MNCaRT: An Open-Source, Multi-Architecture Automata-Processing Research and Execution Ecosystem

Kevin Angstadt, Jack Wadden, Vinh Dang, Ted Xie, Dan Kramp, Westley Weimer, Mircea Stan, *Fellow, IEEE*
Kevin Skadron, *Fellow, IEEE*

Abstract—We present MNCaRT, a comprehensive software ecosystem for the study and use of automata processing across hardware platforms. Tool support includes manipulation of automata, execution of complex machines, high-speed processing of NFAs and DFAs, and compilation of regular expressions. We provide engines to execute automata on CPUs (with VASim and Intel Hyperscan), GPUs (with custom DFA and NFA engines), and FPGAs (with an HDL translator). We also introduce MNRL, an open-source, general-purpose and extensible state machine representation language developed to support MNCaRT. 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. We hope that our ecosystem and representation language stimulates new efforts to develop efficient and specialized automata processing applications.

1 Introduction

Years of research and development have resulted in high-throughput *automata processing* architectures and software engines [1]–[3]. This has lead to the discovery of new use-cases and application domains for finite automata, such as natural language processing, network security, graph analytics, high-energy physics, bioinformatics, pseudo-random number generation and simulation, data-mining, and machine learning [4].

Unfortunately, the software frameworks for the construction, manipulation, and translation of automata are frustratingly fractured (e.g. have inconsistent serialization formats) and restrictively licensed (e.g., Micron licenses a comprehensive SDK, but it is closed-source and specifically targets their D480 Automata Processor, or AP [2]). While these tools are useful for developing applications for the AP, the tools do not allow researchers to easily evaluate designs across hardware platforms, such as CPUs, GPUs, and FPGAs. The tools also 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.

Therefore, we have developed a suite of tools for creating, manipulating, and executing finite automata, which we refer to as MNCaRT (the MNRL Network Computation and Research Testbed, pronounced "minecart").[1] MNCaRT collects a diverse set of automata processing tools and algorithms into a central location and will grow as new tools are developed. We currently provide support for compiling state machines from Perl compatible regular expressions (PCRE) to automata, high-speed execution of NFAs and DFAs using Intel Hyperscan [3], and optimization and simulation of experimental automata designs with the Virtual Automata Simulator (VASim) [5]. Further, we provide back-ends for executing on GPUs [6], FPGAs [7], and the AP [2]. Finally, we allow users to explore routing constraints for experimental spatial architectures via the Automata-to-Routing (ATR) tool [8].

To support our ecosystem, we have created MNRL, the MNRL Network Representation Language (pronounced "mineral"), a JSON-based, open-source language to support the development of, and experimentation with, new automata-based applications and architectures. MNRL allows a user to define a *network* (or collection) of MNRL *nodes*, which represent the states within automata. Each node stores configuration information (such as node type, name, etc.) and connections to other nodes within the network. The language specification is general, allowing state machines other than finite automata to be represented. 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. Both MNRL and the tools in MNCaRT are publicly available (typically under BSD licenses), allowing both academics and industry experts to contribute to, and use, the ecosystem.

This work makes the following technical contributions:

- MNCaRT, an comprehensive repository of compatible tools for developing and experimenting with automata processing on CPUs, GPUs, and FPGAs.
- MNRL, an extensible, open-source JSON specification for representing state machines.
- Python and C++ APIs for reading, creating, manipulating, and writing MNRL files.
- Extensions to Intel's Hyperscan PCRE engine, supporting compilation to and execution of MNRL 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. In a non-deterministic finite automaton (NFA), it is possible to transition to multiple states on the same input symbol. Automata are often represented as

- *K. Angstadt and W. Weimer are with the Computer Science and Engineering Division, Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109. E-mail: {angstadt, weimerw}@umich.edu.*
- *J. Wadden, V. Dang, T. Xie, D. Kramp, M. Stan, and K. Skadron are with the Department of Computer Science, University of Virginia, Charlottesville, VA 22904. E-mail: {wadden, vqd8a, ted.xie, dankramp, mircea, skadron}@virginia.edu.*

Manuscript submitted: 04-Oct-2017. Manuscript accepted: 30-Oct-2017.

1. https://github.com/kevinaangstadt/mncart

a graph, defining the topological layout of the computation. Computation is therefore decoupled from the definition of the state machine, allowing for a common execution engine, which defines the execution model, to process arbitrary automata, improving code reuse and reducing sources for bugs.

In the remainder of this section, we briefly highlight some existing automata processing engines and discuss limitations of current automata representation languages.

