Learning Temporal Specifications from Imperfect Traces Using Bayesian Inference

Artur Mrowca, Martin Nocker
Technical University of Munich, Germany
artur.mrowca@tum.de,martin.nocker@tum.de

Sebastian Steinhorst, Stephan Günnemann
Technical University of Munich, Germany
sebastian.steinhorst@tum.de,guennemann@in.tum.de

ABSTRACT

Verification is essential to prevent malfunctioning of software systems. Model checking allows to verify conformity with nominal behavior. As manual definition of specifications from such systems gets infeasible, automated techniques to mine specifications from data become increasingly important. Existing approaches produce specifications of limited lengths, do not segregate functions and do not easily allow to include expert input. We present *BaySpec*, a dynamic mining approach to extract temporal specifications from Bayesian models, which represent behavioral patterns. This allows to learn specifications of arbitrary length from imperfect traces. Within this framework we introduce a novel extraction algorithm that for the first time mines LTL specifications from such models.

KEYWORDS

specification mining, data-driven verification, bayesian inference, path merging

1 INTRODUCTION AND RELATED WORK

Software systems are increasing in complexity, resulting from growing numbers of dynamically interdependent components, distribution of software modules and interacting functionalities and features. Complete verification of such systems in terms of both functionality and communication is needed to ensure system reliability. Verification of software often uses specifications and model checking to identify locations with deviations from nominal behavior in a trace. Specifications are either defined manually, extracted automatically from program code or extracted from traces from execution. Manual approaches become intractable, due to high system complexity and limitations of the human cognition, yielding erroneous or incomplete specifications. Hence, there is the need for scalable automatic specification mining approaches. In this work we present *BaySpec*, a dynamic approach to extract formal specifications from historical traces, which is illustrated in Fig. 1. While existing approaches are mostly sequence-based (i.e. working directly on a trace), we propose to aggregate temporal and causal functional behavior in a machine learning model to subsequently extract formal Linear Temporal Logic (LTL) specifications from the model. The learning of Bayesian Networks (BN) to represent functional behavior is well studied in the Data Mining community. Therefore the focus of this work is on the extraction of specifications from trained models. We show, that this approach yields more precise and expressive specifications than existing approaches in a reasonable runtime. This is, as existing approaches are template based (i.e. find only known patterns), do not segregate functions, cannot be optimized incrementally and are less robust. Using BNs levers out those limitations as its probabilistic compact representation allows robust learning of

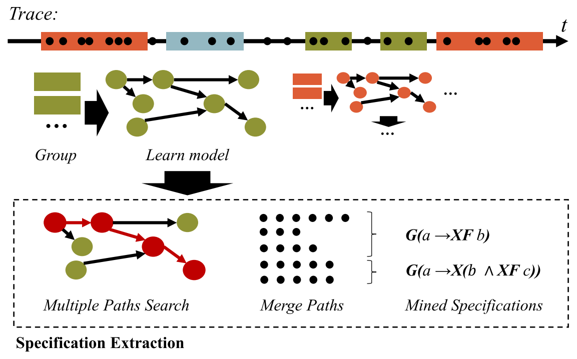


Figure 1: The proposed specification mining approach *BaySpec* starts from a given trace of events (indicated as circles), finds groups of similar behavior and uses those to train Bayesian Networks. By finding and merging most likely paths, specifications of appropriate strictness are found. The framed steps indicate the focus of this work.

causal event correlations, enables experts to interact and incremental improvement by adding samples. To enable extraction of meaningful specifications from such models, for the first time, an extraction approach is presented in this work. The basic idea is that sequential processes consist of events, where each event is a node in the BN and edges between nodes represent temporal causality between events (e.g. event A=a causes event B=b with prob. 0.3). Therefore, any likely path through the BN is a likely sequence of events which was observed in the data. Such likely paths represent likely behavior and are thus, potential specifications. Extraction is challenging due to the following reasons. *Valid LTL:* Translating a BN into LTL semantic is difficult as nodes represent probabilistic occurrences of sequences rather than strict paths (e.g. as in automatons). *Strictness:* Resulting specifications need to be meaningful, i.e. neither too strict nor too soft. *Complexity:* With growing size BNs increase in complexity. Thus, effective path inspection is required. *Noise:* Noisy data result in variations of probabilities and structure of the BN. This requires effective merging strategies to find valid specifications. *Path search:* Finding paths of maximal average likelihood requires to modify existing shortest path algorithms.

Related Work: "Specification mining" algorithms can be categorized as static [1, 7, 17, 18] or dynamic [2, 3, 5, 9–11, 13–15, 21]. Static miners infer specifications from program code, while dynamic miners extract specifications from simulation or execution traces. The scope of this work focuses on dynamic miners. Existing dynamic miners are the following. *Model-based:* In [2] specifications are extracted by learning probabilistic finite automata (FSA) that represent temporal, as well as data dependencies from traces of correct system behavior. This is extended in [14] by prior cleansing and clustering of traces. FSAs that satisfy binary properties of three different types are found by the approach in [3], which improves precision through refinement and coarsening. Such approaches produce automata, that are hard to interpret and whose complexity increases with growing functionality. This e.g. aggravates expert input. *Non-model-based:* Perracotta [21] mines two-event temporal patterns from execution traces. These patterns can then be chained together to form larger rules. Javert [10] also uses chaining rules to construct more complex specifications from simpler patterns. Such

