MNRL and MNCaRT

An Open-Source, Multi-Architecture State Machine Research and Execution Ecosystem

Technical Report TR# CS-2017-01

Kevin Angstadt Jack Wadden Westley Weimer Kevin Skadron
Department of Computer Science
University of Virginia
Charlottesville, Virginia 22904
{angstadt,wadden,weimer,skadron}@virginia.edu

Abstract

We present MNRL, an open-source, general-purpose and extensible state machine representation language. The representation is flexible enough to support traditional finite automata (NFAs, DFAs) while also supporting more complex machines, such as those which propagate multi-bit signals between processing elements. The specification is based on JSON, a data interchange format that is supported across general-purpose programming languages. We also provide Python and C++ APIs for direct reading and writing of MNRL files.

We also discuss MNCaRT, the software ecosystem built around MNRL. MNCaRT is an umbrella repository of previously-published tools, which have been adapted to support MNRL, as well as new tools, which were specifically designed for MNRL. Tool support includes manipulation of MNRL files, execution of complex machines, high-speed processing of simplified MNRL files, and compilation of regular expressions to MNRL. We support the execution of MNRL networks on CPUs (with VASim and Intel Hyperscan), GPUs (with a custom DFA engine), and FPGAs (with a MNRL to HDL translator). As with MNRL, all of the tools in MNCaRT are open-source, allowing for development and use in both academia and industry.

CCS Concepts • **Software and its engineering → Specialized application languages**; **Software libraries and repositories**; • **Theory of computation →** *Automata extensions*;

Keywords automata processing, state machine, language design, software tools

Department of Computer Science
University of Virginia
Technical Report TR# CS-2017-01, May 2017

1 Introduction

Years of research and tools development have resulted in high-throughput *automata processing* architectures and software engines [9, 11, 13, 16, 26, 28, 34, 38]. These developments have lead to the discovery of new, non-obvious use-cases and application domains for finite automata, such as natural language processing [41], network security [24], graph analytics [23], high-energy physics [37], bioinformatics [21, 22, 30], pseudo-random number generation and simulation [31], data-mining [35, 36], and machine learning [29].

Unfortunately, the software frameworks for the construction, manipulation, and translation of finite automata are frustratingly fractured and restrictively licensed. For example, Micron licenses a comprehensive, closed-source software development kit that specifically targets their D480 Automata Processor [17]. While these tools are useful for developing applications for the AP, the tools do not allow a researcher to easily evaluate their designs across hardware platforms, such as CPUs, GPUs, and FPGAs. Additionally, the tools are closed-source and therefore cannot be easily extended to support new architectures and automata paradigms. Instead, a general and extensible framework is needed to enable the development of platform-independent applications and to support experimental automata designs.

We therefore present MNRL, the MNRL Network Representation Language (pronounced "mineral"), a JSON-based, open-source language and associated Python and C++ APIs to support the development of, and experimentation with, new automata processing-based applications and architectures. MNRL allows a user to define a *network* (or collection) of MNRL *nodes*, which represent the states within finite automata. Each node stores configuration information (such as node type, name, etc.) as well as connections to other nodes within the network. Note that the language specification is general, allowing state machines other than finite automata to also be represented in the language. We provide initial definitions for traditional finite automata states, homogeneous states, up-counters, and Boolean logic in the MNRL specification; additional node types may be defined by the user for specific applications. We provide a simplified specification, NFAMNRL, for representation of pure DFAs and NFAs.

Additionally, we have developed a suite of tools for creating, manipulating, and executing MNRL files, which we refer to as MNCaRT (the MNRL Network Computation and Research Testbed). MNCaRT collects a diverse set of automata processing tools and algorithms into a central location. As new projects are contributed to the MNRL ecosystem, they will become available as part of MNCaRT. We currently provide support for compiling state machines from Perl compatible regular expressions (PCRE) patterns [19] to MNRL, high-speed execution of NFAs and DFAs written in MNRL using Intel Hyperscan [13], and optimization and simulation of experimental automata designs with the Virtual Automata Simulator (VASim) [34]. Further, we provide back-ends for executing DFAs on GPUs [32], FPGAs [34], and exploring routing constraints for experimental spatial architectures via the Automata-to-Routing (ATR) tool [33].

