Related Work

[3]  
kTail algorithm: finite state model => more compact one  
Approaches that leverage kTail to infer models without developer supervision

[30]  
There have been numerous work in the research of automoaton-based specification mining [ … ]  
A et al. have …. [x]  
B et al. provided… [x]

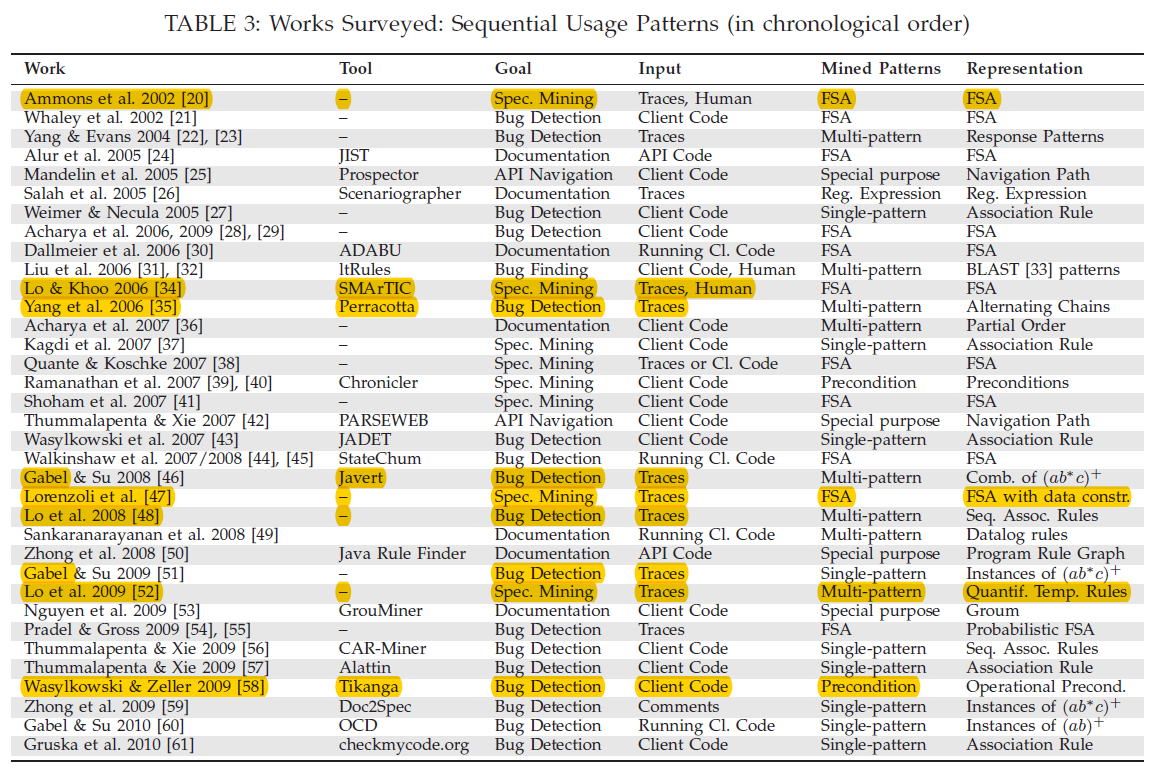
[19] SMArTIC  
There have been numerous work in the area of specification mining. They can be classified into two groups, depending on how the mined specifications are represented: **automaton-based** [ … ] and **non-automaton based** [ … ] specification mining.

[Periodic Task Mining in Embedded Systems]  
formalisms including **state machines** [3] [39], **Petri nets** [42], various types of **invariants** [21] [19] and **UML models** [29] [2]. Some formalisms support timing, such as mining **LTL expressions** [33], CTL expressions [6], or hybrid system automata [37].

