Topaz: Mining High-Level Safety Properties from Logic Simulation Traces

Ahmed Nassar Fadi J. Kurdahi
EECS Department, University of California, Irvine, CA 92697, USA
{anassar,kurdahi}@uci.edu

Salam R. Zantout
American University of Beirut, Lebanon
salam.zantout@gmail.com

***Abstract*—Formal specifications are hard to formulate and maintain for evolving complex digital hardware designs. Specification mining offers a (partially) automated route to discovering specifications from large simulation traces. In this paper, we embark on a novel and rigorous mining methodology (data preparation, mining algorithms, selection criteria, etc.) for finite-state automata checkers using an iterative and interactive mining tool, called Topaz. Topaz is evaluated using an open-source 32-bit RISC CPU design as a case study to demonstrate extraction of complex temporal properties cross-cutting through all CPU pipeline stages, guided by the CPU instruction set specification.**

I. Introduction

Thorough functional verification helps to reduce design respins due to functional bugs and enables first-pass silicon success. All verification techniques, static or dynamic, decide correctness of an implementation against some notion of a specification (in the form of *properties*, golden models, etc.). Unfortunately, formal specifications are not always used mainly because of their high development and maintenance cost and complexity, since they require substantial expertise in formal specification languages and their decision procedures, as well as abstraction techniques. This difficulty hinders formal specifications from coping with agile, fast-paced development environments that foster rapidly evolving systems.

A key insight is that specification of a design module can be implicit in how it is being used by, and reacting to, high-quality client code [18]. Therefore, *specification mining* [13] emerged as an automated technique used to discover formal specifications of systems from samples of their behavior. From Table I, inferred properties can take many forms and can be examined, refined, abstracted or corrected by designers and verification specialists. Once validated, these specifications will drive functional verification, regression testing, or serve as invariants or documentation. Specification mining for digital hardware designs has been gaining much traction recently [3], [6], [7], [8], [10], [12], [15], [17]. Many of these tools are of great practical value but most of them still suffer from serious limitations (detailed in Section II) such as the use of inadequate design state abstractions, limited expressive power, and using a finite time window which can easily miss long-range temporal relations.

In this paper, we present *Topaz*, a tool that discovers design safety properties, in the form of deterministic finite automata (DFAs), from large simulation traces. Topaz relies on a novel specification mining, and language learning, technique that is the first (to the best of our knowledge) to use *multiple sequence alignment* (MSA) [5] in order to obviate many limiting assumptions made by prior tools and resolve the long-standing *initial-state uncertainty* problem in *offline* specification mining [16]. Using MSA, Topaz can reconstruct properties with *abstract* state spaces which do not merely duplicate the hidden design state space and whose sizes are dictated solely by the complexity of observed simulation traces and can capture temporal relations among widely separated logic events. More abstraction can then be selectively applied among *simulation-equivalent* states [2] to trade off precision for conciseness or understandability. MSA also naturally adopts a scoring scheme that can be used to quantify the statistical significance of inferred specifications and reject spurious properties. Topaz enables controlling abstraction levels of mined properties by taking advantage of user-defined logic events, that can enable extracting transaction-level properties from RTL designs.

The rest of the paper reviews prior work on specification mining, and then develops terminology for later sections. Next, we explain the Topaz mining flow, followed by a case study on the open-source Amber CPU design. We conclude with a summary of results and venues for future work.

II. Related Work

A brief survey of specification mining tools for digital hardware designs is shown in Table I. *Inferno* [10] scours simulation traces for *transaction diagrams* and generates Verilog checkers for them. However, Inferno identifies the internal design state with its output control signals, thus producing FSMs having modest precision in constraining observable design behavior. *Dianosis* [15] analyzes simulation traces and builds property candidates over all combinations of signals given a set of parameterized basic property templates, such as OVL checkers. Extracted basic properties are then recursively combined into higher-level transactions. However, it cannot, for example, express (non-)consecutive open-ended repetition. Moreover, being template-based restricts expressiveness.