DAC '19, June 2–6, 2019, Las Vegas, NV, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6725-7/19/06…$15.00
https://doi.org/10.1145/3316781.3317847

approaches miss out some rules and thus, deliver only partially complete solutions [14]. Also, a single violation of a pattern prevents it from being mined, thus, requiring perfect traces. Perracotta is extended in [13] to mine temporal properties of hardware designs. Response patterns between sequences of events are inferred in [15]. In contrast to our work, those approaches do not separate functional processes, making them prone to produce false positives. Daikon [9] infers invariants of values of program variables and in [11] Texada is introduced, which finds instances of user-provided property templates of arbitrary complexity. However, this only allows to find known patterns and misses out properties of any other structure.
In temporal assertion mining approaches were proposed for verification of hardware designs [4, 6, 19]. Unlike BaySpec, such approaches often exploit system design knowledge and tend to produce hard-to-read properties. Also, different than those approaches BaySpec is applicable to any behavioral process that can be expressed as a Bayesian Network.
Thus, existing approaches do not separate functionality, are less interpretable and are either unable to find properties of arbitrary length or exhibit high false positive rates. This makes them inapplicable for specification mining in imperfect traces that contain multiple functions. We tackle those issues with our approach *BaySpec*.
Contributions: First, we introduce *BaySpec*, an approach that learns BNs to robustly represent functional behavior of imperfect traces. Second, two approaches for extraction of specifications of appropriate strictness from BNs are proposed in Sec. 3 and Sec. 4. Third, existing algorithms only supports summation, for finding shortest paths. We require shortest paths in terms of averages and thus, in Sec. 2 for this we present an extended version of Yen's algorithm, that is generally applicable.
Overview: The proposed miner consists of the following steps (see Fig. 1). First, real world traces originate from multiple functional processes. Those traces are split into subtraces, each containing data from one function only, as specifications should be defined within one functional scope. Second, all subtraces of a group are used to train a BN that captures causality between events and represents temporal processes in an aggregated manner. This allows to reduce noise and allows for expert input at this step, e.g. by removing edges which are learned from noise. Third, as this BN now contains all branches of behavior of a functional process, individual specifications need to be extracted. This is done by finding the most probable paths in the BN. However, usually those paths contain behavior that is too strict, and thus, needs to be softened such that the level of confidence of found specifications is neither too high nor too low. We solve this by merging similar paths until a metric is met (approach 1: Sec. 3) or by additionally validating those on a second BN (approach 2: Sec. 4) learned from the same functional process.

2 FINDING MOST PROBABLE PATHS

We assume that functional segmentation was performed and a BN from one functional process is given. Next, in this section a shortest path search is presented that allows to extract the most probable behavior from such a model.

2.1 Mining Graph Conversion

Temporal and causal dependencies of functional processes are captured by the BN. As causalities between trace-events are represented as edges, a path with strong causal dependencies (i.e. high conditional probabilities) among its nodes is likely to represent a candidate specification of the BN's function. This is comparable to the extraction of single paths along various branches from a process model, with the advantage that temporal causality between events is captured. To be able to adjust strictness of specifications multiple such paths are required and found using path search. Existing search approaches for solving this, find the maximum a posteriori hypothesis (MAP) for particular RVs (best configuration of a BN),

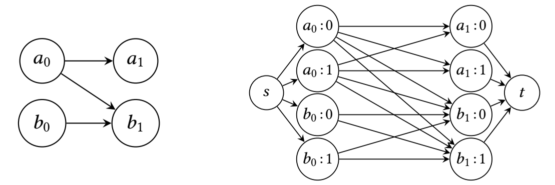


Figure 2: A Bayesian Network with two states 0 and 1 per node is shown on the left and the resulting *Mining Graph* after conversion is shown on the right.

which cannot be used here as the set of RVs has to be determined beforehand. Other algorithms for finding most likely paths, e.g. the Viterbi algorithm [20], are defined for one parent per node only and hence cannot find paths in BNs with multiple parents. We solved this by marginalizing out parent-nodes per candidate path to receive a weighted graph called *Mining Graph (MG)* (see Fig. 2). This graph can now be searched for likely paths, i.e. candidate specifications, using the approach proposed in this section. **Network Structure:** Our approach is valid for arbitrary BNs. However, to capture temporal specifications a defined structure is used that captures related events of multiple sources, as exemplified in Fig. 2 (left). There, each RV is an event in the trace, that can have multiple states and each dimension corresponds to one source that produces events (e.g. in Fig. 2 a and b). Interdepending events are connected via edges, as a subsequent events of the same source (e.g. a_0 and a_1). In real data this could be the states of a lamp e.g. $lamp_0 \in \{on, off\}$. Further, during parameter estimation we assume the BN to store histograms of the number of intermediate events seen between two RVs under given conditions. From this BN a Mining Graph is extracted as follows.
Definition of Mining Graph: Given a BN (G, Θ), with graph G and parameters Θ over RVs $\mathbf{X}$, a Mining Graph is a DAG $(\bar{V}, \bar{E})$, with

$$\bar{V} = \left(\bigcup_{i=1}^{|\mathbf{X}|} \bigcup_{j=1}^{|\Omega_{X_i}|} \vartheta_{ij} \right) \cup \{s, t\}$$

$$E^{(X)} = \left\{ \left(\vartheta_{im}, \vartheta_{jn}\right) \mid \vartheta_{im}, \vartheta_{jn} \in \bar{V} \wedge \left(X_i, X_j\right) \in E \right\}$$

$$E^{(s)} = \left\{ (s, \vartheta_{im}) \;\middle|\; \vartheta_{im} \in \bar{V}, X_i \text{ is the first instance of a RV} \right\}$$

$$E^{(t)} = \left\{ (s, \vartheta_{im}) \;\middle|\; \vartheta_{im} \in \bar{V}, X_i \text{ is the last (not first) instance of a RV} \right\}$$