[Automatic Steering of Behavioral Model Inference]  
Techniques that extract and generate behavioral models can be classified into **automaton** (for example, [3, 7, 17, 23, 31, 34]) and **non-automaton based** (for example, [9, 8, 20, 35]) techniques, depending on the nature of the models that they generate. Techniques of both kinds generate models that can **support test case generation** [13, 24], **debugging** [30, 25] and **verification** [26]. In this paper, we focus on automaton based techniques, namely techniques that generate finite state automata (FSA) from execution traces

[20]  
There have been both static and dynamic approaches proposed for property mining. Static property mining is found to be effective and accurate but faces challenge in scaling with the program size [2, 12]. On the other hand, dynamic property mining approaches guarantee scalability but lack in the quality of mined properties which depend on various factors such as the observed executions and the test suite used for stimulating the design [8].  
There are two main subsets of dynamic property mining approaches: invariant miners [13, 14, 6, 17] and temporal property miners [4, 24, 11, 27, 25]. Of all the available tools, Daikon [14] has proven to be most successful in inferring the most probable invariants. It has been used widely for debugging, testing, documentation and maintainability

[Bonato (cites [10])]  
Static property mining is effective and accurate, but it does not scale sufficiently well with program size. On the contrary, dynamic property mining guarantees a better scalability, but the quality of mined properties depends on the observed executions and then on the test suite used to stimulate the design.



**Yang et al. (*Perracotta, “Mining Temporal API Rules from Imperfect Traces”*)**

**mines two-event temporal rules** from execution traces. To infer these rules, Perracotta uses a set of predened rule templates and partitions input traces to several sub-traces. It computes satisfaction rate of a template, which is the number of partitions satisfying the template divided by the number of total partitions

[Gecco] A simple heuristic to explore the space of possible patterns is to search for specific pattern templates. In this work, simple two-events patterns are mined from the execution traces. These patterns follow the template event1 always followed by event2. Such simple patterns, once mined, can be combined to produce larger patterns.

[Rule-based specification mining leveraging learning to rank] that **mines two-event temporal rules** from execution traces. To infer these rules, Perracotta uses a set of predefined rule templates and partitions input traces to several sub-traces. It computes satisfaction rate of a template, which is the number of partitions satisfying the template divided by the number of total partitions

[10] **mines several two variable response patterns** and chains alternating properties together to form multi-variable properties. It also supports approximate inference to filter out uninteresting properties

[11] adopted a similar approach for locating alternating events, Perracotta, that introduced novel methods for handling imperfect traces. Sources of imperfection include interleaved concurrent executions, omitted information (like memory addresses), or bugs. In this work, the authors briefly describe a heuristic for combining simple alternating patterns, but the approach is limited to finding simple sequencing patterns

[14] present an interesting work on **mining two-event temporal logic rules** (i.e., of the form G(a→XF(b)), where G, X and F are LTL operators), which are statistically significant with respect to a user-defined ‘satisfaction rate’. The algorithm presented, however, does not scale to mine multi-event rules of arbitrary length. To handle longer rules, Yang et al. suggest a partial solution based on the concatenation of mined two-event rules. Yet, the method proposed might

[19] present their work in mining a restricted form of response pattern [10] called an “alternating” pattern using a set of templates. They only **consider rules involving two events**; ie., of the form a ! b. In order to handle longer rules, Yang et al. introduce “**chaining**”. For example, if both A ! B and B ! C are significant, they can be concatenated to form A ! B ! C which will be significant too. However, the reverse may not be always true: A ! B ! C might be significant although only rule A ! B is significant while B ! C is not. For such cases, the rule A ! B ! C cannot be generated by inferring from twoevent rules and chaining them. Hence, chaining only gives a partial solution rule set for multi-event (> 2) sequential patterns.

[28] A simple heuristic to explore the space of possible patterns is to search for specific pattern templates. In this work, **simple two-events patterns are mined** from the execution traces. These patterns follow the template event1 always followed by event2. Such simple patterns, once mined, can be combined to produce larger patterns. As stated by Lo et al., such combination methods might miss some multi-event patterns or create ones that are not meaningful.

