**Spec Mining**

From:

* Logs (=dynamic) from program execution, logging of communication (of running system)
  + Positive & negative set [21]
* Program Code (=static)
* Method Call Sequences => Infer Protocols [13]
* Interface Interaction [6]

Purpose:

* Dependency detection
* Anomaly/Fault detection [1,2]
* Performance debugging
* Input for SW Verification => Bugs

Output:

* Invariants/ Specifications [3,4,10,11,14,21, …]
* Model/ Automaton [3,6,17,19] (FSM => “Grammar Inference Problem”)
* Live Sequence Charts [15,16]
* “specification mining algorithms extract specifications in the form of an automaton or two-event rules“ [14]

Tools (with names):

* Texada (Traces-> Spec) [10]
* Daikon (Traces -> Data Invariants)
* Quarey = Daikon + Texada
* Synopsis (Traces -> Invariants -> Model) [3]
* Perracotta (Traces -> Spec) [4]
* SAM (Scalable Assertion Miner) [5]
* Javert [11] uses [12]
* Perfume (Traces -> Model) [17]
* Smartic [19]
* SpecForge [25]
* ARTINALI [26]
* PeTaMi [27]
* OCD [31]

Problems with Programs, Testing, Specifications:

* Programs are difficult to debug and understand [3]
* Developers rarely write down specifications => Programs lack formal specifications [10]
* Improper management of software evolution, compounded by imprecise, and changing requirements, along with the “short time to market” requirement => lack of up2date specifications [19]
* Not kept up2date
* properties must be specified in advance, a time-consuming process
* inspect execution logs and documentation (hard process, doc out of date or incomplete)

Specifications for:

* [10] Testing, malware detection, data structure repair, supporting program evolution, debugging
* [11] Design, Development and maintenance of software
* [20] testing, verification, anomaly detection, and debugging purposes.

Evaluation Metrics:

* Time vs.
  + Trace length, #Traces, unique events, support threshold, confidence threshold
* Mined rules
  + Support threshold, confidence threshold

Mining Metrics:

* Support & Confidence
  + [10,14,15,16]

From [5]:

1.3:   
Term coined by [6] (This paper describes *specification mining*, a machine learning approach to discovering formal specifications of the protocols that code must obey when interacting with an application program interface or abstract data type. Starting from the assumption that a

working program is well enough debugged to reveal strong hints of correct protocols, our tool infers a specification by observing program execution)  
Roots from [23,24]

1.3.3:  
**Dynamic:** Design under analysis is mostly correct, one can mine likely specifications by observing simulation or execution traces of the design. A common philosophy shared amongst these techniques is that frequently occurring patterns are likely specifications.  
E.g. [7]

**Static:** by reasoning about the program statically. on source code of a program to directly infer a set of rules that the program should obey.  
E.g. [8,9]

**Static** and **dynamic** analyses complement each other. Static and dynamic information can also be combined to generate better quality specifications.

1.3.4: Applications: Design Understanding, Verification, Synthesis, Classification (+description)

From [4]:

2.3: **Previous** work on dynamic inference […] assumes the test program executions are perfect and will not infer properties that are not completely satisfied by the traces.

Our […] designed for imperfect traces typically found in an industrial setting and can tolerate bugs in the trace as long as the majority of the trace is correct.

From [18]:

3.3: **Dynamic Specification Mining Techniques**  
Dynamic specification mining attempts to mine a specification from a set of execution traces.  
  
Generate and Check: Pattern Checking  
This approach uses a small predefined set of patterns to generate hypothesis specifications and check these against examples. This is motivated by the idea that most specifications use only a few small patterns. Taking either a known or inferred alphabet an instance of each pattern for each combination of symbols is generated and checked. For example, the pattern (ab)\* over alphabet {x; y; z} generates (xy)\*; (yx)\*; (xz)\*; (zx)\*; (yz)\*; (zy)\* to be checked. These patterns capture **desired** rather than undesired behaviour.

3.4: **Static Specification Mining Techniques**

Data-mining techniques on information extracted from the source code.