In [3], extracted specifications express sequential relations among *events* which are unique combinations of signal values. Extracted assertions take the implication form $\mathbf{A} \Rightarrow \mathbf{C}$, where both $\mathbf{A}$ and $\mathbf{C}$ are *episodes* of events. To reduce computational complexity, only relations among episodes within a preset sliding window are considered. Another window-based tool is [6], where a simulation trace is divided into fixed-size

978-3-9815370-6-2/DATE16/

windows. In PropGen [7], extracted properties are low-level Boolean formulas over the inputs, state variables and outputs within a moving finite window W. Such formulas might not give as much insight as higher-level or more abstract properties. All window-based tools have severely limited ability to discover long-range temporal relations among events, which are quite common in pipelined designs. IODINE [8] extracts dynamic invariants from logic simulation traces. The FSMs mined by IODINE are based on *explicit* state vectors, which lacks any abstraction or generalization, and is thus not scalable and may duplicate design bugs in the extracted specification. In [12], recurring temporal behaviors matching a set of pattern templates in a trace are mined and synthesized into more complex patterns by using inference rules. Finally, GoldMine [17] uses data mining to automatically generate LTL assertions. However, the GoldMine methodology is also applicable within a bounded time window and "*cannot generate unbounded safety or liveness properties*". In this paper, we allow the salient design behaviors speak for themselves, without constraining the *forms* of specifications inferred or capturing temporal relations within a bounded time window.

III. Preliminaries

In this section, we develop the terminology and notation used throughout the paper. In a synchronous digital design, a formal specification consists of one or more *properties*, where each property establishes a relation between signal values spanning multiple, not necessarily consecutive, clock cycles. Binary signals taking part in a property specification are represented by a set $\mathbf{AP}$ of atomic propositions. The design under verification (DUV) can be described by a labeled transition system [2] $TS = (S, \rightarrow, I, \mathbf{AP}, L)$, where S is the set of states, $\rightarrow \subseteq S \times S$ is a transition relation, $I \subseteq S$ is a set of initial states, and $L : S \rightarrow 2^{\mathbf{AP}}$ is a labeling function. A *path* in TS is a sequence of states $s_0, s_1, s_2, \ldots$ such that $s_0 \in I$ and $s_i \rightarrow s_{i+1}$ for $i \geq 0$. For every path $s_0, s_1, s_2, \ldots$, there is a *trace* $L(s_0), L(s_1), L(s_2), \ldots$. The set $Traces(TS)$ is the set of all traces of TS starting from an initial state. Let $(2^{\mathbf{AP}})^\omega$ be the set of infinite words over $2^{\mathbf{AP}}$. A linear-time (LT) property φ over $\mathbf{AP}$ is a subset of $(2^{\mathbf{AP}})^\omega$. Transition system TS satisfies LT property φ, written as $TS \models \varphi$, iff $Traces(TS) \subseteq \varphi$. A LT property φ is a *safety property* if every violating trace $\tau \in (2^{\mathbf{AP}})^\omega$ (i.e., $\tau \notin \varphi$) has a *finite bad prefix* (i.e., a prefix all of whose infinite extensions also violate φ). A regular safety property is a safety property whose bad prefixes constitute a regular language and, hence, can be recognized by a standard finite automaton [2] given by a tuple $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$. An automaton $\mathcal{A}$ can be *deterministic* (a DFA) or *nondeterministic* (a NFA). Topaz extracts DFAs from simulation traces to capture regular safety properties, which can then be verified formally or by simulation.

