A Comparative Study of Specification Mining Methods for SoC Communication Traces

Md Rubel Ahmed[1], Hao Zheng[2]
U of South Florida, Tampa, FL
{mdrubelahmed,haozheng}@usf.edu

Parijat Mukherjee[3], Mahesh C. Ketkar[3], Jin Yang[3]
Intel, Hillsboro, Folsom, Hillsboro, OR, CA, OR
{parijat.mukherjee,mahesh.c.ketkar,jin.yang}@intel.com

***Abstract*—This paper aims to study how existing trace mining methods work, and their strengths and weaknesses in the context of communication-centric system-on-chip (SoC) traces. SoC traces are unique because of the inherent concurrency, which makes SoC trace mining very challenging. We select seven well-known trace mining methods both from the hardware and software domains. We perform comprehensive experiments, and offer our understanding of the pros and cons of each mining method. We evaluate the interestingness of the mined outcomes for each tool on a benchmark trace set. This trace benchmark includes sophisticated communication traces generated synthetically and realistically. We provide a comprehensive analysis of the performance of the tools, and release the benchmark trace dataset to facilitate future works in SoC trace mining.**

***Index Terms*—SoC validation, trace analysis, specification mining, SoC execution model, system-on-chip**

I. Introduction

Trace analysis or logging is a widely used debugging method for hardware and software systems. Tracing has enabled many complex systems to grow, evolve, and function correctly these days. Today's computing systems are so complex that it is not feasible to debug the tiny SoC in our wristwatch manually. Many tools and algorithms are introduced to tackle the problem from different perspectives [1]. However, the behavior of a computing system is observed over its execution traces. Therefore, the attempts to automate the debugging through trace analysis becomes a rich avenue of research, and many techniques have been developed. A systematic comparison of such tools and techniques solely targeting SoC communication traces can help us understand these methods' strengths and weaknesses, and applicability on SoC trace mining.

Modern SoCs house various intellectual property (IP) blocks together procured from a wide range of sources. Such SoCs contain complex and concurrent interconnects to support parallel operations. Besides sophisticated system level protocols are incorporated to realise these operations. Debugging the correctness of such protocols posses a big challenge as the design evolves. Hence communication-centric debug is proven to be more practical than traditional computation-centric debug methods [2]. Trace analysis helps us to create debug specifications with statistical rigors. For example, the sample SoC in Fig. 1 implements two message flows or flow specifications or IP communication protocols for CPU downstream write. Such flow specifications are necessary to ensure the correctness of the corresponding write policies. Still, in practice, they are not available as the design evolves, and manually updating the protocol documentation is often impractical for large and complex SoC designs.

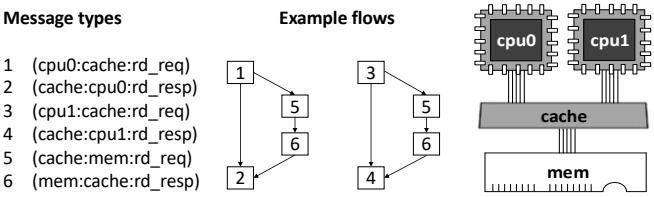

Fig. 1: A CPU downstream write flow [3] for a sample SoC

There exists a pool of tools that can work on the traces, such as one in eq. 1, and analyze them to find interesting patterns which can serve as flow specifications. Readers are referred to [3] to know more about the flows and SoC execution traces. We want to investigate if the sequential patterns or *Ground Truths GT*[1] patterns derived from the example flows such as $(1, 2)$, $(1, 5, 6, 2)$, $(3, 4)$, $(3, 5, 6, 4)$ can be mined from the execution trace/s such as eq. 1 using different trace mining tools. An example trace from executing the flows in Fig. 1 is

$$(\{1, 3\}, 1, 2, 5, 1, 5, 6, 2, 4, 6, 2) \tag{1}$$

The numbers in the curly braces indicate that the transaction `cpu0:cache:rd_req` and `cpu1:cache:rd_req` are occured in the same timestamp, or in parallel. So the true order between messages (or events) '1' and '3' can not be known. We select seven tools from hardware and software trace mining domains and evaluate their strengths and weaknesses. We perform comprehensive experiments to answer the following questions.