[Bonato] is a mining tool developed to detect **likely binary patterns from an execution trace**. The intuition behind these works is that frequently occurring behaviors that match temporal patterns are likely to be true. The main idea to achieve scalability to large traces is to mine binary specification first, in form of automata, and then post-process them with inference rulesto form more complex state machines. The kind of pattern mined by Perracotta is (P G S), where P and S are placeholders for generic events and G a placeholder for a generic binary pattern.

[ARTINALI] derive temporal logic propositions, and capture sequences of events by tracking dynamic traces

**Lo D. et al. (*SMArTIC-2006, SpecForge, “Mining […] SW Maintenance”-2008)***

[Gecco] as stated by Lo et al., such combination methods might miss some multi-event patterns or create ones that are not meaningful. Alternatively, Lo et al. generalize the two-event template to a multi-event template, and mine patterns in the form of whenever a series of events occurs, eventually another series of events will occur.

[Rule-based specification mining leveraging learning to rank]   
2008 extend Yang et al.’s approach by inferring from execution traces **temporal rules with arbitrary lengths instead of two-event rules**;  
SMArTic that applies a variant of k-tails automaton learning algorithm to **infer finite state automatons** (FSAs) from a set of execution traces;  
SpecForge that synergizes different FSA-based specification miners by introducing novel concepts of model fission and model fusion

[25] SMArTIC: that **infers a finite state automaton** from a set of execution traces [6]. This approach is built on a variant of **k-tails** automaton learning method that infers a probabilistic FSA and employs trace filtering and clustering. Erroneous traces are removed from the input execution traces and rather than learning a model from all the traces, the **traces are clustered** into groups and a separate FSA is learned from each group. These FSAs are later combined together into one FSA by identifying equivalent transitions – the goal is to get a larger FSA that accepts all the sentences accepted by the smaller FSAs.

[28] **generalize the two-event template to a multi-event template**, and mine patterns in the form of whenever a **series of events occurs**, eventually **another series of events will occur**.

**Lemieux et al. (*Texada, “General LTL Specification Mining”*)**

[Gecco] mentioned approaches cannot mine complex patterns such as When event1 occurs, either event2 occurs just after and event3 will never occur after, or event2 will not occur just after and event3 will eventually occur. Lemieux et al. [1] proposed a model-checking-base tool, Texada, to mine such **complex patterns**. However, the pattern templates used for the search have to be **specified beforehand**. In other words, it is impossible to find what you are not explicitly looking for. This precludes exploratory searching, which can reveal previously unanticipated patterns

[Rule-based specification mining leveraging learning to rank] introduce Texada that mines temporal specifications in the form of linear temporal logic (**LTL**) of arbitrary **length and complexity**

[28] proposed a model-checking-base tool, Texada, to mine such complex patterns. However, the pattern templates used for the search have to be specified beforehand. In other words, it is impossible to find what you are not explicitly looking for.

[ARTINALI] derive temporal logic propositions, and capture sequences of events by tracking dynamic traces

**Ernst et al. (*Daikon, “Dynamically Discovering Likely Program Invariants”*)**

[4] Daikon is a tool that automatically infers **likely program invariants** using statistical inference from a program's execution traces.

[6] proposed automatic deduction of formal specifications. Their Daikon tool works by learning **likely invariants** involving program variables from dynamic traces.

[11] is a dynamic technique that is similar in spirit to our own analysis. Daikon locates **invariants** on the values of variables, while we locate invariants on the sequencing of function invocations

[14] propose an interesting work called Daikon that discovers **value-based program invariants** occurring at a certain program point by analyzing program execution traces. These value-based invariants are usually in the form of **algebraic equations** or **boolean expressions** (e.g., X>Y, Z<Y , etc.). Different from Daikon, in this work we focus on mining temporal properties, capturing ordering among events.

[Mining assumptions for synthesis] is one of the earliest template-based specification mining tools that generates **single-state invariants** or **pre-/post-conditions** in programs.