The language specification and all supporting tools are publicly available (typically under a BSD 3-clause license), allowing both academics and industry experts to contribute to, and use, the codebases.

This work makes the following technical contributions:

- MNRL, an open-source (BSD 3-clause) and extensible JSON specification for representing state machines as well as a simplified JSON specification for DFAs and NFAs (nfaMNRL).
- Python and C++ APIs for reading, creating, manipulating, and writing MNRL files.
- Extensions to Intel's Hyperscan regular expression processing engine to support the compilation and execution of nfaMNRL files.
- Extensions to Hyperscan supporting the compilation of PCRE to nfaMNRL files.
- An extended version of VASim, which supports reading and writing of MNRL files.

2 Background and Related Work

A finite automaton includes of a set of states and a set of transitions defining how the states become active based on symbols observed in an input stream [25]. In a non-deterministic finite automaton (NFA), it is possible to transition to multiple states on the same input symbol. These are often represented as a graph, defining the topological layout of the computation. By providing this topological specification, computation is decoupled from the definition of the state machine. This allows for a common execution engine, which defines the execution model, to process arbitrary automata, improving code reuse and reducing sources for bugs. Automata can also be represented as a set of regular expressions, which define the search pattern the automata recognize.

In the remainder of this section, we briefly describe current automata representation languages and discuss their limitations.

2.1 Automata Network Markup Language (ANML)

ANML is a proprietary automata description language. This representation defines a special type of homogeneous automata [7],[1] and is unable to describe arbitrary finite automata networks. ANML has specific tags for each hardware element on the Automata Processor, but the schema does not allow for additional kinds of elements to be prototyped. Further, additional annotations cannot be added to elements in ANML while maintaining support for current tools. Although ANML could be easily extended to support arbitrary automata, its use is licensed and controlled by Micron Technology. Therefore it is not a good choice for an open automata language.

2.2 Becchi NFA Serialization Format

The tool chain associated with Becchi et al.'s publications on accelerating finite automata processing uses a simple representation of NFAs [2]. The representation is based on the theoretical definition of an NFA and therefore cannot be easily extended to support more complex state machines. Additionally, the language is custom, and there is no support in general-purpose programming languages for reading and manipulating these files.

2.3 JFlap

JFlap [20] is a software package for experimenting with formal languages and has an undocumented file format. The language, however, is not formally specified and is targeted for theoretical research. Therefore, state machine descriptions are limited to theoretical models and do not support architecture-oriented descriptions.

2.4 Dot

Dot [10] is the graph language associated with the GraphViz graph visualization software. Dot is more general than a state machine language, but the language is more targeted to defining visual layouts than serializing state machines. Additionally, Dot lacks sufficient parsers, and emitters for this language are often custom for each application.

2.5 Regular Expressions

Regular expressions are another common option for representing a search pattern; however, these also suffer from maintainability challenges. For many applications that can benefit from automata-based algorithms, such as particle tracking [37], motif searches [22], and rule mining [35, 36], the regular expression representing the search is non-intuitive and may simply be an exhaustive enumeration of all possible strings that should be matched. Additionally, programming of regular expressions can be extremely error-prone due to

[1] In a homogeneous NFA, all incoming transitions to any given state *must* occur on the same input character.

variations in regular expression syntax, which leads to high rates of runtime exceptions [27].

3 MNRL: A New Automata Language

We have developed an open-source and extensible automata representation language called MNRL. This language allows for the topological specification of a collection of finite state machines using JSON syntax [15]. JSON is programming language-independent, but is supported by most common general-purpose programming languages. Therefore, the format of MNRL allows for easy use in projects regardless of the codebase's primary language.