$$\bar{E} = E^{(X)} \cup E^{(s)} \cup E^{(t)}$$

The set of vertices $\bar{V}$ consists of a vertex for each discrete value a RV $X_i \in \mathbf{X}$ can take and two additional vertices s and t representing an artificial start and terminal vertex, respectively. s and t are required as we define those as start and target nodes for the shortest path search. The set of edges $\bar{E}$ contains an edge between two vertices from $\bar{V}$ if their corresponding RVs X_i and X_j are connected in the BN. Vertex s is connected to all vertices of initial RVs per source (RVs with index 0). Vertex t is connected with all vertices whose RVs is the last instance per source. To avoid paths with a single vertex, vertices of a single-instance RV are not connected to t, prohibiting paths like (s, ϑ_{ij}, t).
Edge weights and Marginalization: Per edge in the MG a unique weight value is required, while in the BN the probability of a RV's value is given by multiple parents. Thus, to enable shortest path search in the MG, the following marginalization of parent RVs is required at each vertex in the BN to find its edge weights. For a target node X_j of an edge (X_i, X_j), let
$Y\left(e = \left(\vartheta_{im}, \vartheta_{jn}\right)\right) = \{Y_1, \ldots, Y_k\} := \mathrm{Par}(X_j) \setminus \{X_i\}$ be the set of all parent RVs without source node's RV X_i. The conditional probability $P(X_j \mid X_i)$ is defined as

$$P(X_j|X_i)=\sum_{Y_1} \cdots \sum_{Y_k} \left(P(X_j|X_i, Y_1, \ldots, Y_k) \cdot P(Y_1) \cdot \ldots \cdot P(Y_k) \right). \quad (1)$$

Let $\mathrm{Par}(X_i) := \left\{R_1, \ldots, R_p\right\}$. The marginal probability of $P(X_i)$ is defined by

$$P(X_i)=\sum_{R_1} \cdots \sum_{R_p} \left(P(X_i|R_1, \ldots, R_p) \cdot P(R_1) \cdot \ldots \cdot P(R_p) \right). \quad (2)$$

Algorithm 1:
Minimum Average Edge Weight Path (modified Dijkstra)
Input: ***graph G, start node ϑ_s, target node ϑ_t, root path π_r***
Output: ***minimum average edge weight path from ϑ_s to ϑ_t given π_r***

```
 1: $V^*$ = reachable($G$, start)
 2: $V^*$.topologicalSort( )
 3: for each vertex $v$ in $V^*$ do
 4:     distance[$v$] = new map(default = infinity)
 5:     previous[$v$] = new map(default = undefined)
 6: distance[$\vartheta_s$][$\pi_r$.edges] = $\pi_r$.probability
 7: while not $V^*$.isEmpty( ) do
 8:     $u$ = $V^*$.pop(0)
 9:     for each neighbor $v$ of $u$ do
10:         for each edges, avg in distance[$u$] do
11:             if $u$ is $s$ or $v$ is $t$ then
12:                 newAverage = distance[$u$][edges]
13:                 edges = edges−1
14:             else
15:                 newAverage = (edges · avg + weight($u$, $v$)) / (edges+1)
16:             if distance[$v$][edges+1] > newAverage then
17:                 distance[$v$][edges+1] = newAverage
18:                 previous[$v$][edges+1] = $u$
19: return previousToPath(previous, $\vartheta_s$, $\vartheta_t$)
```

The function w assigns a weight to each edge and is defined by

$$w(e{=}(\vartheta_{im},\vartheta_{jn})){=}\begin{cases}0 & \text{if } \vartheta_{jn} = t\\ 0 & \text{if } \vartheta_{im} = s\\ 1 - P\left(X_j = x_{jn} \mid X_i = x_{im}\right) & \text{otherwise}\end{cases} \tag{3}$$

With this, the edge weight between a node ϑ_{im} (denoting RV $X_i = x_{im} \in \Omega_{X_i}$) and a node ϑ_{jn} is $1 - P\left(X_j = x_{jn} \mid X_i = x_{im}\right)$. If X_j has further parents besides X_i, the probability $P\left(X_j = x_{jn} \mid X_i = x_{im}\right)$ is calculated by marginalization using (1) which uses (2) to calculate the marginal probability. Edges to t and from s have weight 0.

2.2 Minimum Average Path Search

We assume that most likely paths in the BN are desired and corresponds to potential specifications. Thus, we aim to find all most probable paths in the MG that exhibit a maximum average likelihood p_{min} (i.e. a minimum average edge weight $w_{\text{max}} = 1 - p_{\text{min}}$). Existing k-shortest path search algorithms look for paths with minimum length by considering summation of edge weights. In case of BN this would prefer shorter paths in terms of lower number of edges, thus limiting the length of candidate specifications. Therefore, we aim to maximize the average path likelihood. Existing approaches do not support finding paths of minimum average weights and do not stop based on a metric, but rather find the k shortest paths. Also, sidetrack-based k-shortest path algorithms, e.g. Eppstein's algorithm [8], cannot be used for minimum average path search, thus, we extend Yen's search [22] algorithm to find paths of minimum average edge weight, in the following manner.