Given $\mathbf{AP}$, the set of logic signals used in specification mining, the alphabet set Σ of an extracted DFA is a *partition* over $2^{\mathbf{AP}}$. That is $\bigcup_{\sigma \in \Sigma} \sigma = 2^{\mathbf{AP}}$ and $\sigma_1 \neq \sigma_2 \Rightarrow \sigma_1 \cap \sigma_2 = \emptyset$. Users of Topaz specify the alphabet symbols of Σ as Boolean formulas over $\mathbf{AP}$, since the set of Boolean formulas over $\mathbf{AP}$ is isomorphic to the power set of $2^{\mathbf{AP}}$. For any $p, q \in Q$, if $q \in \delta(p, \sigma)$, this is abbreviated as $p \xrightarrow{\sigma} q$.

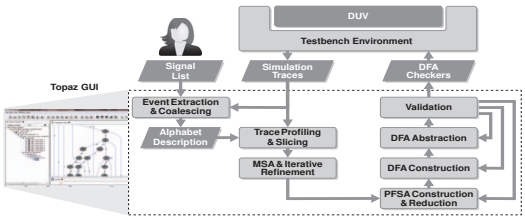

Fig. 1: Topaz specification mining flow. Topaz is a Java application with 55,000 lines of code.

A transition system TS can be verified to satisfy a regular safety property φ described by a NFA $\mathcal{A}^\varphi = (Q, \Sigma, \delta, Q_0, F)$, where $\Sigma = 2^{\mathbf{AP}}$, with the help of the product $TS \bigotimes \mathcal{A}^\varphi = (S', \rightarrow', I', \mathbf{AP}', L')$, defined in [2] as:

$$S' = S \times Q, I' = \{(s_0, q) | s_0 \in I, \delta^{-1}(q, L(s_0)) \cap Q_0 \neq \emptyset\} \quad (1)$$

$$\forall s, t \in S, \forall p, q \in Q : s \rightarrow t, p \xrightarrow{L(t)} q \Rightarrow (s, p) \rightarrow' (t, q) \quad (2)$$

To verify that $TS \models \varphi$, it is sufficient to check that for all $(s, q) \in S'$ reachable from I', we have $q \notin F$, since accepting states in F indicate violation of φ.

IV. Specification Mining Flow

Topaz specification mining flow is shown in Fig. 1.

Trace Recording. Topaz consumes a database of VCD simulation traces generated by the testbench environment that provides sufficient coverage of system behavior.

Event Extraction and Coalescing. Topaz helps users build the event alphabet by extracting the set of unique combined values of a user-specified list of signals. These values can then be coalesced into coarser (i.e., more abstract) events by the user. If the user-specified alphabet set Σ is not a partition over $2^{\mathbf{AP}}$, Topaz *completes* Σ by adding an event **`NONE`** that is complementary to all user-defined events.

Trace Profiling. Simulated designs typically exhibit phase transitions, as shown in the spectrograms in Fig. 3 produced by Topaz for `Amber` [1], an open-source design studied later here. Moreover, given an alphabet set Σ, a logic simulation trace T can be irrelevant for specification mining with Σ. In Fig. 2, Topaz profiled 66 VCD files of Amber testcases over 5 alphabet sets. Relevant events are those specified by each profiled alphabet (i.e., all events other than **`NONE`**).

Trace Slicing. Trace slicing extracts the event traces from raw logic simulation traces and divides each trace into multiple *trace slices* for subsequent MSA. It is crucial to select slice boundaries properly so as to ensure that slices contain complete episodes of the behaviors of interest.

Sequence Alignment. Most automata-learning algorithms assume the existence of a means to *reset* the automaton being learned to a known start state. However, these techniques are only meaningful for *online* learning of automata. In this paper, only *offline* learning using passive observation of simulation traces is used. To obviate the use of resets, we now show that MSA, which revolutionized biological sequence analysis [5], holds the answer to the *initial-state uncertainty problem* [16]

TABLE I: Classification of specification mining tools.