1) Which tools can deal with SoC communication or Transaction Level Model (TLM) traces?
2) Which tool/s can handle complex and concurrent communication traces?
3) What are the factors that cause performance variation among different tools on SoC traces?

To the best of our knowledge, this is the first paper that presents a comprehensive study of different mining tools for SoC transaction level traces. The **contributions** of this work are two folds:

[1]Correct ordering of messages are referred as *GT* patterns

1) We present a benchmark trace set that have vital characteristics of real-world SoC traces, which can be used to comprehensively evaluate different SoC trace mining methods
2) A thorough evaluation of seven representative trace mining tools and algorithms on the benchmark

The paper is organized as follows. The next section gives a brief review and classifies existing pattern mining methods. Section III provides an overview of the selected mining tools. Section IV describes the benchmark trace generation. Section V presents details of the experiments and observations. The last section concludes the paper, and offers future directions.

II. Classification of Trace Miners

We review tools and techniques that mine sequential patterns or extract models from traces. This list includes many tools both from the software and hardware domains. The classification of the existing tools broadly depends on their underlying principles, methods, and output reports. We can loosely separate them in the following classes, but an accurate margin is often very tough to draw between them.

Automata Based Miners Automaton based approaches construct Finite State Machines (FSMs) to interpret or synthesize the traces in the form of a set of states and transitions. The synthesized automaton can also be regarded as execution models [4]–[6]. Often the model synthesis is formulated as a constraint satisfaction problem, and SAT finds acceptable solutions. Both tools [4] and [5] achieve similar objectives. However, we choose [4], and refer as *Trace2Model* from this category, comparing the number of citations it received than other.

LTL Miners Producing Linear Temporal Logic (LTL) [9] formula to correctly explain the traces or system behavior is studied in many works. Model inference approach *Synoptic* [7] first mines invariants from logs of sequential execution traces where concurrency is recorded in the partial order. It then generates an FSM that satisfies the mined invariants. It builds execution models to enhance programmer comprehension of the traces. Another tool called *Texada* [8] works with user-specified templates in the form of LTL and produces instances of that formula using some interestingness measures. *Texada* is considered the representative tool from this category, as it claims to perform better than many other tools.

Rule based Pattern Miners Analyzing traces statistically and deriving rules for the events is a rich venue of research. Some works from this category are [10], *Perracotta* [11], *FlowMiner* [3]. These works scan the traces to deduce rules that can formulate the event relation in the traces. Work *FlowMiner* can find patterns from complex SoC trace and tool *Perracotta* is a benchmark tool for many other tools. So both of them are considered to be studied in this paper.

Assertion based Miners Assertions are interesting and useful resource for SoC debug. Assertions are often extracted in the form of sequential patterns [12], [13]. We consider [13] as a representative tool which we refer to as *TLMine*.

General Sequential Pattern Miners Myriad of works from the data mining community try to discover patterns in sequential data [14]. Agrawal and Srikant [15] first proposed their works on frequent itemset mining in the form of sequential pattern. There exists a handful of algorithms designed for mining sequential pattern mining. Among many well-known algorithms, *PrefixSpan* [16] is considered because of its performance superiority.

Machine Learning Methods Machine Learning algorithms have already been contributing to solving exciting pattern mining problems. SoC traces possess strong temporal relations among the events. Therefore Recurrent Neural Networks (RNN) have become an excellent tool to find the probabilistic relations among the events, finally forming patterns. An example work presented in [17] that utilizes Bayesian Inference to interpret the execution model using the LTL formula. [18], [19] are example works that employ a special type of RNN for mining patterns or specifications. We take [18] because it works on non-trivial SoC design traces.

III. Selected Trace Miners

For tool selection, we primarily focus on the generality of each mechanism that deals with trace mining. We briefly describe the tools listed below and summarize the tools' critical features in Table I.