It is important to note that the MNRL format specifies the layout of a machine but does not specify how elements behave, allowing for different "styles" of state machines to be represented. This includes traditional NFAs [25] and homogeneous NFAs [7].[2] The behavior of elements is left for the execution engine to specify and implement (and allows for MNRL to be an extremely flexible file format). Therefore, MNRL is similar in intent to the Unified Modeling Language (UML), which allows developers to describe and design software systems while eliding implementation details [12].

In the remainder of this section, we provide an overview of the MNRL language, describe a restricted NFAMNRL schema, and present means for extending and using the language.

3.1 MNRL Format

A MNRL file is a representation of a single MNRL *network*—a collection of one or more state machines that are executed in parallel using the same input. The file contains an array of MNRL nodes, which define each element in the network. A node consists of:

- A unique identifier
- A node type (state, homogeneous state, up counter, boolean, etc.)
- How the node is enabled
- Whether the node reports (generates an output signal) when activated
- An array of input ports
  - Each input port has a unique identifier and a specified width (number of wires)
- An array of output ports
  - Each output port has a unique identifier and a specified width (number of wires)
  - Each output port specifies an array of connected elements
- A collection of custom attributes, specific to each element type

With this information, a developer is able to specify the topological layout of the state machines within the network and to specify the sort of behavior the underlying execution engine should assign to each node. The implementation of behavior is *not* defined in the MNRL file; instead, the computation engine that processes a MNRL network is responsible for specifying the semantics for each node type. Therefore, node types and execution engines are typically co-designed. If the environments needs information (e.g. symbol sets for matching against an input stream), this additional configuration can be embedded in a MNRL node's attributes. For the standard node types, we have specified additional attributes to support their respective expected behaviors.

We provide the specification of MNRL as a JSON schema [14], which allows for validation of file syntax. The MNRL schema defines four node types: standard automata states (*state*), homogeneous automata states (*hState*), saturating up-counters (*upCounter*), and combinatorial logic (*boolean*). Custom attributes for each of these node types are described in Table 1. Each of these node types also defines a *reportId* attribute,

```
1  {
2    "id": "0t_15l_5r",
3    "type": "hState",
4    "enable": "onActivateIn",
5    "report": true,
6    "inputDefs": [
7      {
8        "width": 1,
9        "portId": "i"
10     }
11   ],
12   "outputDefs": [
13     {
14       "width": 1,
15       "activate": [],
16       "portId": "o"
17     }
18   ],
19
20   "attributes": {
21     "reportId": 5,
22     "latched": false,
23     "symbolSet": "[\\xFF]"
24   }
25 }
```

Figure 1. Sample MNRL homogeneous hState Node. The node is enabled (performs computation) only after an incoming edge is active (line 4), and this node matches against the input character \xFF (line 23). When this occurs, the node generates a report signal (line 5). Lines 6-11 define a single input port for incoming edges. Lines 12-18 define a single output port for outgoing edges. The array on line 15 is empty, indicating that there are no outgoing edges.

[2] In fact, MNRL is general enough to represent machines that are more powerful than finite automata (e.g. push-down automata, cellular automata, and Turing machines).

Table 1. Custom Attributes for MNRL Node Types

Node Type	Attribute	Required?	Attribute Type	Description
state	symbolSet	YES	object	Mapping from each output port name to a symbol set string representing the matched character set that enables the outgoing connections from the given port
	latched	NO	Boolean	Determines whether a state remains enabled after the first enable signal
hState	symbolSet	YES	string	Represents the matched character set that enables the outgoing connections
	latched	NO	Boolean	Determines whether a state remains enabled after the first enable signal
upCounter	threshold	YES	number	The internal value at which the counter enables outgoing connections
	mode	YES	enum	"trigger": enable the outgoing connections for one clock cycle when the threshold is reached "high": enable the outgoing connections for all subsequent clock cycles while the internal value if at the threshold "rollover": similar to trigger, but also reset the internal value
boolean	gateType	YES	enum	Must be one of the following values: 'and', 'or', 'nor', 'not', or 'nand'