Tool		Analysis			Target Formalism			Prior Info.	Requires Source Code[b]	Statistical Metric	Mining Techniques	Limitations
		Static	Dynamic	Hybrid	FSMs	Assertions	Rules					
Topaz		✖	✔	✖	✔	✖	✖	Interface signals	No	Statistical significance	MSA	(Only safety)[a]
Dianosis	[15]	✖	✔	✖	✖	✔	✖	OVL templates	No	None	N.A.	I[a]
Chang et al.	[3]	✖	✔	✖	✖	✖	✔	Mining window size	No	Support/Confidence	Sequential data mining	I, II[a]
Mandouh et al.	[6]	✖	✖	✔	✖	✔	✖	Mining window size	Yes	Support/Confidence	Sequential data mining	I, II[a]
GoldMine	[17]	✖	✖	✔	✖	✔	✖	Mining window size	Yes	Support/Confidence	Decision Tree Learning	II[a]
IODINE	[8]	✖	✔	✖	✔	✔	✖	Templates/Analyzers	No	Confidence	Template-specific analysis	I[a]
PropGen	[7]	✖	✔	✖	✖	✔	✖	Mining window size	Yes	None	Bit-vector pattern search	I, III[a]
Inferno	[10]	✖	✔	✖	✔	✖	✖	Interface signals	No	None	Transact. boundary search	III[a]
Li et al.	[12]	✖	✔	✖	✖	✔	✖	Property templates	Only hierarchy	Frequency, variance	Pattern matching	I[a]

[a](I) Limited expressiveness. (II) Finite time span. (III) Lack of state-space abstraction. [b]If yes, it cannot be used in reverse engineering.

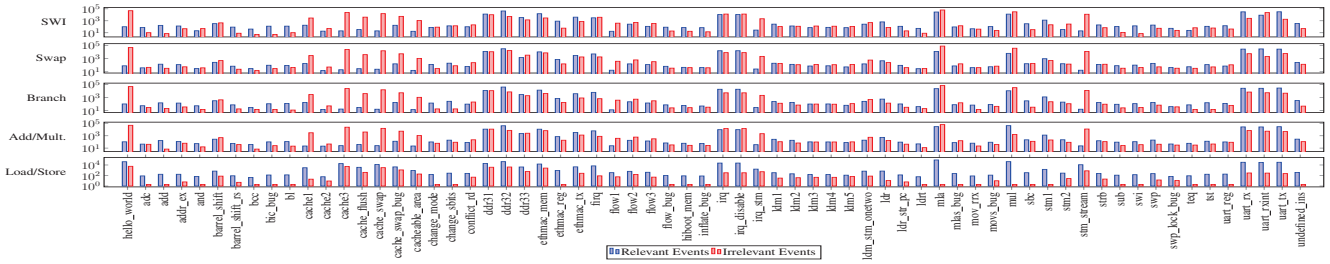

Fig. 2: Testcase alphabet profiles. The x-axis is a set of 66 testcases from the Amber test suite. The y-axis is a set of 5 alphabet sets used in Section V.