First, we stop searching paths if a metric of a path falls below a given threshold. Second, in its kernel, Yen needs to use our modified Dijkstra algorithm (listed in Alg. 1) to find paths with minimum average weights for a given start and end node. This modification is required, as when searching minimum average paths, greedy algorithms, e.g. Dijkstra's algorithm, can result in non-optimal solutions, e.g. in Fig. 3 taking a local optimum would not lead to an optimal solution. For b the path with minimum average path weight is (a, b). But, the globally best path depends on the number of edges taken so far and the edge weights to come, so that the path with minimum average edge weight is (a, x, b, c). We solve this in our algorithm by remembering for each vertex the best path for every number of edges reaching that vertex (initialization in lines 4-6). A further loop is added that not only iterates over all neighbors of a vertex u but also over all best paths

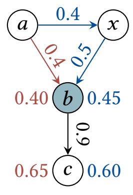


Figure 3: Limitations of Dijkstra Algorithm.

Algorithm 2:
Metric based approach
Input: ***paths P, target range***
Output: ***specifications s***

```
 1: $D$ = editDistances($P$)
 2: $s$ = [ ]
 3: for each path $p_i \in P$ do
 4:     $r$ = $p_i$.toRegex( )
 5:     for each path $q \in P$ sorted by $D_{i-}$ do
 6:         $r$.merge($q$)
 7:         if $r$.isRedundant($s$) then
 8:             break
 9:         if metrics in target range then
10:             $s$.removeRedundantRules($r$)
11:             $s$.add($r$)
12:             break
13:         else if any metric surpasses target range then
14:             break
15: return $s$
```

with a different number of edges reaching u (line 10). The new average path weight is computed in lines 11-15 where edges including vertices s and t have to be handled separately as those nodes are neglected. This results in a candidate set of likely paths in the BN. Those resemble a set of strict temporal properties of the system.

3 METRIC-BASED PATH MERGING

Likely paths in a BN already express temporal specifications of the system, e.g., if we require a path to be present in the trace in its found order. However, those properties are too strict and thus meaningless, as they are rarely satisfied by traces under test. To overcome this, with Alg. 2 we propose successive loosening of properties, such that resulting properties are still strict enough, e.g. not trivial or even a tautology. The basic idea is to find most similar paths (from the set of found paths) and iteratively merge those until looser specifications are obtained, that lie within a defined strictness range (defined by *literal ratio*, *combination count*, *conformity*). This is explained in the following by first, introducing our similarity metric (Levenshtein distance), which is an edit distance commonly used to measure differences in terms of modifications between two sequences, and then describing the algorithm.

Levenshtein matrix and distance [12]: The Levenshtein matrix $M^{\pi_a,\pi_b} \in \mathbb{N}^{m \times n}$ for two paths $\pi_a = (a_1, \ldots, a_m)$ and $\pi_b = (b_1, \ldots, b_n)$ is recursively defined by

$$M^{\pi_a,\pi_b}_{i0} = \textstyle\sum_{k=1}^{i} w_{\text{del}}\,, \qquad M^{\pi_a,\pi_b}_{0j} = \textstyle\sum_{k=1}^{j} w_{\text{ins}}\,,$$

$$M^{\pi_a,\pi_b}_{ij} = \begin{cases} M^{\pi_a,\pi_b}_{i-1j-1} & \text{if } a_i = b_j\,,\\ \min\begin{cases} M^{\pi_a,\pi_b}_{i-1j} + w_{\text{del}}\\ M^{\pi_a,\pi_b}_{ij-1} + w_{\text{ins}}\\ M^{\pi_a,\pi_b}_{i-1j-1} + w_{\text{sub}}\end{cases} & \text{otherwise}\,,\end{cases}$$

where w_{del}, w_{ins} and w_{sub} are weighted costs for deleting, inserting and substituting a symbol. In our approach we set $w_{\text{del}} = w_{\text{ins}} = w_{\text{sub}} = 1$. The Levenshtein distance $L(\pi_a, \pi_b)$ between two paths $\pi_a = (a_1, \ldots, a_m)$ and $\pi_b = (b_1, \ldots, b_n)$ is given by M^{π_a,π_b}_{mn} where M^{π_a,π_b} is the Levenshtein matrix for π_a and π_b.

i) Calculating edit distances between paths: To ensure more similar paths to be merged first, similarity between paths is computed in line 1 of Alg. 2. The order in which paths q are merged with a starting path p is determined by the Levenshtein distance between them. If two paths have the same distance to a starting path, the path with the higher average probability is merged first. All pairwise computed Levenshtein distances are stored in a symmetrical matrix D. For a set of paths $P = \{\pi_1, \ldots, \pi_m\}$ the edit distance matrix $D \in \mathbb{N}^{m \times m}$ is defined by its elements

$$\left(D_{ij}\right)_{0 \le i,j \le m} = L(\pi_i, \pi_j) \tag{4}$$

	literal a	?-quantifier a?	alternation (a | b)
delete	a?	a?	(a | b)?
substitute with c	(a | c)	(a | c)?	(a | b | c)
insert c (at end)	a c?	a? c?	(a | b) c?

Table 1: Edit operation rules for regular expressions.

Also, the minimal number of editing operations (insertion, deletion, substitution) that are required to transform one path into the other is called the *minimum edit distance.*