Table 2. Modes for Enabling MNRL Nodes

Enable Mode	Description
always	The node is enabled on every cycle
onActivateIn	The node is enabled on the clock cycle following a high signal to an input port
onStartAndActivateIn	The node is enabled on the first clock cycle and then follows the "onActivateIn" mode
onLast	The node is only enabled for the final clock cycle

which allows an additional string or integer to be associated and returned with any reporting event during execution. MNRL states and hStates map directly to notions of NFA states and homogeneous NFA states. We provide upCounter and boolean node types to maintain compatibility with Micron's D480 Automata Processor [9]; however these element types are general and similar elements have seen use in other engines and automata styles [5, 8, 18, 39].

Additionally, the MNRL schema defines four valid modes for *enabling* a node. These modes are described in Table 2. An enabled node performs a predefined computation on a given clock cycle. These modes are the same as those used in common state machine engines, such as the AP [9] and Intel Hyperscan [13].

3.2 NFAMNRL: A Simplified Specification

We also provide a minimal specification, NFAMNRL, which only supports NFA states and hStates. This specification may be used for applications that wish to only support traditional finite automata. NFAMNRL is valid with respect to the full MNRL specification. That is, any MNRL file that validates against the NFAMNRL spec will also validate against the more general MNRL spec.

3.3 Extending the MNRL Schema

MNRL is designed to be extensible, enabling research on new, custom automata and state machine functionality. To support this, the MNRL schema is organized so that researchers can quickly define custom attributes for new node types. Because custom node types become part of the JSON schema, prototype extensions to the MNRL format can still be statically checked with minimal effort from the developer.[3] The MNRL file format could easily be extended to support additional node types such as non-deterministic counters [5], jump states [40], and stacks (to support push-down automata). Because MNRL supports variable-width ports, it is also possible to represent elements that share more than a single bit of data with elements downstream.

3.4 MNRL APIs in Python and C++

While common general-purpose programming languages support the manipulation of MNRL files directly through

[3] One of the authors extended the schema to support floating point comparisons. Total time to add support into both the Python and C++ APIs was approximately two hours.

JSON libraries, we provide open-source programmatic APIs in Python and C++ for MNRL. These APIs perform additional validation checks on MNRL files for constraints that cannot be captured by a JSON schema. For example, a homogeneous hState can only have one input and one output port. Supported functionality includes: reading MNRL files, writing MNRL files, creating MNRL networks programmatically, and manipulating MNRL networks and nodes programmatically.

4 MNCaRT: Automata Analysis, Execution, and Transformation

Our goal with the MNRL language is to enable the development of a rich and vibrant ecosystem of compatible tools for manipulating and executing state machines (especially finite automata). We are collecting these tools in an umbrella repository, the MNRL Network Computation and Research Testbed (or MNCaRT, pronounced "minecart"). As new tools are developed, they will be added to MNCaRT. By keeping tools catalogued in a single location, we hope to maintain the interoperability of tools and reduce fracturing in the ecosystem.

Figure 2 describes the interaction between tools provided with MNCaRT. Our ecosystem supports workflows beginning with high-level languages, such as PCRE, and ending with execution on CPUs, GPUs, and FPGAs. Through conversion to ANML, we also support execution on Micron's D480 Automata Processor. In this section, we briefly describe to tools that make up the initial release of MNCaRT.

