we measure the time it takes to extract specifications, focussing separately on checking and combining times.

E. Results

The results of our evaluation are detailed in Table II, giving precision and recall results, and Table III, giving efficiency results. Overall the results are positive, with even the more complex specifications being reconstructed well.

There were only three cases where a specification was not identified. In two cases this was due to short traces not matching any patterns and in one case this was due to long traces matching disjoint patterns.

In general, mined specifications are larger than their reference models. In the extreme, the mined specification for ColIter with long traces and path coverage has 136 states, compared to the reference model's 10. This explosion in size is due to capturing unintended orderings between events. The mined specification for LockOrder with long traces and path coverage achieves full recall and precision even though it has 5 states and the reference model has 12. The reference model describes the order in which locks can be taken and contains symmetric behaviour relating to which lock is taken first - this introduces some non-determinism. The extracted model is a minimised version that makes use of this non-determinism to omit one half of the behaviours. This demonstrates that our technique can extract *concise* specifications.

We now discuss results for recall, precision and efficiency.

1) Recall: Generally, path-cover leads to better recall than state-cover, and longer traces lead to better recall. However, there are a few cases where this does not hold. In all of these cases the mined specification incorrectly conclude that loops must be unfolded a number of times as there are no short traces in their training sets. This demonstrates how over-precise patterns in the pattern-library can prevent useful generalisation, and the need for traces of varying lengths. In the case of James we have very poor recall for state-cover, this is because there is a single accepting state (when the protocol completes) and not all paths to that state are covered. Path-coverage only means that all paths are covered in the trace as a whole, not each individual per-binding subtrace.

There are four cases where we have zero recall (and precision) as the mined specifications reject all trace tests. In all cases this was due to loops not being exercised in any trace in the training set and then being used in every trace in the test set. In practice these training traces were unrealisticly short.

2) Precision: Mined specifications are generally precise. As expected, path coverage leads to better precision than state coverage. In some cases shorter traces lead to better precision. This is due to longer traces including 'noise' causing patterns with irrelevant parts to be matched and included.

3) Efficiency: Table III focuses on the long traces, path coverage category as it is the most complex. For checking there is a correlation with the size of alphabet (not length of traces) as this determines the number of event pattern checkers

TABLE I
TARGET SPECIFICATIONS, GIVING NUMBER OF STATES, SIZE OF ALPHABET AND NUMBER OF QUANTIFIED VARIABLES.

Name	Description	$\|Q\|$	$\|\mathcal{A}\|$	$\|X\|$
James	The SMTP protocol (without authentication) as used in the Apache JAMES mail server	7	5	1
Satellites	There exists a satellite that has established a communication link with all known field units.	4	2	2
Commands	Issued commands succeed and are only reissued after failure.	4	4	1
SocketOutput	A stream is used after being connected to a socket and not after that socket is closed.	6	3	2
ColIter	An iterator created from a collection is not used after the collection is updated.	10	5	2
MutualExcl	No lock should be held by two threads at the same time.	4	4	3
LockOrder	Locks are always taken in a consistent order.	12	4	2
IOCallDriver	The I/O stack must be setup before calling the IOCallDriver	4	2	1
KeAcquireSpinLock	Locks and releases of a spin lock alternate, with two alternate locking calls.	3	3	1
ZwRegistryCreate	Registry keys are created before being used, open when used and not used after being deleted.	5	5	1

TABLE II
TRACE LENGTHS AND RESULTS (S=STATE,P=PATH).

Model	Trace lengths								Results									
	Training						Test		Recall						Precision			
	Short		Medium		Long				Short		Medium		Long		Short		Medium		Long	
	S	P	S	P	S	P	+	-	S	P	S	P	S	P	S	P	S	P	S	P
Jam	12	22	31	23	410	140	137	829	0.55	0.56	0.23	0.75	0.05	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Sat	-	5	-	11	-	93	57	31	-	1.0	-	1.0	-	1.0	-	1.0	-	1.0	-	1.0
Com	-	12	-	44	-	1620	2526	2160	-	0.28	-	0.78	-	0.78	-	1.0	-	1.0	-	1.0
Soc	5	7	12	15	27	27	39	82	0.08	1.0	1.0	1.0	1.0	1.0	1.0	0.98	0.98	0.98	0.98	0.98
Col	10	-	68	43	709	94	39	122	0.01	-	0.82	0.97	0.87	0.4	1.0	-	0.94	0.82	0.94	0.97
Mut	1	3	-	26	-	100	10	40	-	1.0	-	0.64	-	-	-	1.0	-	1.0	-	-
Loc	9	16	13	23	-	88	10	18	0.0	0.0	1.0	1.0	-	1.0	0.0	0.0	1.0	1.0	-	1.0
IOC	2	8	-	42	-	2036	1318	1462	0.0	1.0	-	0.67	-	0.67	0.0	1.0	-	1.0	-	1.0
KeA	5	12	-	56	-	1518	951	907	0.09	1.0	-	1.0	-	1.0	1.0	1.0	-	1.0	-	1.0
ZwR	25	39	44	66	554	666	356	798	0.44	0.4	1.0	0.56	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0