Automata Processing Engines. Micron's D480 AP [2] is a custom hardware accelerator which directly executes homogeneous finite automata.[2] Becchi et al. have developed a set of tools and algorithms for efficient CPU-based automata processing [9]. Other CPU engines include Intel's Hyperscan [3]. Automata processing engines have also been developed for GPUsand FPGAs (e.g., [10], [11], and [12]). Unfortunately, existing engines do not share a common automata representation, making cross-architecture, comparison and development of automata-based algorithms challenging and time-consuming.

Limitations of Automata Representation Languages. The Automata Network Markup Language (ANML) is a proprietary description language developed for the Micron D480 AP. Licensing restrictions make the language challenging to use for prototyping new automata elements, and additional annotations cannot be added to elements in ANML while maintaining support for current tools. Therefore it is not a good choice for unifying automata processing engines.

Becchi et al.'s tools use a simple NFA representation based on the theoretic definition of NFAs and cannot be easily extended to support more complex state machines. The language is custom, and there is no support in general-purpose programming languages for reading and manipulating these files.

Regular expressions are commonly used to generate automata, but are difficult to develop and maintain. Many applications (e.g., particle tracking, motif searchers, and rule mining) would be represented by non-intuitive regular expressions that are often exhaustive enumerations of all possible matches. Additionally, programming of regular expressions can be extremely error-prone due to variations in regular expression syntax, which leads to high rates of runtime exceptions [13].

While other automata representation languages exist (e.g., Dot and JFLAP), these present similar licensing, generalizability, and maintainability challenges.

3 MNRL: A New Automata Language

We have developed MNRL, an extensible, open-source automata representation language, which allows for the topological specification of a collection of finite state machines using JSON syntax. While JSON is supported by most common general-purpose programming languages, we provide C++ and Python bindings to support additional validation checks.

It is important to note that the MNRL format specifies the layout of a machine but does not specify how elements behave, allowing many types of state machines to be represented, including traditional NFAs [14] and homogeneous NFAs [15].[3] Behavior is left for the execution engine to specify and implement (allowing MNRL to be an extremely flexible file format). Therefore, MNRL is similar in intent to the Unified Modeling Language (UML), in which developers describe and design software systems while eliding implementation details [16].

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

3. MNRL is general enough to represent more powerful machines (e.g. push-down automata, cellular automata, and Turing machines).

3.1 MNRL Format

A MNRL file contains 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 with a unique ID and specified width (number of wires)
- An array of output ports, each with a unique ID, specified width, and list of connected nodes
- Custom attributes, specific to each element type

A developer can encode 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 an engine needs information (e.g. symbol sets for matching against an input stream) to process a node, this 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.[4]

3.2 Extending the MNRL Schema

MNRL is designed to be extensible, enabling research on new, custom automata functionality and allows researchers to 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.The MNRL file format could easily be extended to support additional node types such as non-deterministic counters [17], 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.

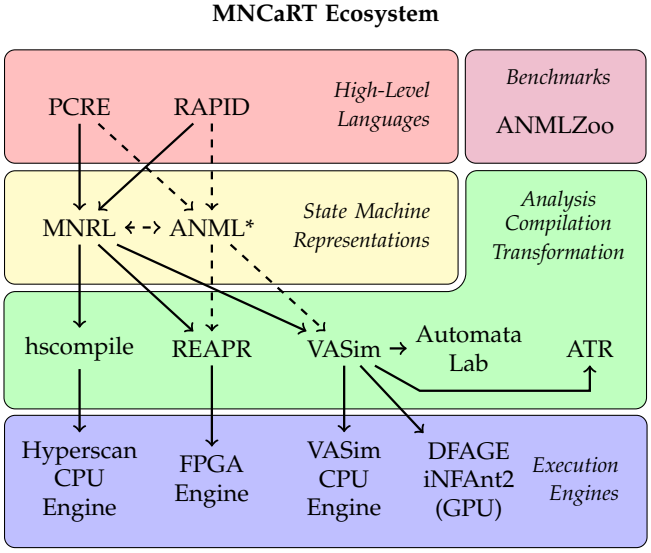
4 The MNCaRT Ecosystem

Our goal with the MNRL language is to enable the development of a rich, vibrant ecosystem of compatible tools for manipulating and executing automata. We are collecting these tools in an umbrella repository, the MNRL Network Computation and Research Testbed (or MNCaRT). By keeping tools catalogued in a single location, we hope to maintain the interoperability of tools and reduce fracturing in the ecosystem. We also provide a Linux container configured to use all of the MNCaRT tools.[5]

Figure 1 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. We also support execution on Micron's Automata Processor via conversion to ANML. Additionally, we provide compatible benchmarks for testing experimenting with tools in MNCaRT. In this section, we briefly describe to tools that make up the initial release of MNCaRT.