in offline learning contexts. Moreover, MSA enables modeling the observed system behavior with a NFA $\mathcal{A}$ whose abstract state space Q is not set in advance. In a MSA, as shown in Fig. 4, two or more traces are arranged as rows of a 2-d matrix so that *identical* logic events common to one or more traces are aligned in the same column. Due to the intrinsic variability of design behavior, different traces may not be perfectly aligned. Non-alignable positions are filled with *gaps*. It is prohibited to align different symbols (i.e., $\sigma_1, \sigma_2 \in \Sigma$ with $\sigma_1 \neq \sigma_2$) in the same column. For every *observation sequence* $\boldsymbol{\sigma} = (\sigma_0, \sigma_1, \sigma_2, \ldots) \in \Sigma^\omega$ of logic events, there is an implicit *unknown* labeling $\mathscr{L} : \mathbb{N} \to S \times Q$ of each event with a *hidden state* of transition system $TS' = TS \bigotimes \mathcal{A}$. Conceptually, a labeling $\mathscr{L} = (s_0, q_1), (s_1, q_2), \ldots$ represents two *synchronous* and *parallel* runs $s_0, s_1, \ldots$ and $q_1, q_2, \ldots$ of TS and $\mathcal{A}$, respectively, where there is $q_0 \in Q_0$ such that $\forall i \geq 0 : \sigma_i = L(s_i)$, $q_0 \xrightarrow{\sigma_0} q_1 \xrightarrow{\sigma_1} q_2 \ldots$ and $s_0 \to s_1 \to s_2 \ldots$. Given n observation sequences $\{\boldsymbol{\sigma}^1, \ldots, \boldsymbol{\sigma}^N\}$ and a corresponding set of hidden-state labelings $\{\mathscr{L}_1, \ldots, \mathscr{L}_N\}$, we can impose an alignment on these sequences, where any two events σ_i^m and σ_j^n from $\boldsymbol{\sigma}^m$ and $\boldsymbol{\sigma}^n$, respectively, can be aligned only if $\mathscr{L}_m(\sigma_i^m) = \mathscr{L}_n(\sigma_j^n)$. Conversely, hidden-state labelings can be recovered if an alignment of the observation sequences can be established, which can then be used to reconstruct an abstract version of $TS \bigotimes \mathcal{A}$. However, there is not a unique hidden-state labeling for every observation sequence due to nondeterminism of both TS and $\mathcal{A}$. By using enough simulation traces, a faithful image of all nondeterministic choices in $TS \bigotimes \mathcal{A}$ can be reconstructed. The details of MSA are largely standard [5] and are omitted for space constraints.

Graph Representation of MSAs. We now construct a probabilistic finite-state automaton (PFSA) [4], [14] $\mathcal{G}_\mathbf{m} = (V, \Sigma, p)$, as shown in Fig. 5, where V is a nonempty set of states, Σ is the alphabet set, and $p : V \times \Sigma \times V \to [0, 1]$ is the transition probability function such that for all $v \in V$, we have $\sum_{v' \in V} \sum_{\sigma \in \Sigma} p(v, \sigma, v') = 1$. The PFSA $\mathcal{G}_\mathbf{m}$ is a directed acyclic graph (DAG) constructed by noting that all identical letters $\sigma \in \Sigma$ in the same column $\mathbf{m}_i$ of $\mathbf{m}$ stand for a single *unknown* hidden state (s, q) of $TS \bigotimes \mathcal{A}$. We use a unique pair of labels (s, q) for every column of $\mathbf{m}$. Later, we will merge PFSA states that look sufficiently similar. A PFSA state $v \in V$ is connected by a directed edge to another state v' if, for at least one of the aligned trace slices, a letter in column $\mathbf{m}_v$ directly follows a letter in column $\mathbf{m}_{v'}$, possibly with intervening gap symbols only. An edge in $\mathcal{G}_\mathbf{m}$ is annotated with transition probability according to how many trace slices follow that edge in $\mathbf{m}$. The label $\sigma \in \Sigma$ of a PFSA edge $(u, \sigma, v) \in V \times \Sigma \times V$ in $\mathcal{G}_\mathbf{m}$ can be inferred by reversing Eq. 2:

$$\forall s, t \in S, \forall p, q \in Q : (s, p) \to' (t, q) \Rightarrow p \xrightarrow{L(t)} q$$

So every PFSA edge is labeled with the alphabet symbol in the MSA column associated with its sink PFSA state.

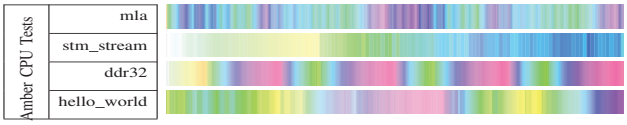

Fig. 3: Examples of distinct design operation phases vs. cycle time as captured by the largest two principal components (displayed as color hue and saturation, resp.) of the feature-vector histogram of the **design bit-vector** (DBV) within a moving window of size 100 clock cycles. The DBV combines values of all logic signals in one giant bit vector. DBV features are counts of (overlapping) binary strings from "`0`" to "`111`".