TABLE III
EFFICIENCY RESULTS - TIMES IN SECONDS

Model	Times		Patterns passed
	Check	Combine	
James	1.98	11.68	79
Satellites	0.365	0.209	144
Commands	4.474	19.77	39
SocketOutput	0.515	0.299	22
ColIter	3.818	175.0	33
MutualExcl	24.38	2.761	1824
LockOrder	2.627	0.044	8
IOCallDriver	1.145	0.142	78
KeAcquireSpinLock	1.833	0.043	20
ZwRegistryCreate	2.435	1.568	341

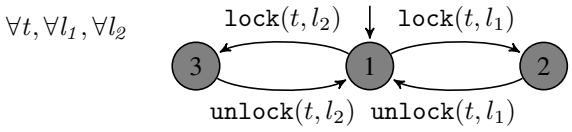

Fig. 5. The MutualExl property as a QEA.

that must be updated for each event. Note that the number of quantified variables also has an effect - demonstrated in the case of MutualExcl. For combining there is not a strong correlation with patterns passed. In this case of MutualExcl it takes under 3 seconds to combine 1824 patterns, yet in the case of ColIter it takes 175 seconds to combine 33 patterns (as the resultant specification has 136 states these 33 patterns are likely to be complex). Again this is due to the relative alphabet sizes - combining two open automata with the same symbols is linear and the likelihood of successful patterns sharing symbols increases greatly with shorter alphabets. As expected, the size of the alphabet plays a large part in deciding the time it takes to extract a specification. However, the pattern-library also plays a key part, it is likely that in the cases where combination was very expensive many of the successful patterns were redundant. One extension would be to detect and remove these redundant specifications.

F. Real world traces

Additional tests reported in [16] show that the tool can be applied to traces produced by real world applications. For example, a trace consisting of 14 million events relating to the usage of Java iterators was analysed in less than 20 minutes.

VI. RELATED WORK

The field of specification mining is growing and we briefly consider related work - [16] gives a more thorough review.

There are two other approaches that focus on mining parametric specifications with quantified variables. TARK [12] uses techniques from data-mining to identify predetermined quantified binary temporal rules with equality constraints (QBEC). Their focus is on extracting sets of rules rather than a single specification and therefore do not include a notion of combination. However, the use of support and confidence

allows them to address imperfections in traces. JMINER [8] mines parametric specifications with universally quantified variables with the restriction that an event name may only occur in the alphabet once. The sk-strings algorithm is used to infer a probabilistic finite state automata from *trace slices* using an approach similar to our trace projection. JMINER provides functionality for automatically discovering likely specification alphabets, whereas TARK considers all events in the trace. Previous techniques [18] could abstract over a single value by slicing the trace. Neither approach is able to mine the MutualExcl model (Fig. 5), as JMINER's underlying language cannot represent it and it cannot be decomposed into the binary rules of TARK. Additionally, neither could capture Satellites as it uses existential quantification. Our evaluation included examples of models that were mined by both of these tools.

There have also been approaches that extract unquantified parametric specifications with free variables and guard transitions from sets of traces. GkTail [13] combines the Daikon invariant-detection tool [4] with a state-merging technique based on the kTails approach to mine a form of Extended Finite State Machine. KLFA [14] extracts data recurrence patterns by augmenting the names of events with symbols representing discovered recurrence patterns and then applying the propositional kBehaviour [15] technique. Lo et al. [10] mine live sequence charts enriched with invariants by first identifying frequent charts and then using these to identify subtraces for invariant mining. In [11] Lo et al. explore whether such techniques that necessarily produce higher quality models than their propositional counterparts. They compare gkTail and KLFA with their underlying propositional algorithms kTails and kBehaviour and conclude that neither significantly improves precision or recall. There is, however, a fundamental difference between techniques that focus on free, rather than quantified, uses of parameters. The free variable approach augments a specification mined by a propositional technique, whereas the quantified variable approach uses quantifications to identify the propositional parts. Applying a propositional technique to the trace $\texttt{open}(1).\texttt{open}(2).\texttt{open}(3).\texttt{close}(1).\texttt{close}(3).\texttt{close}(2)$ would not produce the specification that correctly abstracts the data, i.e., $\forall f.\texttt{open}(f).\texttt{close}(f)$.