4. For additional details, please see Angstadt et al. [4].

5. https://hub.docker.com/r/kevinaangstadt/mncart



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

Fig. 1. Tools supplied as part of MNCaRT. These fall into four categories: front-end representations (both high-level and representation languages), benchmarks, transformation and compilation tools, and hardware and software execution engines.

4.1 High-Level Languages

Our framework supports programming models that represent pattern searches at a higher level of abstraction.

We compile PCRE to MNRL files using Intel Hyperscan's parsing and compilation routines [3]. Hyperscan is an open-source, high-performance regular expression processing library supported by Intel. The tool returns a graph representation of the compiled state machine, which we traverse to generate a MNRL file. Our regular expression compiler (**pcre2mnrl**) reads a file of regular expressions and compiles the given set of patterns to a single MNRL 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.

RAPID is a high-level programming language for execution of sequential pattern-matching applications [18]. 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 Benchmarks

The ANMLZoo benchmark suite contains a diverse set of automata applications and associated input stimuli [6]. Applications range from configurable, synthetic benchmarks to algorithms not easily represented by regular expressions and can therefore demonstrate vastly different execution characteristics. We have generated MNRL for all benchmarks in the suite.

4.3 Analysis, Transformation, and Compilation

Hyperscan Compilation. We provide an extension to Hyperscan (**hscompile**) that parses MNRL 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.

VASim. We have extended VASim [5] to support parsing of MNRL files. VASim is a general-purpose framework for automata simulation, optimization, transformation, and performance modeling. The tool enables prototyping, debugging, simulation, and analysis of automata-based applications and architectures. Additionally, VASim can parse Micron ANML files, allowing for conversion with MNRL.

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 [19] and a literal matching engine [3].

Automata Lab. Automata Lab is a web-based graphical environment for visualizing, editing, and simulating finite automata [20]. The tool uses VASim to manipulate automata, and the resulting state machines are displayed graphically, allowing for user interaction. Users may upload MNRL files or choose from applications in the ANMLZoo benchmark suite.

REAPR. We adapt REAPR [7], a tool for generating highly-efficient FPGA automata accelerator kernels, to support MNRL. The tool generalizes prior work [2], [11], [12] to be applicable for automata processing applications other than just regular expressions. Hardware automata accelerator engines such as REAPR take advantage of the one-to-one mapping between the spatial distribution of automaton states and hardware resources such as lookup tables (LUTs), block RAM (BRAM), and wires.

In REAPR, there are two main types of RTL elements to consider: 1) the state transition element (STE), which contains state activation information and transition logic; and 2) the wiring between all of the STEs in the automaton. Von Neumann automata engines iterate over every active STE and check whether the current input symbol will activate outgoing transition(s). If so, the next cycle's activation state is updated with the list of STEs that the current state affects. In an FPGA circuit generated by REAPR, STEs that affect each other are physically connected with wires, and if a single STE has multiple incoming transitions, they are combined in an OR gate so that any incoming transition can change the activation state of an STE.

Automata-to-Routing. We extend the Automata-to-Routing (ATR) [8] tool to support placement and routing of MNRL state machines. ATR utilizes the Versatile Place and Route (VPR) tool to model spatial automata-processing architectures [21]. We use VASim to emit VPR-readable circuits of MNRL networks and 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.

4.4 Execution Engines

Hyperscan CPU Engine. We provide a tool (**hsrun**) for processing MNRL 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 database and prints out human-readable reporting information (e.g. MNRL ID and input stream offset). If multiple compiled MNRL files and/or input files are passed to **hsrun**, the tool will execute all pairings of the files using a supplied number of threads.

VASim CPU Engine. In addition to support for transformation and analysis of finite automata, VASim supports simulation of a diverse set of finite automata models. While Hyperscan achieves higher throughput, VASim's modular design allows for quick prototyping to test new automata elements and designs, such as those including custom compute units.

FPGA Engine. In addition to generating hardware NFA kernels, REAPR can also generate a full platform execution environment for certain automata applications. The REAPR platform has been demonstrated to offer up to $183\times$ speedup over best-effort CPU implementations [7]. We are also actively developing a general-purpose reporting architecture to support execution environments for all automata kernels.