A. PrefixSpan

This is a sequential pattern mining tool based on a pattern-growth algorithm that employs depth-first-search [14] technique to explore the candidate pattern search space. It takes a minimum support threshold to identify frequent patterns of single items in the sequence database. Then it generates more extended patterns by left or right extensions sequentially. This algorithm comes with a massive cost in memory usage but avoids searching for patterns that do not exist in the database. We obtain the implementation of this algorithm that is ported with the open-source data mining library called SPMF [20]. For the trace shown in (1) *PrefixSpan* could find 2463 patterns which also include four *GT* patterns $(1, 2)$, $(1, 5, 6, 2)$, $(3, 4)$, $(3, 5, 6, 4)$.

B. Trace2Model

It works on system execution traces and synthesizes the traces into a concise and abstract model. The synthesized model is a finite state automaton (FSA). It finds the edge transitions by solving a constraint problem using a Boolean (SAT) solver. We utilize the incremental searching version of the tool. It starts with N states, and then looks for $(N+1)^{th}$ state to be satisfied by a C Bounded Model Checker. The synthesized model helps to understand the system behavior under a particular program execution. It utilizes a sliding window technique as an optimization method for scalability. Other than the window size *w*, there is another hyperparameter *l* that controls the degree of generalization of the learned automaton. It shows significant improvements over state merge algorithms for generating behavioral models. The model in

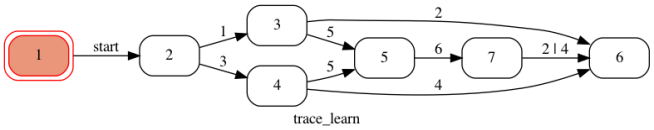

Fig. 2: FSA model produced using the tool *Trace2Model*

Fig. 2 is resulted from the trace in (1). The numbers on the edges represent the message indices of the example flows.

C. TLMine

TLMine is an episode mining framework that mines assertions in the form of frequent episodes from simulation traces of transaction-level models (TLMs). The algorithm works for longer episodes incrementally. It describes an abstraction of communication actions as events, and an episode is an ordered sequence of events. It looks for episodes of multiple elements for a given support[2] and confidence thresholds. An episode is called frequent if it appears at a certain number of time windows. It introduces the confidence of an episode, which is the ratio between an episode's supports values and its prefix. As they mine assertions, 100% confidence value is employed for an episode to be considered an assertion. It shows the benefit of applying the sliding window technique over sequential pattern mining in terms of run time and number of episodes mined. For the trace in (1), this tool could find 30 episodes. Among them {1, 2} is present as the only *GT* pattern.

D. FlowMiner

FlowMiner can find meaningful patterns from the IP communication traces of an SoC execution. These patterns are supposed to be utilized as flow specifications for IP communications protocol debug. The specialty of this tool is that it mines patterns from highly concurrent and interleaved traces. This tool's input is a set of execution traces over messages observed in various communication interfaces in an SoC design. It utilizes the well-known support confidence framework to find binary rules which are also invariant over the traces. Further, mined binary rules are chained to more extended patterns through some inference techniques, while patterns could also be treated as invariants. [3] also shows some optimization techniques to reduce the mining complexity and improve the mined patterns quality. The tool could find all the four *GT* patterns and seven other patterns from the example trace.

E. Perracotta

This work mines temporal properties in the form of API rules from program execution traces. It mines properties based on some user-defined templates. There is an inference engine that works on the input traces incorporating a set of property templates. For each property template, an FSM is used to scan the traces to find that property's instance. This FSM plays a vital role in making this tool scalable to a large set of traces.

The inferred properties go through further post-processing before being reported to the user. There are eight different property templates which are abstracts of a set of concrete properties. At first *Perracotta* targets mining temporal properties of two events or values. Among the eight property templates, we find alternating properties are most interesting for the SoC communication pattern mining context as patterns are also invariants. Therefore, we further restrict our discussion on this tool to the alternating properties only. The paper also describes a technique to produce *alternating chains* that is the composition of alternating properties to deduce complex rules out of smaller properties. This chaining saves massive computation inherent to finding longer patterns from the sequential traces.