ii) Merging paths: To be able to merge paths, we represent them as regular expressions (*regex*) such that any path $\pi = (\vartheta_1, \vartheta_2 \ldots, \vartheta_n)$ results in a strict *regex* $\vartheta_1\,\vartheta_2\,\ldots\vartheta_n$ of n single literals. Each literal resembles a RV and its value (e.g. $a_0 = 1$). However, for better readability we use one symbol for such pairs. Path translation to a *regex* happens in line 4. As we deal with finite paths, a reduced syntax is used, which contains the ?-quantifier and the alternative notation |, where e.g. (a | b)c, matches either a path *ac* or *bc*. Two *regex*s can be merged by the edit operations *delete*, *substitute* and *insert* to give new *regex*s matching at least both input expressions (line 6). Table 1 lists all rules for edit operations on each of the three *regex* symbol types, e.g. when substituting a literal a with c it results in a looser expression $(a \mid c)$. The order of edit operations is derived using the Levenshtein matrix of the two expressions. For this the order in which to execute the edit operations is precomputed using a "backtracking" algorithm, which searches a non-increasing path within the Levenshtein matrix M from M_{mn} to M_{00}. This is performed during merging in line 6. Thus, the edit operation list for two paths π_1 and π_2 is computed as follows. Given the algorithm is currently at M_{ij}, for a step to the left an insertion of symbol π_{2_j} at position i is added, for a step upwards a deletion at position i is is added and for a diagonal step to the upper left, where the value at the upper left is lower, a substitution with symbol π_{2_j} at position i is added to the list. The operation positions refer to positions in the path π_1. For a diagonal step to the upper left, where the value at the upper left is not lower, no entry is added to the operation list, as then symbols are equal. After computing the edit operations, the two expressions are merged by executing the corresponding merge operations defined in Table 1, e.g., the paths $\pi_1 = (a, b, c, d)$ and $\pi_2 = (a, c, D)$ are transformed into the *regex*s *abcd* and *acD* respectively. Merging them results in the new *regex* $a\,b?\,c\,(d \mid D)$ by applying the rules for delete and substitute from Table 1. Table 2 shows the Levenshtein matrix for those expressions with the highlighted backtracking path.

		a	c	D
	0	1	2	3
a	1	0	1	2
b	2	1	1	2
c	3	2	1	2
d	4	3	2	2

Table 2: Backtracking

iii) Metrics: The stop criterion for merging paths is defined by the current strictness of the specification measured as *literal ratio* and *combination count.* For this a target range is defined for both criteria. The *literal ratio* is the ratio between the number of single literal characters and the total number of symbols in the expression. The *combination count* is the total number of paths in the BN that match the current specification, e.g., the *regex* $a\,b?\,c\,(d \mid D)$ has a literal ratio of 0.5 and matches *abcd*, *abcD*, *acd* and *acD*, giving a combination count of 4. If both metrics fall into the acceptance window, merging stops and the *regex* is added to the result list (line 9). If one metric overshoots its acceptance the *regex* is dismissed (line 13). From the resulting specifications, symbols with a ?-quantifier are removed from the front and end of each *regex* as they do not add relevant information to the property

iv) Conversion to LTL: All found *regex*s are converted into a LTL *formula* of events, premises and conclusions in the shape below, such that it becomes applicable for model checking on traces. This formula specifies that if the premise is found in the trace, the conclusion must follow.

$$\begin{aligned} \textit{formula} &:= \mathrm{G}(\textit{premise}) \\ \textit{premise} &:= \textit{event} \longrightarrow \mathrm{XG}(\textit{premise}) \mid \textit{event} \longrightarrow \mathrm{X}(\textit{conclusion}) \\ &\quad\mid \textit{event} \longrightarrow \mathrm{XF}(\textit{conclusion}) \\ \textit{conclusion} &:= \textit{event} \mid \textit{event} \land \mathrm{X}(\textit{conclusion}) \\ &\quad\mid \textit{event} \land \mathrm{XF}(\textit{conclusion}) \mid \textit{event}\,\mathrm{U}\,(\textit{conclusion}) \end{aligned}$$

Event: An event represents a symbol of the *regex.* For a literal the event is just the literal itself, for an alternation the event is the disjunction of the alternation's literals, e.g. $(a \mid b \mid c)$ results in the event $(a \lor b \lor c)$. Thus, literals and alternations are converted to those symbol types. *Premise:* To ensure that the expression matches only in locations on the trace where the exact behavior is present, a premise is used. The premise comprises a prefix of symbols from the *regex* that is unique within all combinations. Only if this premise is given the LTL check is executed. *Conclusion:* The conclusion comprises all symbols of the *regex* that are not part of the premise.

Statistical regex conversion: To be able to convert paths into a LTL formula of appropriate strictness, the right symbol between two path elements is required. To be able to assess the required strictness, during learning of the BN a histogram of numbers of events that occurred between two nodes is counted and stored per edge. *Subsequence:* Subsequent events in a regex (e.g. *ab*) can either result in a "next" X, implying b to appear immediately next, or in a "finally" XF operator, which implies b to appear eventually in the future.[1] In this case, the further conversion is used if the number of intermediate traces makes up a higher percentage than a threshold (we say it has *intermediate noise*) and else the latter conversion is applied.

?-Quantifier: As symbols with ?-quantifier (e.g. $a?b$) are optional, those are translated into the "until" U operator (e.g. $a\,U\,b$). Similar to the case above, this requires all as until a b occurs to be directly subsequent and thus, the number of intermediate events between all as until a b occurs to be zero. If this is not satisfied, the ?-quantified symbol is omitted (e.g. $a?b$ results in b).

Multiple ?-Quantifiers: Multiple consecutive ?-quantified symbols, e.g. $a?\,b?\,c?\,d$, are translated recursively in a similar manner. Starting from the last symbol d we recursively check whether the preceding ?-quantified symbol has *intermediate noise.* If no noise is present, the corresponding symbol is translated to the "until" operator U. Else, all ?-quantified symbols left to the current symbol are dropped, e.g., given $(a?, b?)$ as the only pair with *intermediate noise* in $a?\,b?\,c?\,d$ results in the LTL formula $b\;U\;c\;U\;d$.

v). Removal of redundant specifications: Single merging can result in meaningless specifications. Thus, after each merging step we measure the expressiveness of the found specification. As soon as the current specification is the result of merging paths that already made up another found specification, the current iteration is terminated and the specification discarded (lines 7 and 10).