3.4.1: **Comparing** Static and Dynamic  
The source code describes the exact behaviour of the program, but does not include runtime information or indicate the most common behaviours. Additionally mining from source code may lead to specifications of implementation decisions not design decisions. For this, and other, reasons mining from execution traces seems preferable.

3.5: **Applications**: Program Comprehension, Testing, Verification, Security, Controlling Programs, Music Recognition (+infos)

3.6: Summary  
Noise = coincidental or uninteresting relations between events => creates many uninteresting specifications

From [21]:

Making sense of the observed behavior of complex systems is an important problem in practice. It arises in debugging, reverse engineering, specification mining, and modernization of legacy systems.

Formulas in LTL which are meant to distinguish between desirable and undesirable executions of a system (e.g. to explain the root-cause of a bug).

The particular choice of LTL is motivated by two observations:

* Concise; relatively easy for humans to comprehend
* LTL & CTL is widely considered to be the de facto standard for specifying temporal properties. Many engineers are familiar with its use.

From [22]:

With this approach, it is not possible to acquire new knowledge about the system directly from data, since it requires the designer to be very specific about the form of system properties that are investigated => Machine Learning??

From [28]:

Some work has been done on mining temporal specifications from libraries in the form of **automata** [6] or **rules** [4, 7]. Others have attempted to uncover latent behavior in **UML models** [5]. But the graph may be very complex and not practical that mining approaches generally rely on predetermined templates, such as the property patterns of Dwyer et al. [2]. Thus, only specific classes of constraints can be identified, instead of all the possibilities that a person could build, require, or understand data. Indeed, when a candidate pattern occurs very frequently, it might be the sign of triviality or tautology. Similarly, one may decide to consider only candidate patterns that are never contradicted in the learning data or tolerate a certain degree of falsification.

In other words, it is impossible to find what you are not explicitly looking for.

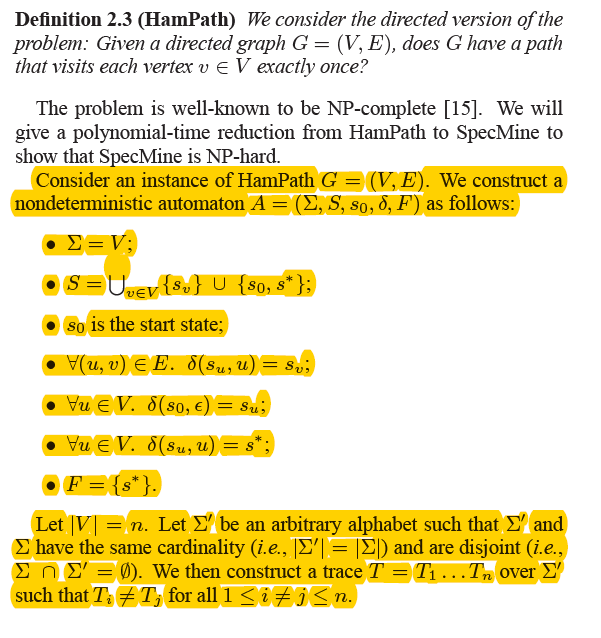
Penalize patterns that under support. Patterns with low support should be penalized compared to other patterns. Since low support patterns, while they can have high confidence, are usually more specific patterns that are not easily generalizable. As such, they are less interesting than patterns with high support.  
Penalize patterns that over support. Patterns with extremely high support are so general, that they are usually either trivial or not worthy of interest.