4.1 RAPID

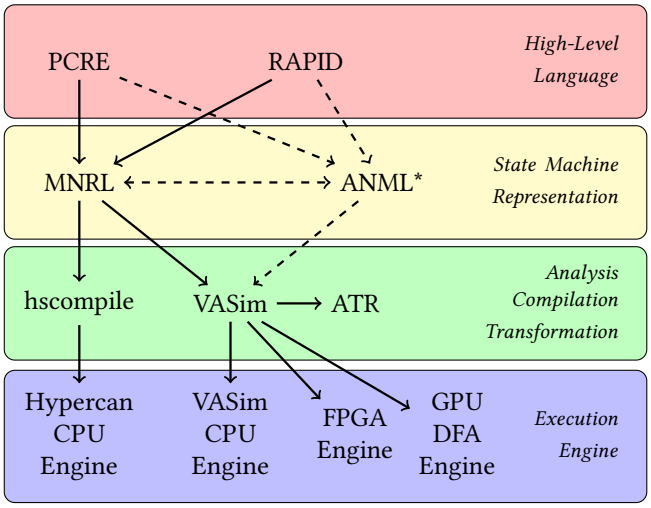
RAPID is a high-level programming language and combined imperative and declarative model for execution of sequential pattern-matching applications [1]. This C-like language is extended with three keywords to support parallel matching of patterns against a single data stream as well as sliding window pattern recognition. We have extended the RAPID compiler to emit MNRL files, allowing for high-level programming within the MNCaRT ecosystem.

4.2 Hyperscan-Based Tools

Hyperscan [13] is an open-source, high-performance regular expression processing library supported by Intel. We utilize this tool's PCRE parsing algorithms to provide a regular expression to NFAMNRL compiler. Additionally, we have extended Hyperscan to support the compilation and execution of NFAMNRL files.

pcre2mnrl. Our regular expression compiler reads in a file of regular expressions separated by newlines and compiles the given set of patterns to a single NFAMNRL file. The line number of each given PCRE pattern is used as the report ID to allow for easy identification of matched patterns in processing output.

MNCaRT Ecosystem



While ANML is not officially part of MNCaRT, we indicate where this alternate representation falls within the MNCaRT ecosystem.

Figure 2. Tools supplied as part of MNCaRT. These fall into three categories: front-end representations (both high-level and representation languages), transformation and compilation tools, and hardware and software execution engines.

hscompile. We provide an extension to Hyperscan that parses NFAMNRL files and compiles the finite automata to a serialized Hyperscan pattern database, allowing offline compilation. Additionally, our tool serializes a mapping from MNRL node IDs and report IDs to Hyperscan's internal naming for each state machine element. This mapping enables human-readable output when processing input data using Hyperscan.

hsrun. Finally, we provide a tool for processing NFAMNRL files against an input stream using the Hyperscan execution core. This tool deserializes the Hyperscan pattern database and node mapping produced by *hscompile*. The tool then scans the given input file against the pattern database and prints out human-readable reporting information containing the MNRL node ID, report ID, and offset in the input stream. We also provide an output format that prints the total number of reports for each reporting node. If multiple compiled NFAMNRL files and/or input files are passed to *hsrun*, the tool will execute all pairings of the files using a supplied number of threads.

4.3 VASim

We have extended VASim [34] to support parsing of MNRL files. VASim is, to the best of our knowledge, the first extensible, general-purpose framework that combines automata simulation, optimization, transformation, and performance modeling into one unified and open source code base. This

framework enables easy prototyping, debugging, simulation, and analysis of automata-based applications and architectures. Additionally, VASim can parse Micron ANML files, allowing for conversion between this proprietary format and MNRL.

The platform can support simulation and analysis of a diverse set of finite automata models, such as classical NFAs, AP-style NFAs, JFAs [40], counting finite automata [5], and hybrid approaches [3].

VASim also provides a common codebase for applying state-of-the-art optimizations, transformations, and static and dynamic analyses to finite automata. This platform allows researchers to easily and quickly share new algorithms, and perform fair apples-to-apples comparisons to prior work, accelerating automata-processing research. We provide several optimizations in the core of VASim, including common prefix merging [4] and a literal matching engine [13].