[Bonato] to be the more effective tool to infer **likely invariants**, i.e., properties that are true in specific program points over all the observed executions.

[ARTINALI] was the first dynamic analysis-based technique to derive **(likely) invariants about data value relations** [14], and falls into the first class of techniques. Daikon can be placed on the data axis as it produces a model for data constraints without taking into account the events or timing of the system

**Gabel and Su (*Javert, “Fully Automatic Mining of General Temporal Properties from Dynamic Traces”*)**

[Gecco] combination (“chaining”) strategy was also proposed by Gabel and Su

[10] also mines alternating patterns, along with resource ownership patterns (i.e., (ab\*c)\*) and composes them into more complex properties

**Dwyer (“Property Specification Patterns for Finite-State Verification”)**

**Ammons (*“Mining Specifications”*) ‘The Origin’**

[11] first characterized the inference of temporal specifications as a language learning problem. In this work, the authors used a probabilistic finite automaton learner to extract likely specifications. A key challenge with their approach was simplifying specifications to an acceptable level of precision

[12] develop a specification miner, Strauss, that mines specification by learning a probabilistic finite state automaton. Unlike our approach, Strauss requires the input alphabet of the automaton to be specified, but it does have the potential to find more complex specifications.

[19] There, a machine-learning approach is employed to discover program specifications by analyzing program execution traces. Under the assumption that the program being mined must “reveal strong hints of correct protocols” during its execution, Ammons et al. demonstrate that correct specifications can be obtained by their technique. Specifically, their technique focuses on mining of specifications which reflect temporal and data dependency relations of a program through traces of its API-client interaction.

**Li (“*Scalable Spec Mining for Verification and Diagnosis” – 1st Lady w/ hammer”*)**

extend Yang et al.'s work by extracting simple linear temporal logic (LTL) rules from execution traces for hardware design

[Rule-based specification mining leveraging learning to rank] also extend Yang et al.’s work by extracting simple linear temporal logic (LTL) rules from execution traces for hardware design

[10] extend Perracotta to mine simple LTL patterns from traces and merge these to analyze digital circuits. These same LTL patterns are mined between data invariants in [4], [8].

[Mining assumptions for synthesis] proposed a scalable technique for mining temporal specifications from traces produced by digital systems, and showed that the mined specifications are effective in localizing bugs in designs

**Beschastnikh et al. (Synoptic, “Leveraging Existing Instrumentation to Automatically Infer Invariant-Constrained Models”)**

[Rule-based specification mining leveraging learning to rank] automatically mines three types of temporal rules from execution traces and uses them to generate a concise finite state automaton (FSA) model that satisfies these rules

[17] which infers behavioral models that obey temporal properties without resource constraints. As Figure 1 illustrates, Synoptic models are likely more concise but less precise than Perfume models.

**“Gecco” Saied et al. (“Towards the Automated Recovery of Complex Temporal API-Usage Patterns”)**

“More recently”

Genetic algorithm to …

**Reger et al. (Automata-based Pattern Mining from Imperfect Traces) -> Levenshtein**

[10] Reger et al. use a similar patterncomposition technique [24] and extend it to accommodate imperfect traces **[self]**

**GKTail**

[11] Combining the ideas of invariant detection and temporal property mining, Lorenzoli et al. have developed a dynamic analysis algorithm for extracting software behavioral models [20]. The algorithm, GK-tail, builds an Extended Finite State Machine from a set of dynamic traces. The transitions in these extended models include both a called function or method and a set of constraints on the parameters or environment.

[25] to mine extended FSAs that incorporate data flow information. gkTail is able to infer algebraic constraints which specify restrictions on the values of some variables/arguments in the transitions of the FSAs.

**Uddin (“Temporal Analysis of API Usage Concepts”)**

These projects attempt to ease 90 the task of a programmer by synthesizing API usage examples, evolution of API usage, and extracting knowledge from API documentation. Again, most of these techniques work on APIs on a single platform.