*Graph Mining*

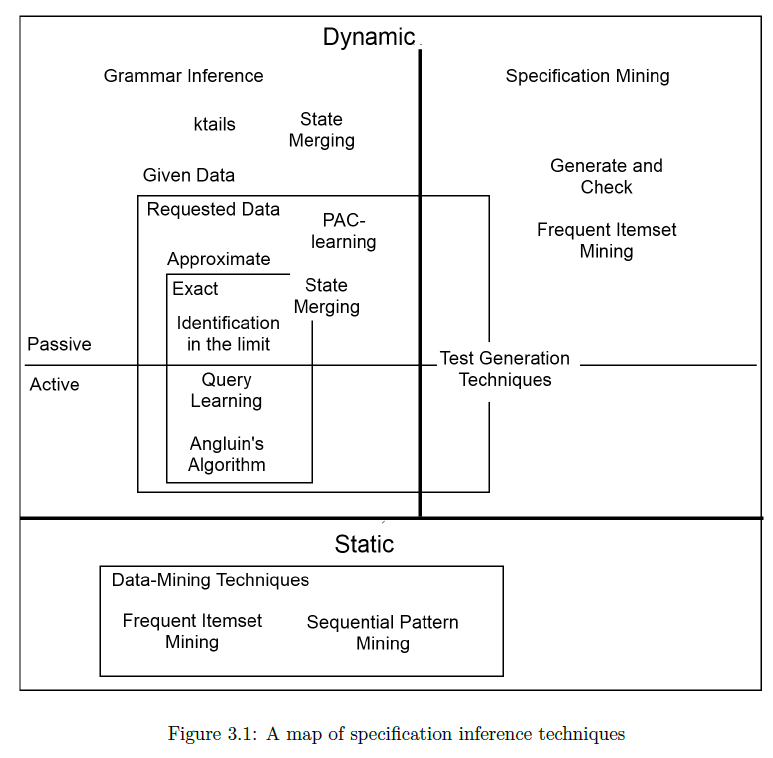
= Finden von häufigen Subgraphen, Graphen-Patterns

Wir wissen noch nicht wonach wir suchen!

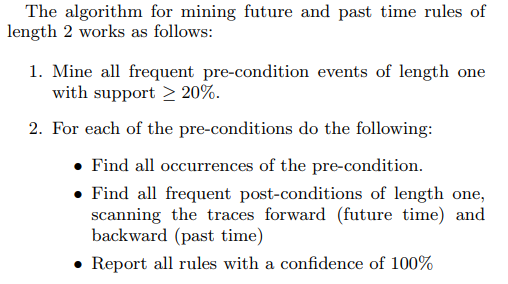
From [12]:



From [18]:



From [Automatic Steering of Behavioral Model Inference]:



* Pre condition support = absolute probability of nodes in BN

Literature

1. Multiresolution Abnormal Trace Detection Using Varied-Length n-Grams and Automata
2. Mining Invariants from Console Logs for System Problem Detection
3. Leveraging Existing Instrumentation to Automatically Infer Invariant-Constrained Models  
   (Synopsis)
4. Perracotta: Mining Temporal API Rules from Imperfect Traces
5. Specification Mining: New Formalisms, Algorithms and Applications
6. Mining Specifications
7. The Daikon system for dynamic detection of likely invariants
8. Bugs as deviant behavior: A general approach to inferring errors in systems code
9. Synthesis of Interface Specifications for Java Classes
10. General LTL Specification Mining (Texada)
11. Javert: Fully Automatic Mining of General Temporal Properties from Dynamic Traces
12. Symbolic Mining of Temporal Specifications
13. Online Inference and Enforcement of Temporal Properties
14. Mining temporal rules for software maintenance
15. Mining Branching-Time Scenarios
16. Mining modal scenario-based specifications from execution traces of reactive systems
17. Behavioral Resource-Aware Model Inference
18. An Overview of Specification Inference
19. SMArTIC: Towards Building an Accurate, Robust and Scalable Specification Miner
20. Mining Timed Regular Expressions from System Traces
21. Learning Linear Temporal Properties
22. A Decision Tree Approach to Data Classification using Signal Temporal Logic
23. The synthesis of loop predicates
24. Finding Invariant Assertions For Proving Programs
25. Synergizing Specification Miners through Model Fissions and Fusions (SpecForge)
26. ARTINALI: Dynamic Invariant Detection for Cyber-Physical System Security
27. Periodic Task Mining in Embedded System Traces
28. Towards the Automated Recovery of Complex Temporal API-Usage Patterns
29. GKtail\_Automatic\_generation\_of\_software\_behavioral\_models