4 COMPARISON-BASED PATH MERGING

Using only the stated metrics as a measure of appropriate strictness is a good solution when only one BN is available. However, when only metrics are included as a stop criterion, the method might keep merging longer than required. Also, behavior between systems or its instances might differ (e.g. due to system modifications or environmental conditions). This results in structural variations of BNs that are learned from individual instances of the system. Thus, including multiple such BNs for specification mining can be exploited to find traces of appropriate strictness by using a second BN, called validation model, to decide when to stop merging paths.

Both BNs differ in structure, but as both BNs resemble the same system behavior the same specifications should likely be present. Thus, in our extended approach, which is presented here, merging paths filters out this structural noise such that a specification with validity in both BNs is found. For this, during merging, each path is checked

[1] XF is used to allow two equal successive events which is not possible by using F only.

on the validation model and accepted if a minimum average likelihood is achieved on it. This additional stop criterion is explained in this section. The resulting algorithm is identical to Alg. 2, except that after line 2 $K_{\text{val}} = A_{\text{val}}$.toKripke() is inserted and lines 5 to 14 are replaced by the following code, with validation BN A_{val} as additional parameter.

```
1:  for each path $q \in P$ sorted by $D_{i-}$ do
2:      $r$.merge($q$)
3:      if $r$.isRedundant($s$) then
4:          break
5:      if any metric surpasses threshold then
6:          break
7:      if ModelCheck($r$, $K_{\text{val}}$) then
8:          $s$.removeRedundantRules($r$)
9:          $s$.add($r$)
10:         break
```

Comparison-based stop: Checking whether a *regex* that represents merged paths exists in the validation model is done using model checking tools. The found specification is only accepted if at least one path in the validation model matches the *regex* formula and exhibits the same minimum likelihood. Lastly, besides structural checks a metric based check is performed as well. If the literal ratio falls below or the combination count is above a threshold, merging is stopped to avoid specifications that are too soft.

5 SYNTHETIC EVALUATION

In this section *BaySpec* is evaluated on synthetic data[2].

Dataset: Synthetic data is generated by sampling from multiple BNs, with the structure defined in Sec. 2 and by time shifting each sampled sequence. Each BN represents one functional process of the system and RVs per BN are unique across all BNs. BNs can be modified in terms of probability distributions, as well as numbers of nodes, edges and states. Per BN a validation BN is generated by removing cross-edges, i.e. edges between nodes of different dimensions, from the original BN. To simulate dominant behavior of functions, the CPD of a randomly picked state per RV is set between 0.6 and 1.0. Generated BNs exhibit 4 to 5 dimensions, 4 to 5 nodes per dim., 2 states and $0.8 \cdot \#$nodes cross-edges to 3 other dims. per dim. The minimum literal ratio and maximum combination count is set to 0.5 and $2.5^{n/2}$ respectively, where n is the *regex*' number of symbols.

Metrics: *Complexity* of learned specifications is measured by height and unique event count of the resulting LTL formula's syntax tree. The height is the number of nodes between the root and the deepest leaf. The unique event count is the number of leaves with different events. The frequency of specifications of certain complexity is measured. More frequent specifications of higher complexity are desired, as this corresponds to better expressiveness. The *False-Positive (FP) Rate* is the percentage of specifications that are falsely learned. In our scenario this corresponds to specifications that contain events from more than one functional process (i.e. from RVs of different BNs). Lastly, the *runtime* of the approaches is measured.

Setup: Experiments are conducted on a Lenovo™ T480s equipped with two Intel® Core® i5-8350U 1.70GHz CPUs with 16 GB of RAM. Further, BaySpec is run without refinement of the histogram.

Evaluation of proposed Mining: First, we run the comparison based approach on synthetic data produced by one BN and its validation BN for 50 scenarios. Averaging produces the results shown in Fig. 4. On the *left*, we vary the validation BN's structure by randomly removing 0 to 75 % of cross-edges from the original BN for several minimum average weight thresholds $p_{\min}$. More removed cross-edges result in less found specifications, as more paths need to be merged which softens the property to be found in the validation BN. This shows that stricter specifications are found from more similar BNs. Further, higher $p_{\min}$ require more expressive specifications to be mined only, which can be seen as curves with higher $p_{\min}$ produce less specifications. Thus, $p_{\min}$ allows to parameterize the level of strictness in our approach. Second, both approaches (metric and comparison-based) are compared by testing 2 model sizes with (dimension, nodes per dimension) of (4,4) and (5,5), each sampled 50 times randomly. As Fig. 4 (*mid*) shows, the metric based approach finds more specifications as it is only restricted by $p_{\min}$ and the target range for *literal ratio* and *combination count* which are set to [0.5, 0.8] and $[2^{n/5}, 2^{n/2}]$ respectively, where n is the *regex*' number of symbols. The comparison based approach is additionally restricted by the BN structure, i.e. adding a validation BN with 20% removed cross-edges increases strictness and thus, meaningful specifications. Further, for both approaches too high $p_{\min}$ result in nearly no specification, while low $p_{\min}$ produce potentially many and thus, meaningless properties. Third, runtime of the comparison based approach (together with Mining Graph extraction and path computation) is measured by averaging over 50 models of above structure (Fig. 4 (*right*)), when $p_{\min}$ is increased at various removed cross-edge ratios ξ. Results are normalized with the average BN size. Lower $p_{\min}$ result in higher runtimes as more paths are found and thus, more merging and model checking operations are performed. Also, with less than 4 seconds our approach shows reasonable runtimes.