4.4 Automata-to-Routing

We extend the Automata-to-Routing (ATR) [33] framework to support placement and routing of MNRL state machines. Automata-to-routing utilizes the Versatile Place and Route (VPR) tool to model hypothetical spatial automata-processing architectures [6]. VPR is so flexible that it is capable of modeling any spatial architecture that has a well-defined set of spatial processing elements. We thus extend VASim to emit VPR-readable circuits of MNRL networks. We also provide guidance to construct custom, parameterizable, spatial architecture description files to accept these custom state machine circuits. ATR is thus capable of modeling spatial architectures that are purpose-built to accept MNRL state machines.

5 Conclusions

MNRL is a general and extensible format for representing state machines. The language specification and associated tools are released with open-source licenses to promote collaboration and usage within both academia and industry. MNRL is supported by general-purpose programming languages because it is based off of the JSON format. Further, we provide MNRL-specific APIs for Python and C++ to perform more direct manipulation and validation of networks.

Additionally, we provide MNCaRT, a suite of tools for analyzing, executing, and transforming MNRL networks. We support execution of MNRL networks on CPUs, GPUs, and FPGAs, and we provide a workflow for execution on Micron's AP. Support for high-level pattern-matching languages, such as PCRE and RAPID is also provided as part of MNCaRT. Finally, we allow for design space exploration through analysis functionality in the VASim and ATR tools.

We hope that this new state machine language and accompanying software ecosystem will stimulate new efforts to develop efficient and specialized automata processing applications.

Acknowledgments

This work was supported in part by grants from the NSF (CCF-1116673, CCF-1629450, CCF-1629450, CCF-1619123, CNS-1619098), the Jefferson Scholars Foundation, the Achievement Rewards for College Scientists (ARCS) Foundation, a grant from Xilinx, and support from C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

Acknowledgment Of Support And Disclaimer: (a) Contractor acknowledges Government's support in the publication of this paper. This material is based upon work funded by AFRL, under AFRL Contract No. FA8750-15-2-0075. (b) Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of AFRL.

References

[1] Kevin Angstadt, Westley Weimer, and Kevin Skadron. 2016. RAPID Programming of Pattern-Recognition Processors. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. 593–605. DOI: http://dx.doi.org/10.1145/2872362.2872393

[2] Michela Becchi. 2011. Retular Expression Processor. http://regex.wustl.edu. (2011). Accessed 2017-04-06.

[3] Michela Becchi and Patrick Crowley. 2007. A Hybrid Finite Automaton for Practical Deep Packet Inspection. In *Proceedings of the ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT '07)*. Article 1, 12 pages. DOI: http://dx.doi.org/10.1145/1364654.1364656

[4] Michela Becchi and Patrick Crowley. 2008. Efficient Regular Expression Evaluation: Theory to Practice. In *Proceedings of Architectures for Networking and Communications Systems (ANCS '08)*. 50–59. DOI: http://dx.doi.org/10.1145/1477942.1477950

[5] Michela Becchi and Patrick Crowley. 2008. Extending Finite Automata to Efficiently Match Perl-compatible Regular Expressions. In *Proceedings of the ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT '08)*. Article 25, 12 pages. DOI: http://dx.doi.org/10.1145/1544012.1544037

[6] Vaughn Betz and Jonathan Rose. 1997. VPR: A new packing, placement and routing tool for FPGA research. In *Proceedings of the International Workshop on Field Programmable Logic and Applications*. Springer, 213–222.

[7] Pascal Caron and Djelloul Ziadi. 2000. Characterization of Glushkov automata. *Theoretical Computer Science* 233, 1 (2000), 75–90.

[8] A. K. Chandra and L. J. Stockmeyer. 1976. Alternation. In *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*. 98–108. DOI: http://dx.doi.org/10.1109/SFCS.1976.4

[9] Paul Dlugosch, Dave Brown, Paul Glendenning, Michael Leventhal, and Harold Noyes. 2014. An Efficient and Scalable Semiconductor Architecture for Parallel Automata Processing. *IEEE Transactions on Parallel and Distributed Systems* 25, 12 (2014), 3088–3098. DOI: http://dx.doi.org/10.1109/TPDS.2014.8

[10] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen North, Gordon Woodhull, Short Description, and Lucent Technologies. 2001. Graphviz ÃŚ open source graph drawing tools. In *Lecture Notes in Computer Science*. Springer-Verlag, 483–484.