The inference engine uses a metric *satisfaction rate* as a threshold to rank alternating properties. A trace is partitioned into small sub-traces, and then a conformity check is done in these sub-traces to calculate the satisfaction rate. Fourteen patterns are found from the example trace (1). Among them, $(3, 4)$ is the only *GT* pattern obtained in the form of alternating patterns.

F. Texada

Texada mines program behavior in the form of LTL formulae. The input to this tool is an LTL property template and execution log or traces. It outputs a set of concrete LTL formulae which conforms to the property template. It utilizes an efficient representation of input traces to avoid unnecessary traversal over the traces. Property instances are validated over the traces in a linear recursive fashion. *Texada* also includes some optimization techniques such as state memoization for validating property instances to further reduce the search complexity of the tool. We use the LTL formula `G(x → X(F(y)))` to find all the instances of x, y where `"x is always followed by y"`. With this pattern template, *Texada* finds 15 patterns. Among them $(1, 2)$, $(1, 5)$, $(1, 6)$, $(3, 4)$, $(3, 5)$, $(3, 6)$, $(5, 6)$, $(6, 2)$ are interesting as they conform to message orderings of the *GT*.

G. YLSTM

Work in [18] (we refer to as *YLSTM*) utilizes LSTM (Long Short Term Memory based on RNN) to extract sequential patterns from SoC transaction-level traces. LSTM can capture "long-term" dependencies and has many applications in natural language processing, or sequence modeling works. Mined patterns resemble dependencies between various events. This paper exclusively focuses on the concurrent nature of SoC traces. At first, a set of LSTM networks are trained on the SoC traces for different lengths of sequences. The trained networks can predict the next event upon given an input sequence of the particular length it has been trained. However, the input to a model that take length l sequences can have $l!$ number of candidate sequence, which can explode the search space. To handle this phenomenon, all the unique events observed in the traces are fed to the first LSTM model that only predicts the next event with a probability threshold. The output of this model (a set of sequences of length two) is then fed to the next model for extracting sequences of length three. This

[2]Please checkout to the original paper for definition.

TABLE I: Tools Features

Tool	Application	Method	Input	Output
FlowMiner	Message flow mining, design automation, SoC validation, Work flow mining, Specification inference	Association rule mining, inference for chaining, use of support-confidence framework	SoC concurrent communication traces, Support confidence threshold	Flow specifications in the form of sequential patterns
Perracotta	API rule mining, Mining program behavior, Program correctness checking	Uses property templates and looks for template instances using FSMs. Uses chaining to obtain complex rules	Set of traces, Concurrency not considered satisfaction rate	property instances, patterns
PrefixSpan	Sequential data analysis, click stream, CyberSecurity	recursively grows frequent patterns using depth first search, uses projected database	Sequential data, takes multiple traces support threshold	Sequential patterns for that threshold
TLMine	Digital designs, assertion based verification, transaction level models	Mines episodes incrementally for a given support, confidence threshold, uses maximum lifetime of a transaction for windowing	TLM level traces, parallel executions are interleaved in the traces, support, confidence threshold, maximum life time of a transaction	assertions in the form of frequent episodes of different length
Texada	Program behavior analysis, model correction checking, mining relation between the events in the program execution logs	Uses template, looks for template instances, checks for counter example over the traces	Multiple traces, concurrency is expressed as interleaving of events in the traces, LTL formula or templates according to the user's interest	LTL formula instances
Trace2Model	Model building from the program execution traces for hardware or software system	builds nfa and then increments the states until counter example is found in the trace, finally converts the nfa to dfa	Sequential traces, concurrency is not considered, degree of generalization	FSM model that conforms to the observed traces
YLSTM	SoC validation, SoC system protocol specification mining, electronic design automation	trains LSTM networks to find the sequential dependency among the events in a trace, incrementally extracts sequential patterns from the LSTM models	SoC system level transaction traces, probability threshold	Sequential patterns that can describe the SoC system protocol specification

extraction continues until patterns are extracted from all the trained models. For this approach to be practical, a lot of trace data is needed for training. So result for the example trace is not listed here.