Evaluation against existing approaches: Given our discussion in Sec. 1, Perracotta [21] and Synoptic [3] were chosen for comparison, as both are among the most prominent state-of-the-art approaches. We extended Synoptic such that it is able to find specifications with a confidence below 1 (i.e. to handle imperfect traces). To compare complexity, we randomly sample from BNs to get trace sets of diverse complexities, i.e. we vary the number of traces per trace set and the number of sampling repetitions within one trace, both between 2 and 5. These trace sets are used as inputs for the tools. In the validation BNs, 20 % of cross-edges are removed and $p_{\min}$ is set to 0.85. Resulting complexities are illustrated in Fig. 5 (*left* and *mid*), with bigger circles indicating more frequent occurrence of complexities. It can be seen that first, *BaySpec*'s extraction produces more variable arbitrary length patterns of higher complexity, which results from capturing longer sequences of event correlations. In a second experiment, shown in Fig. 5 (*right*), we randomly select an increasing number of BNs from a pool of 50 BNs to produce trace sets of multiple functions (each BN as one function). All miners were run on this trace set, where we assume that *BaySpec* learned the ground truth models. The FP ratio is measured for the compared approaches. In the validation BNs, 30% of cross-edges are removed and $p_{\min}$ is set to 0.8. In *BaySpec*, mined specifications might match paths that were not used during merging. Thus, we consider properties with a combination count bigger than the number of merged paths as FPs. Other approaches tend to mix functional behaviors and, thus, produce high numbers of FPs, while prior segmentation in our approach results in specifications with less FPs.

6 CASE STUDY: AUTOMOTIVE SOFTWARE

Background and Goal: In vehicles Electronic Control Units (ECUs) exchange sensor signals to run its software. To ensure the correct functional behavior of the software, traces are recorded from invehicle networks. Each signal represents a certain event type, i.e. dimension (e.g. driving state, speed, indicator state) [16]. Processes of those signals and causality between events of those signals can be modeled as BNs of the shape described in Sec. 2. Structure is learned using a constraint-based discovery approach and parameters are learned using Maximum Likelihood estimation. With this, we model the indicator behavior. The indicator light is switched on by a driver using the handle bar, when turning left or right. This function consists of a handle that *(de-)activates* the indicator either in *steady mode* or for *3 seconds*. This influences the state of the indicator (e.g. *left indicator on*). Depending on this, in steady mode synchronization is *started* and a *new indication cycle* begins. Once the handle changes back to its *default state* from the *3 second* state, depending on this state synchronization is *stopped*. If it is in *steady mode*, it is only *stopped* with

[2]Python Implementation: https://github.com/arturmrowca/bayspec

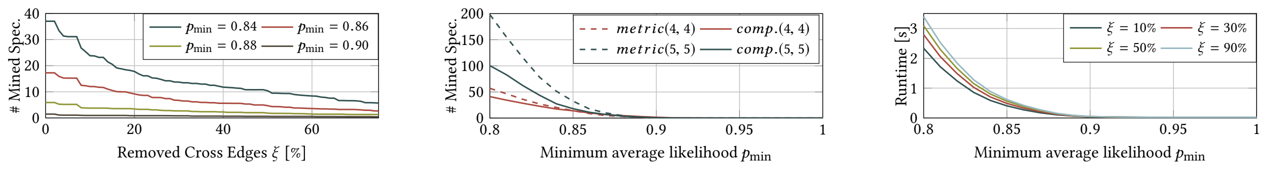


Figure 4: *Left:* Number of mined specifications under various percentages of removed cross-edges ξ between original and validation BN, under various minimum average likelihoods $p_{\min}$. *Mid:* Comparison of metric based and comparison based approach in terms of # Mined Specifications for various minimum average likelihoods $p_{\min}$ for 2 different BN sizes. *Right:* Runtime of the approach under various average likelihoods $p_{\min}$ for 4 cross-edge deletion percentages ξ.

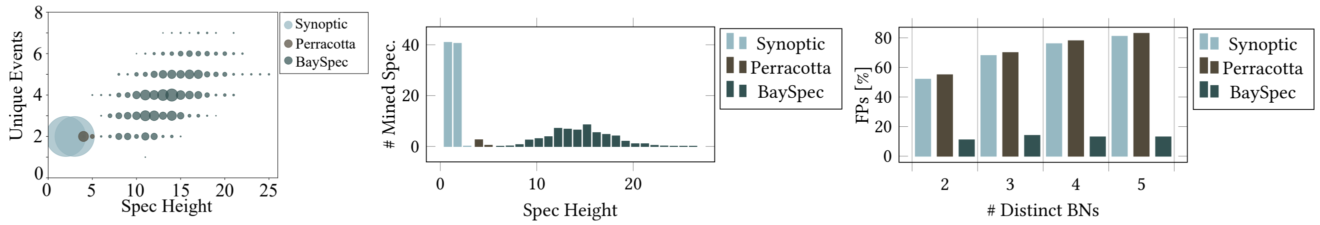
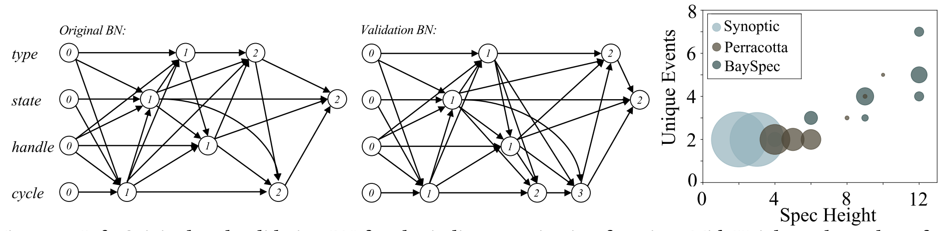


Figure 5: *Left:* Height and number of unique events of found specifications for three approaches, with circle size being frequency of occurrence. *Mid:* Height of Specifications against its frequency. *Right:* Ratio of FPs mined by three approaches with traces of increasing number of functions.