[11] Yuanwei Fang, Tung T Hoang, Michela Becchi, and Andrew A Chien. 2015. Fast support for unstructured data processing: the unified automata processor. In *Proceedings of the ACM International Symposium on Microarchitecture (Micro '15)*. 533–545.

[12] Martin Fowler. 2004. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley. https://books.google.com/books?id=nHZslSr1gJAC

[13] Intel. 2017. Hyperscan. https://01.org/hyperscan. (2017). Accessed 2017-04-07.

[14] Internet Engineering Task Force. 2013. *JSON Schema: core definitions and terminology*. Number json-schema-core. http://json-schema.org/latest/json-schema-core.html

[15] JSON 2013. *The JSON Data Interchange Format* (1st edition ed.). Technical Report Standard ECMA-404 1st Edition / October 2013. ECMA. http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

[16] Anil Krishna, Timothy Heil, Nicholas Lindberg, Farnaz Toussi, and Steven VanderWiel. 2012. Hardware acceleration in the IBM PowerEN processor: Architecture and performance. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*. 389–400.

[17] Micron Technology. 2016. The Automata Processor Software Development Kit. http://www.micronautomata.com. (2016). Accessed 2017-04-05.

[18] Marziyeh Nourian, Xiang Wang, Xiaodong Yu, Wu chun Feng, and Michela Becchi. 2017. Demystifying Automata Processing: GPUs, FPGAs, or Micron's AP?

[19] PCRE. 2017. Perl Compatible Regular Expressions. http://www.pcre.org. (2017). Accessed 2017-04-07.

[20] S.H. Rodger and T.W. Finley. 2006. *JFLAP: An Interactive Formal Languages and Automata Package*. Jones and Bartlett. https://books.google.com/books?id=494hTkZ_gu4C

[21] Indranil Roy. 2015. *Algorithmic Techniques for the Micron Automata Processor*. Ph.D. Dissertation. Georgia Institute of Technology.

[22] Indranil Roy and Srinivas Aluru. 2014. Finding Motifs in Biological Sequences Using the Micron Automata Processor. In *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium*. 415–424. DOI: http://dx.doi.org/10.1109/IPDPS.2014.51

[23] I. Roy, N. Jammula, and S. Aluru. 2016. Algorithmic Techniques for Solving Graph Problems on the Automata Processor. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS '16)*. 283–292. DOI: http://dx.doi.org/10.1109/IPDPS.2016.116

[24] I. Roy, A. Srivastava, M. Nourian, M. Becchi, and S. Aluru. 2016. High Performance Pattern Matching Using the Automata Processor. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS '16)*. 1123–1132. DOI: http://dx.doi.org/10.1109/IPDPS.2016.94

[25] Michael Sipser. 2006. *Introduction to the Theory of Computation*. Vol. 2. Thomson Course Technology.

[26] Ioannis Sourdis, João Bispo, João M. P. Cardoso, and Stamatis Vassiliadis. 2008. Regular Expression Matching in Reconfigurable Hardware. *Journal of Signal Processing Systems* 51, 1 (2008), 99–121. DOI: http://dx.doi.org/10.1007/s11265-007-0131-0

[27] Eric Spishak, Werner Dietl, and Michael D. Ernst. 2012. A Type System for Regular Expressions. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs (FTfJP '12)*. 20–26. DOI: http://dx.doi.org/10.1145/2318202.2318207

[28] Titan IC Systems. 2017. Helios RXPF Soft IP for FPGA Security Analytics Acceleration. http://titan-ic.com/products/helios-rxpf. (2017). Accessed 2017-04-05.