IV. Benchmark Preparation

To the best of our knowledge, there is no SoC transaction-level trace benchmark dataset publicly available. Therefore, we describe how a benchmark trace dataset could be produced. We plan to make this benchmark freely available. We divide the generated traces into two major types. The first type is synthetic traces, which include traces generated by simulating concurrent executions of a set of message flows to mimic the execution of SoC designs. For example, concurrency in flow executions, parallelism of event occurrences, and recurrences of flow instances are essential features of this synthetic trace set's traces. The primary motivation behind preparing this trace set is to compare the quality of the mined patterns with the *GT* patterns. The other type of traces are SoC communication traces captured from simulating a multi-core SoC model developed in gem5 [21] environment. The second trace set allows us to validate mining methods in a practical setting.

Synthetic Trace Generation We produce this set of traces using ten message flows with a total of 64 flow instances as *Ground Truth (GT)* patterns. Section I shows four such patterns obtained from example flows in Fig. 1. It should be noted that four *GT* patterns are found from the two example flows which testifies the fact that some flows have branches. Similar is true for the ten flows we consider to generate the synthetic traces. The patterns are executed in different order to produce three sets of synthetic traces listed in Table III. We intend to simulate concurrency as well as recurrence to mimic the actual SoC traces characteristics. This set's benefit is that we have *GT* patterns in stock to compare the patterns mined by different methods.

GEM5 Trace Generation gem5 is a computer system simulation tool that can run in one of two modes: *Full System (FS) simulation* and *Syscall Emulation (SE)*. We collect traces from both modes. Fig. 3 is a simplified multicore SoC model developed to generate traces. It contains four x86 cores, each of them has a private data cache (64kB) and a private instruction cache (16kB). All cores share a 256kB level 2

TABLE II: Trace Mining Summary, RT = Run Time

Traces	*SNI*		*SI*		*MI*		*FS*		*Snoop*		*Thread*	
Tool	#bin. pat.	RT	#bin	RT	#bin	RT	#bin	RT	#bin	RT	#bin	RT
FlowMiner	67	23s	122	36s	122	28s	430	1.5hr	464	134s	460	490s
Perracotta	30	3s	2	2.5s	N/A		7	300s	45	75s	46	62s
Texada	58	2s	98	5s	N/A		1711	280s	2052	5s	6601	50s
TLMine	12	184s	0	300s	N/A		–	–	26	1528s	–	–
YLSTM	82	N/A	82	N/A	N/A		178	N/A	180	N/A	208	N/A

TABLE III: Synthetic Traces

Trace	Description
SNI	Single event, non-interleaving pattern traces. A *GT* pattern is arbitrarily picked up from the gt pattern pool. It is then executed from the start to the end event before the next pattern is selected. For example, the flow instance patterns from Fig. 1 are executed one after another. There are 100 traces of 59 unique messages totalling a length of 177114.
SI	Single event, interleaved pattern trace, where a few patterns are selected (randomly) interleave among themselves to mimic the idea that a single-core SoC is running multiple tasks interleaving tasks by priority. There are 100 traces of 59 unique messages. The length of this set is 282192.
MI	Multi-event, interleaving patterns traces. As the name suggests, multiple events might be present in each step, such as first step of the trace in (1). This trace set tries to mimic a multi-core system running multiple tasks in parallel.

TABLE IV: Gem5 Traces

Trace	Description
FS	One FS mode trace. Boots Linux kernel 4.1.3. No workload is executed only kernel boots and exits. It has 59 different messages that comprise a length of 40736240.
Snoop	Activates simple snoop protocol implemented for the traditional memory system. Two traces comprised of 104 unique messages of length 732118 combined.
Threads	Two SE mode traces, one captured running Paterson algorithm which is a traditional mutex algorithm. Another trace is captured for distributing a matrix addition. There 135 messages in the traces which has combined length of 8094098.

cache. There is also a DDR3_1600_8x8 memory controller, which has an address space of up to 4GB. These IPs are connected via high-speed and concurrent buses that can handle multiple requests simultaneously. IP blocks in this SoC communicates with each other to realise various operations. We execute different binaries on this bare bones setup to extract realistic and sophisticated traces. We instrument the model with 19 communication monitors to observe the *packet*s (unit of communication) over different communication links. Table IV describes GEM5 simulation traces.