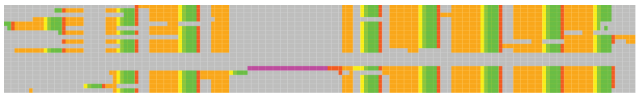

Fig. 4: A section of an example MSA of 20 trace slices from the Amber `stm_stream` testcase. Gaps are gray and every logic event is depicted by a different color.

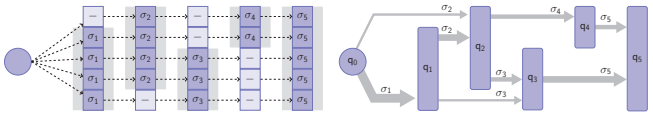

Fig. 5: A MSA viewed as a PFSA. Edge line width is proportional to transition probability.

PFSA Reduction Until now, the PFSA constructed from a MSA is acyclic and, hence, cannot generalize beyond the training set of traces. Topaz uses the sk-strings method [14] to construct an *over-approximation* $\mathcal{G}'$ of a PFSA $\mathcal{G}$. The degree of over-approximation is controlled by a probability parameter $0 < P \leq 1$ and a depth parameter $K > 0$.

Determinization. After a relatively small PFSA has been mined, Topaz users can make more informed decisions on one or more initial states. Topaz then applies power-set construction [9] to a mined NFA to obtain the corresponding *complete* DFA. A failure state is added to the constructed DFA and a *failure edge* is extended to it from every other DFA state, labeled with an event (i.e., a Boolean formula) that is complementary to all other edges of its source state.

Abstraction. To allow users to trade off precision of a mined DFA $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ for more conciseness, equivalence relations over Q with varying granularities are needed. Given an equivalence relation $\sim \subseteq Q \times Q$, the state space Q can be reduced by taking the quotient $\mathcal{A}/\sim$. A first step toward an equivalence relation is a *simulation relation* [2]. A simulation preorder $\preceq \subseteq Q \times Q$ is a relation such that for all $(p, q) \in \preceq$ and $\sigma \in \Sigma$, if $p \xrightarrow{\sigma} p'$, then there is $q' \in Q$ such that $q \xrightarrow{\sigma} q'$ and $(p', q') \in \preceq$. Topaz uses algorithms that compute simulation preorders in time $\mathcal{O}(|\delta|.|Q|)$ [2]. A simulation relation $\preceq$ over Q can give rise to equivalence relations, such as the symmetric closure $\sim = \preceq \cup \preceq^{-1}$ and the symmetric kernel $\cong = \preceq \cap \preceq^{-1}$. Between these two extremes, it is left to Topaz users to identify state equivalences that preserve classes of properties important to them.

Statistical Significance. To estimate the statistical significance (the *p-value*) for a given MSA $\mathbf{m}$, Topaz generates random sequences from the sequences of $\mathbf{m}$ by randomly shuffling order of their events [11]. The p-value for a given alignment with score X_0 is calculated as $p = M/N$, where N is the number of permutations aligned and scored, and M is the number of those experiments scoring $\geq X_0$.

V. Amber Case Study

This section details one case study of Topaz on a fairly rich CPU design, Amber [1], which is an open-source, five-stage, 32-bit RISC processor core implementing ARMv2a instruction-set architecture (ISA).

Target Signal Groups. Given a digital logic design TS, the set $\mathbf{AP}$ of logic signals at the focus of specification mining and verification can be selected in many different ways. For each set Sim of logic simulation traces, Topaz produces a DFA for each set $\mathbf{AP}$ of logic signals. We select sets of signals that span multiple pipeline stages and, hence, their temporal relations may span many clock cycles. The organizing principle here is that for each ARMv2a instruction, there is a set of signals that may exhibit some activity during the processing of that instruction in the pipeline. That activity typically depends on preceding and succeeding instructions (e.g., branches, back-to-back loads or stores), the cache state, etc. This is a treasure trove for specification miners.