[29] Tommy Tracy, Yao Fu, Indranil Roy, Eric Jonas, and Paul Glendenning. 2016. Towards Machine Learning on the Automata Processor. In *Proceedings of ISC High Performance Computing*. 200–218. DOI: http://dx.doi.org/10.1007/978-3-319-41321-1_11

[30] Tommy Tracy II, Mircea Stan, Nathan Brunelle, Jack Wadden, Ke Wang, Kevin Skadron, and Gabe Robins. 2015. Nondeterministic Finite Automata in Hardware—the Case of the Levenshtein Automaton. *Architectures and Systems for Big Data (ASBD), in conjunction with ISCA* (2015).

[31] J. Wadden, N. Brunelle, K. Wang, M. El-Hadedy, G. Robins, M. Stan, and K. Skadron. 2016. Generating efficient and high-quality pseudo-random behavior on Automata Processors. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*. 622–629. DOI: http://dx.doi.org/10.1109/ICCD.2016.7753349

[32] J. Wadden, V. Dang, N. Brunelle, T. T. II, D. Guo, E. Sadredini, K. Wang, C. Bo, G. Robins, M. Stan, and K. Skadron. 2016. ANMLzoo: a benchmark suite for exploring bottlenecks in automata processing engines and architectures. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*. 1–12. DOI: http://dx.doi.org/10.1109/IISWC.2016.7581271

[33] Jack Wadden, Samira Khan, and Kevin Skadron. 2017. Automata-to-Routing: An Open Source Toolchain for Design-Space Exploration of Spatial Automata Processing Architectures. In *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*.

[34] Jack Wadden and Kevin Skadron. 2016. *VASim: An Open Virtual Automata Simulator for Automata Processing Application and Architecture Research*. Technical Report CS2016-03. University of Virginia.

[35] Ke Wang, Elaheh Sadredini, and Kevin Skadron. 2016. Sequential Pattern Mining with the Micron Automata Processor. In *Proceedings of the ACM International Conference on Computing Frontiers (CF '16)*. ACM, New York, NY, USA, 135–144. DOI: http://dx.doi.org/10.1145/2903150.2903172

[36] Ke Wang, Mircea Stan, and Kevin Skadron. 2015. Association Rule Mining with the Micron Automata Processor. In *Proceedings of the 29th IEEE International Parallel & Distributed Processing Symposium*. http://www.cap.virginia.edu/sites/cap.virginia.edu/files/kwang_arm_submitted.pdf

[37] Michael H.L.S. Wang, Gustavo Cancelo, Christopher Green, Deyuan Guo, Ke Wang, and Ted Zmuda. 2016. Using the Automata Processor for fast pattern recognition in high energy physics experiments - A proof of concept. *Nuclear Instruments and Methods in Physics Research* (2016). DOI: http://dx.doi.org/10.1016/j.nima.2016.06.119

[38] Xiang Wang. 2014. *Techniques for Efficient Regular Expression Matching Across Hardware Architectures*. Master's thesis. University of Missouri-Columbia. https://mospace.umsystem.edu/xmlui/bitstream/handle/10355/44433/research.pdf?sequence=1

[39] Pei-Chi Wu, Feng-Jian Wang, and Kai-Ru Young. 1992. Scanning Regular Languages by Dual Finite Automata. *SIGPLAN Not.* 27, 4 (April 1992), 12–16. DOI: http://dx.doi.org/10.1145/131080.131081

[40] X. Yu, B. Lin, and M. Becchi. 2014. Revisiting State Blow-Up: Automatically Building Augmented-FA While Preserving Functional Equivalence. *IEEE Journal on Selected Areas in Communications* 32, 10 (2014), 1822–1833. DOI: http://dx.doi.org/10.1109/JSAC.2014.2358840

[41] Keira Zhou, Jeffrey J. Fox, Ke Wang, Donald E. Brown, and Kevin Skadron. 2015. Brill tagging on the Micron Automata Processor. In *Proceedings of the 9th IEEE International Conference on Semantic Computing*. 236–239. DOI: http://dx.doi.org/10.1109/ICOSC.2015.7050812