Spec. 1: Avg. likelihood = 0.910:
G(handle=tip down
→ XF(state=left indicator on
∧XF state=both indicators off))

Spec. 2: Avg. likelihood = 0.831:
G(handle=tip up
→ XF(handle=no action
∧XF state=both indicators off))

Figure 6: *Left:* Original and validation BN for the indicator activation function. *Mid:* Height and number of unique events of found specifications for three approaches, with circle size being frequency of occurrence. *Right:* Minimal examples of found specifications.

the handlebar returning to *default* by the user. This network that is learned from 223 process segments is shown in Fig. 6 (left).

Specification Extraction: We divide the data into 2 subsets to learn an original and a validation BN as proposed in Sec. 4 and apply our approach to extract specifications with $p_{\min} = 0.8$. The resulting complexity of extracted specifications is shown in Fig. 6. The plot shows that the resulting specifications are still expressive and of various length as circles are on the top right, while existing approaches result in lower complexity specifications. Further, we validate the results using expert knowledge. Two minimal examples of found specifications are shown in the table in Fig. 6. Those results show that, even in a noisy environment of real world traces, specifications of good quality can be found with the proposed model-based approach.

7 CONCLUSION

We presented the *BaySpec* approach for dynamic specification mining. Further, we proposed two approaches to find LTL formulas of appropriate strictness and compared those to existing approaches. We showed that *BaySpec* finds specifications of arbitrary length and higher complexity, while producing less false positives.

ACKNOWLEDGMENT

This work was supported by the BMW Group. With the support of the Technische Universität München - Institute for Advanced Study, funded by the German Excellence Initiative and the European Union Seventh Framework Programme under grant agreement n° 291763.

REFERENCES

[1] R. Alur, P. Černý, P. Madhusudan, and W. Nam. 2005. Synthesis of interface specifications for Java classes. *ACM SIGPLAN Notices* 40, 1 (2005).

[2] G. Ammons, R. Bodík, and J.R. Larus. 2002. Mining specifications. *ACM Sigplan Notices* 37, 1 (2002).

[3] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M.D. Ernst. 2011. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proc. of the ESEC/FSE '11*.

[4] M. Bonato, G. Di Guglielmo, M. Fujita, F. Fummi, and G. Pravadelli. 2012. Dynamic property mining for embedded software. In *ACM Proc. of CODES+ISSS 2012*.

[5] G. Cutulenco, Y. Joshi, A. Narayan, and S. Fischmeister. 2016. Mining timed regular expressions from system traces. In *Proc. of the ASE '16*.

[6] A. Danese, T. Ghasempouri, and G. Pravadelli. 2015. Automatic extraction of assertions from execution traces of behavioural models. In *IEEE Proc. of DATE 2015*.

[7] D. Engler, D.Y. Chen, S. Hallem, A. Chou, and B. Chelf. 2001. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *ACM SIGOPS Operating Systems Review*, Vol. 35.

[8] D. Eppstein. 1998. Finding the k shortest paths. *SIAM Journal on computing* 28, 2 (1998).

[9] M.D. Ernst, J. Cockrell, W.G. Griswold, and D. Notkin. 2001. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. on Software Engineering* 27, 2 (2001).

[10] M. Gabel and Z. Su. 2008. Javert: fully automatic mining of general temporal properties from dynamic traces. In *Proc. of SIGSOFT '08*.

[11] C. Lemieux, D. Park, and I. Beschastnikh. 2015. General LTL specification mining (t). In *Proc. of ASE 2015*.

[12] V.I. Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*.

[13] W. Li, A. Forin, and S.A. Seshia. 2010. Scalable specification mining for verification and diagnosis. In *Proceedings of DAC 2010*.

[14] D. Lo and S.C. Khoo. 2006. SMArTIC: towards building an accurate, robust and scalable specification miner. In *SIGSOFT '06*.

[15] D. Lo, S.C. Khoo, and C. Liu. 2008. Mining temporal rules for software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice* 20, 4 (2008).

[16] A. Mrowca, T. Pramsohler, S. Steinhorst, and U. Baumgarten. 2018. Automated interpretation and reduction of in-vehicle network traces at a large scale. In *Proceedings of DAC 2018*. IEEE.

[17] M.K. Ramanathan, A. Grama, and S. Jagannathan. 2007. Static specification inference using predicate mining. In *ACM SIGPLAN Notices*, Vol. 42.

[18] S. Shoham, E. Yahav, S.J. Fink, and M. Pistoia. 2008. Static specification mining using automata-based abstractions. *IEEE Trans. on Software Engineering* 34, 5 (2008).

[19] S. Vasudevan, D. Sheridan, S. Patel, D. Tcheng, B. Tuohy, and D. Johnson. 2010. Goldmine: Automatic assertion generation using data mining and static analysis. In *IEEE Proc. of DATE 2010*.

[20] A. Viterbi. 1967. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Trans. on Information Theory* 13, 2 (1967).

[21] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. 2006. Perracotta: mining temporal API rules from imperfect traces. In *Proc. of ICSE 2006*.

[22] J.Y. Yen. 1971. Finding the k shortest loopless paths in a network. *management Science* 17, 11 (1971).