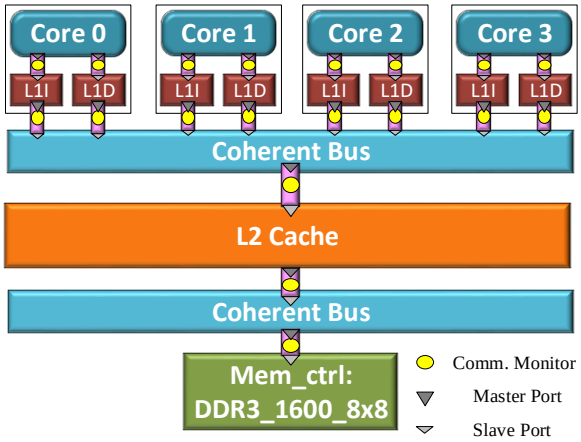

Fig. 3: GEM5 model for tracing

V. Experiments and Observations

One of major goals of this study is to find which tools can produce meaningful results for the benchmark traces within a reasonable time limit. All the experiments are conducted on system of 3.2GHz Intel core i5 processor, 8GB of RAM. Traces *FS, Snoop, Thread* had multiple events in each step. However, these traces are simplified to a single event each step format so that tools *Perracotta*, *Texada*, *TLMine*, *PrefixSpan* and *YLSTM* can be applied to them. Only tools that can complete the mining within 2hrs time limit are listed in Table II.

We only compare baseline performance; for example, the run time and number two event properties. *FlowMiner*, *Perracotta*, and *Texada* do better in finding patterns in most cases. The tree representation of property templates and state memoization are the key factors for such shorter run time. Tool *PrefixSpan* runs out of memory in all the experiments; therefore, results are not listed. The main reason behind this is the projected database it creates for every new pattern it mines that takes up the available memory space. The approach *YLSTM* needs training before any pattern can be extracted. So the run time is not listed for this tool as the training depends on the number of epochs for which it has been trained. We run for 100 epochs for each of the traces for which the tool takes 10minutes to 4hrs to complete the training. The tool *Trace2model* produces complete execution scenarios that takes more than 4 hours for the two set of synthetic traces, and snooping traces. It contains 37 states for the *SNI* traces which signifies that some temporal relations are not captured

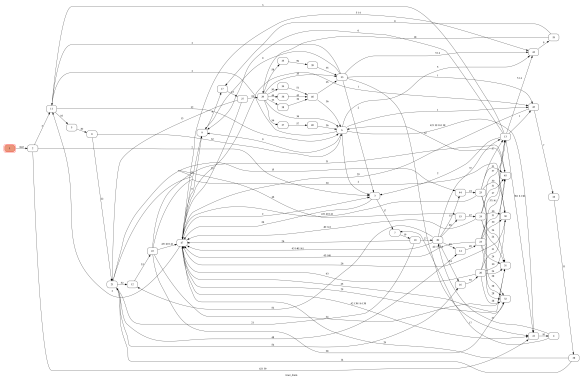

Fig. 4: Model generated for *SNI* traces by *Trace2Model*

as there are 59 events in this set. Synthesized model in Fig. 4 shows many transitions are listed in the model that actually are not real, they appears as they are temporally related. Tool *TLMine* could not finish mining episodes from *FS* and *Thread* traces so the 6th row of Table II are empty for this tool. This phenomenon can be explained from the fact that for every iteration of episode search, it produces a lot of invalid candidate episodes, and search time increases exponentially.

In all the cases, *Texada* and *Perracotta* perform better in terms of run time, but *FlowMiner* performs better in terms of pattern number in some cases. Besides *FlowMiner* could find patterns from the particular type of traces that exerts concurrency inherently. It finds patterns with statistics which helps us to find assertions from the traces. A special features of tools *FlowMiner* and *YLSTM* is that they incorporate structural features available in the SoC traces. This special attention helps them to reduce the run time as well find more interesting patterns. We put special attention to the tools *YLSTM* and *FlowMiner* as they can find complex patterns in pattern length. Flow specifications are mostly expressed in longer patterns; therefore, these two tools show a higher potential for mining message flows.