TABLE II: Amber specification mining parameters and results for 5 different alphabet sets associated with 5 instruction types.

Alphabet Sets ►	Load/Store	Add/Mult.	Branch	Swap	SWI
Alphabet size	18	45	17	30	34
Modeled Signals	10	28	15	23	20
K	4	4	4	4	4
P	0.5	0.5	0.5	0.5	0.5
Trace Slices	225	152	14	158	100
Slice Size	200	100	200	200	200
Alignment time (min)	< 1	< 1	< 1	< 1	< 1
p-value	0%	0%	0%	0%	0%
DFA States	57	172	127	84	137
Sim-preordered pairs	763	277	850	934	281

Extracted Properties. Table II shows the inputs and parameters used by Topaz in 5 mining sessions with 5 different alphabet sets corresponding to 5 instruction families. It also shows some descriptors of the mining outcomes. The number of DFA states reported in each case is before any simulation equivalence is applied. Moreover, the number of simulation-preordered pairs of states is an indication of the room for abstraction present in a mined DFA.

Topaz Mining Run-time. Topaz, as well as RTL simulations, were run on a quad-core Intel i5, 2.5GHz CPU with 8GB of RAM and 64-bit operating systems (Windows for Topaz, and Centos-6 for RTL simulations). The most time-consuming stage in Topaz mining flow is MSA. In all 5 cases shown in Table II, it completed in less than a minute.

SystemVerilog Checkers. For every extracted DFA, Topaz automatically generates a synthesizable SystemVerilog checker module having one output set to logic `1` once a minimal bad prefix has been observed and stays high forever. These checker modules can be bound to design module instances.

References

[1] OpenCores. http://opencores.org/. Accessed: 2015-09-18.
[2] C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.
[3] P.-H. Chang and L. C. Wang. Automatic assertion extraction via sequential data mining of simulation traces. ASP-DAC, 2010.
[4] C. de la Higuera. Learning finite state machines. FSMNLP, 2010.
[5] R. Durbin. *Biological sequence analysis: probabilistic models of proteins and nucleic acids*. Cambridge university press, 1998.
[6] E. El Mandouh and A. G. Wassal. Automatic generation of hardware design properties from simulation traces. ISCAS, 2012.
[7] G. Fey and R. Drechsler. Improving simulation-based verification by means of formal methods. ASP-DAC, 2004.
[8] S. Hangal, N. Chandra, S. Narayanan, and S. Chakravorty. IODINE: A Tool to Automatically Infer Dynamic Invariants for Hardware Designs. DAC, 2005.
[9] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2006.
[10] B. Isaksen and V. Bertacco. Verification through the principle of least astonishment. ICCAD, 2006.
[11] D. Kandel, Y. Matias, R. Unger, and P. Winkler. Shuffling biological sequences. *Discrete Appl. Math.*, 71(1-3):171–185, Dec. 1996.
[12] W. Li, A. Forin, and S. A. Seshia. Scalable specification mining for verification and diagnosis. DAC, 2010.
[13] D. Lo, S.-C. Khoo, J. Han, and C. Liu. *Mining Software Specifications: Methodologies and Applications*. CRC Press, 2011.
[14] A. Raman, J. Patrick, and P. North. The sk-strings method for inferring PFSA. ICML, 1997.
[15] F. Rogin, T. Klotz, G. Fey, R. Drechsler, and S. Rülke. Automatic generation of complex properties for hardware designs. DATE, 2008.
[16] S. Sandberg. Homing and synchronizing sequences. In M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors, *Model-Based Testing of Reactive Systems*, 2004.
[17] S. Vasudevan, D. Sheridan, S. Patel, D. Tcheng, B. Tuohy, and D. Johnson. Goldmine: Automatic assertion generation using data mining and static analysis. DATE, 2010.
[18] A. Zeller. Mining Specifications: A Roadmap. In S. Nanz, editor, *The Future of Software Engineering*, pages 173–182. Springer, 2011.