**About Static**

[11] Various static analyses [3,16,30] take as input a type and produce as output an automaton that encodes legal call sequences of operations on that type. Call sequences are considered legal if they do not lead to an assertion failure or another exceptional control path.

**Engler et al.**

* Static Inference / Specification Mining
* [11] first introduced the idea of matching an alternating pattern over a program to produce possible specification candidates. This approach suffered from imprecision, so the authors used statistical methods to rank the possible properties

**Ramanathan**

[11] static analysis [24] for detecting “function precedence protocols.” These specifications are of the form “function x is called on all paths leading to an invocation of function y.” The authors later generalized this technique to include other predicates like constraints on variables [25]. These specifications are of limited expressiveness; they correspond to the simple pattern (b+a).

**Wasylkowski and Zeller (Tikanga, “Mining Temporal Specifications from Object Usage)**

Program Code => CTL

Templates: AF p, AG(p => AF q), AG(p => EF q)

Static vs. Dynamic

* Static
  + Wasylkowski and Zeller (Tikanga, Mining Temporal Specifications from Object Usage)
  + Ramanathan et al. (Path-sensitive inference of function precedence protocols)
  + Ramanathan et al. (Static specification inference using predicate mining)
  + Shoham et al. (Static specification mining using automata-based abstraction)
  + Engler et al. (Bugs as deviant behavior: A general approach to inferring errors in systems code)
  + Alur et al. (Synthesis of Interface Specifications for Java Classes)
  + Ernst et al. (Static and dynamic analysis: Synergy and duality)
  + Advantages:
    - Bla
  + By Li:
    - Many techniques have been recently proposed to automatically reverse engineer specifications from a program
* Dynamic
  + Yang et al. (Perracotta, “Mining Temporal API Rules from Imperfect Traces”)
  + Lo D. et al. (SMArTIC-2006)
  + Lo D. et al. (“Mining […] SW Maintenance”)
  + Lemieux et al. (Texada, “General LTL Specification Mining”)
  + Ernst et al. (Daikon, “Dynamically Discovering Likely Program Invariants”)
  + Gabel and Su (Javert, “Fully Automatic Mining of General Temporal Properties from Dynamic Traces”)
  + Ammons (“Mining Specifications”) ‘The Origin’
  + Li (“Scalable Spec Mining for Verification and Diagnosis” – 1st Lady w/ hammer”)
  + Beschastnikh et al. (Synoptic, “Leveraging Existing Instrumentation to Automatically Infer Invariant-Constrained Models”)
  + Reger et al. (Automata-based Pattern Mining from Imperfect Traces) -> Levenshtein
  + Iegorov et al. (PeTaMi, Periodic Task Mining in Embedded System Traces)
  + Bonato et al. (Dynamic Property Mining for Embedded Software)
  + Advantages:
    - Run-time information available (user inputs, object instantiation, …)
    - Code often not available (only binary code)
    - Doesn’t deal with Infeasible paths
    - Portable across various programming languages
  + Disadvantages
    - Quality of analysis depends on quality of input trace set
    - Not feasible to produce traces for all possible paths in a program
  + By Li:
    - Many techniques seek to learn specifications dynamically from an execution trace

Model-based vs. Non-model-based

* Model-based
  + FSM/FA
    - Ammons (Mining Specifications)
    - Lo (SMArTIC) -> improve Ammons
    - Li (Scalable Spec Mining)
    - Beschastnikh (Synoptic)
  + Pattern / unclear
    - Yang (Perracotta) as QRE
    - Gabel and Su (Javert)
    - Reger (Levenshtein) Pattern = Automaton
* Non-model-based
  + Invariants
    - Ernst (Daikon) data invariants, algebraic equations
  + LTL
    - Lemieux (Texada)
    - Lo (SW Maintenance)
    - Bonato (Dynamic Property Mining for Embedded Software)
  + CTL

Some vs. Other

* …

Bonato et al (Dynamic Property Mining for Embedded Software)