VI. Conclusion

This paper considers some representative works from different domains that deal with different types of traces and apply them to SoC communication traces. Comparative study results have been listed. It is found that LTL miners are faster in run time, but rule based miners provide more information about the mined patterns. A benchmark of the SoC communication trace dataset has also been presented to facilitate future research on this topic. The performance of different tools on this benchmark has been discussed. Crucial advancement in SoC validation methods can be achieved by incorporating machine learning with domain-specific features.

References

[1] S. Mahmud, B. Olney, and R. Karam. Architectural Diversity: Bio-Inspired Hardware Security for FPGAs. In *Third International Verification and Security Workshop (IVSW'18)*, pages 48–51, 2018.

[2] K. Goossens, B. Vermeulen, R. v. Steeden, and M. Bennebroek. Transaction-based communication-centric debug. In *First International Symposium on Networks-on-Chip (NOCS'07)*, pages 95–106, 2007.

[3] Md Rubel Ahmed, Hao Zheng, Parijat Mukherjee, Mahesh C. Ketkar, and Jin Yang. Mining Message Flows from System-on-Chip Execution Traces, In *22nd International Symposium on Quality Electronic Design (ISQED'21)*, pages 374-380, 2021.

[4] Natasha Jeppu, Tom Melham, Daniel Kroening, and John O'Leary. Learning concise models from long execution traces. In *DAC '20*, June 2020.

[5] Hao Zheng, Md Rubel Ahmed, Parijat Mukherjee, Mahesh C. Ketkar, and Jin Yang. Model synthesis for communication traces of system-on-chip designs, arXiv-2021.

[6] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. POPL '02, page 4–16, New York, NY, USA, 2002. Association for Computing Machinery.

[7] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D. Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 267–277, 2011.

[8] C. Lemieux, D. Park, and I. Beschastnikh. General ltl specification mining (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 81–92, 2015.

[9] E. A. Emerson. *Handbook of Theoretical Computer Science: Volume B: Formal Models and Semantics*. Elsevier, 995-1072.

[10] W. Li, A. Forin, and S. A. Seshia. Scalable specification mining for verification and diagnosis. In *Design Automation Conference*, pages 755–760, 2010.

[11] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: Mining temporal api rules from imperfect traces. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, page 282–291, New York, NY, USA, 2006. Association for Computing Machinery.

[12] P. Chang and L. . Wang. Automatic assertion extraction via sequential data mining of simulation traces. In *2010 15th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 607–612, 2010.

[13] Lingyi Liu and Shobha Vasudevan. Automatic generation of system level assertions from transaction level models. *Journal of Electronic Testing*, 29(5):669–684, Oct 2013.

[14] Philippe Fournier-Viger, J. Lin, R. Kiran, Y. Koh, and R. Thomas. A survey of sequential pattern mining. 2017.

[15] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, page 487–499, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[16] Jian Pei, Jiawei Han, B. Mortazavi-Asl, Jianyong Wang, H. Pinto, Qiming Chen, U. Dayal, and Mei-Chun Hsu. Mining sequential patterns by pattern-growth: the prefixspan approach. *IEEE Transactions on Knowledge and Data Engineering*, 16(11):1424–1440, 2004.

[17] A. Mrowca, M. Nocker, S. Steinhorst, and S. Günnemann. Learning temporal specifications from imperfect traces using bayesian inference. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2019.

[18] Y. Cao, P. Mukherjee, M. Ketkar, J. Yang, and H. Zheng. Mining message flows using recurrent neural networks for system-on-chip designs. In *2020 21st International Symposium on Quality Electronic Design (ISQED)*, pages 389–394, 2020.

[19] Tien-Duy B. Le and David Lo. Deep specification mining. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 106–117, New York, NY, USA, 2018. ACM.

[20] Philippe Fournier-Viger, Antonio Gomariz, Ted Gueniche, Azadeh Soltani, Cheng-Wei Wu, and Vincent S. Tseng. Spmf: A java open-source pattern mining library. *J. Mach. Learn. Res.*, 15(1):3389–3393, January 2014.

[21] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.