The pattern-mining approach taken in this paper has been previously explored in the context of propositional specification mining. The approach was first taken by Yang et al. [18] in the Peracotta tool and later extended by Gabel and Su [6], [7] in the Javert tool. We extend these approaches by introducing the concept of open automata.

VII. CONCLUSION

In this paper we demonstrate that QEA can be used in specification mining by describing and evaluating a pattern-based technique. We have established that our technique is good at extracting specifications that simulate complex scenarios.

This work is significant as the form of specifications that can be mined are richer than previous approaches. Importantly, this work also lays the foundations for further work allowing us to extend our specification mining technique to the full expressiveness of QEA, incorporating both free and quantified variables, along with transition guards and assignments. Extending our technique to the form of QEA described in [2] will involve updating our notion of open automata combination to reason about how variables are updated in holes. Two further extensions that should be considered are the automatic identification of likely alphabets by applying heuristics to traces or the source code that produces them, and introducing techniques for dealing with imperfect traces.

REFERENCES

[1] Windows Driver Development. http://msdn.microsoft.com/en-us/library/windows/hardware/ff551714\%28v=vs.85\%29.aspx.

[2] H. Barringer, Y. Falcone, K. Havelund, G. Reger, and D. E. Rydeheard. Quantified event automata: Towards expressive and efficient runtime monitors. In *FM*, pages 68–84, 2012.

[3] H. Barringer and K. Havelund. Tracecontract: a Scala DSL for trace analysis. In *Proc. of the 17th international conference on Formal methods*, pages 57–72, Berlin, Heidelberg, 2011.

[4] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 449–458, New York, NY, USA, 2000. ACM.

[5] Y. Falcone, K. Havelund, and G. Reger. A tutorial on runtime verification. In M. Broy and D. Peled, editors, *Summer School Marktoberdorf 2012 - Engineering Dependable Software Systems*. IOS Press, 2013.

[6] M. Gabel and Z. Su. Javert: fully automatic mining of general temporal properties from dynamic traces. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 339–349, New York, NY, USA, 2008. ACM.

[7] M. Gabel and Z. Su. Symbolic mining of temporal specifications. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 51–60, New York, NY, USA, 2008. ACM.

[8] C. Lee, F. Chen, and G. Roşu. Mining parametric specifications. In *Proceeding of the 33rd International Conference on Software Engineering (ICSE'11)*, pages 591–600. ACM, 2011.

[9] D. Lo, K. Cheng, and J. Han. *Mining Software Specifications: Methodologies and Applications*. Chapman and Hall/CRC Data Mining and Knowledge Discovery Series. Taylor & Francis Group, 2011.

[10] D. Lo and S. Maoz. Scenario-based and value-based specification mining: better together. *Autom. Softw. Eng.*, 19(4):423–458, 2012.

[11] D. Lo, L. Mariani, and M. Santoro. Learning extended fsa from software: An empirical assessment. *J. Syst. Softw.*, 85(9):2063–2076, Sept. 2012.

[12] D. Lo, G. Ramalingam, V. P. Ranganath, and K. Vaswani. Mining quantified temporal rules: Formalism, algorithms, and evaluation. *Sci. Comput. Program.*, 77(6):743–759, 2012.

[13] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 501–510, New York, NY, USA, 2008. ACM.

[14] L. Mariani and F. Pastore. Automated identification of failure causes in system logs. In *Proceedings of the 2008 19th International Symposium on Software Reliability Engineering*, ISSRE '08, pages 117–126, Washington, DC, USA, 2008. IEEE Computer Society.

[15] L. Mariani, F. Pastore, M. Pezzè, and M. Santoro. Mining finite-state automata with annotations. In D. Lo, S.-C. Khoo, J. Han, and C. Liu, editors, *Mining Software Specifications: Methodologies and Applications*, Data Mining and Knowledge Discovery. CRC Press, 2011.

[16] G. Reger, H. Barringer, and D. E. Rydeheard. A pattern-based approach to parametric specification mining. www.cs.man.ac.uk/~david/sm.html.

[17] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. Automated API property inference techniques. *IEEE Transactions on Software Engineering*, 39(5):613–637, 2013.

[18] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: mining temporal API rules from imperfect traces. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 282–291, New York, NY, USA, 2006. ACM.