GPU Engines (DFAGE and iNFAnt2). MNCaRT contains both a GPU-based DFA engine (DFAGE) and NFA engine (iNFAnt2). The NFA engine was described previously by Wadden et al. [6]; we therefore focus on describing DFAGE in this article. Use of DFAGE first requires compilation to one or more DFAs using VASim. Note that the compilation process is performed offline by the CPU. Often, compiling to a single DFA is inefficient. Therefore, users may partition rulesets into several DFAs, and each DFA consists of a state transition table and an acceptance vector. State transition tables corresponding to different DFAs are stored consecutively in the GPU's global memory. The same layout is applied for acceptance vectors. It should be noted that each transition table is represented by a 2-D array containing the next state identifiers for every pair of current state identifier and input symbol. Similar to previous implementations, our DFA matching engine supports multi-packets processing to take advantage of the extreme parallelism of GPU architectures. Input packets also reside in the GPU's global memory.

Workloads are mapped to a 2-D grid of threads. Similar to Yu et al. [10], different packets are mapped to different blocks on the x-dimension of grid. Each thread within the block processes a different DFA for the assigned packet. However, for large datasets in our benchmark suite where the number of DFAs can exceed the block size, different blocks on the y-dimension of the grid will also be used.

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.

MNRL is a component of MNCaRT, a suite of tools for analyzing, executing, and transforming automata processing applications. 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.

Acknowledgment

This work was supported in part by grants from the NSF (CCF-1116673, CCF-1629450, CCF-1619123, CNS-1619098), AFRL (FA8750-15-2-0075), Jefferson Scholars Foundation, 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. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of AFRL.

References

[1] Titan IC Systems, "Helios RXPF soft IP for FPGA security analytics acceleration," http://titan-ic.com/products/helios-rxpf, 2017.

[2] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes, "An efficient and scalable semiconductor architecture for parallel automata processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 12, pp. 3088–3098, 2014.

[3] Intel, "Hyperscan," https://01.org/hyperscan, 2017.

[4] K. Angstadt, J. Wadden, W. Weimer, and K. Skadron, "MNRL and MNCaRT: An open-source, multi-architecture state machine research and execution ecosystem," University of Virginia, Tech. Rep. CS2017-01, 2017.

[5] J. Wadden and K. Skadron, "VASim: An open virtual automata simulator for automata processing application and architecture research," University of Virginia, Tech. Rep. CS2016-03, 2016.

[6] J. Wadden, V. Dang, N. Brunelle, T. T. II, D. Guo, E. Sadredini, K. Wang, C. Bo, G. Robins, M. Stan, and K. Skadron, "ANMLZoo: a benchmark suite for exploring bottlenecks in automata processing engines and architectures," in *2016 IEEE International Symposium on Workload Characterization (IISWC)*, Sept 2016, pp. 1–12.

[7] T. Xie, V. Dang, J. Wadden, K. Skadron, and M. R. Stan, "REAPR: Reconfigurable engine for automata processing," in *Proceedings of the International Conference on Field-Programmable Logic and Applications*, 2017.

[8] J. Wadden, S. Khan, and K. Skadron, "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)*, 2017.

[9] M. Becchi, "Regular expression processor," http://regex.wustl.edu, 2011.

[10] X. Yu and M. Becchi, "GPU acceleration of regular expression matching for large datasets: Exploring the implementation space," in *Proceedings of the ACM International Conference on Computing Frontiers*, ser. CF '13. New York, NY, USA: ACM, 2013, pp. 18:1–18:10.

[11] R. Sidhu and V. K. Prasanna, "Fast Regular Expression Matching Using FPGAs," in *Proceedings of the the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 227–238.

[12] Y. H. Yang and V. Prasanna, "High-performance and compact architecture for regular expression matching on FPGA," *IEEE Transactions on Computers*, vol. 61, no. 7, pp. 1013–1025, July 2012.

[13] E. Spishak, W. Dietl, and M. D. Ernst, "A type system for regular expressions," in *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*, ser. FTfJP '12, 2012, pp. 20–26.

[14] M. Sipser, *Introduction to the Theory of Computation*. Thomson Course Technology, 2006, vol. 2.

[15] P. Caron and D. Ziadi, "Characterization of Glushkov automata," *Theoretical Computer Science*, vol. 233, no. 1, pp. 75–90, 2000.

[16] M. Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, ser. Object Technology Series. Addison-Wesley, 2004.

[17] M. Becchi and P. Crowley, "Extending finite automata to efficiently match perl-compatible regular expressions," in *Proceedings of the ACM International Conference on emerging Networking EXperiments and Technologies*, ser. CoNEXT '08, 2008, pp. 25:1–25:12.

[18] K. Angstadt, W. Weimer, and K. Skadron, "RAPID programming of pattern-recognition processors," in *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16, 2016, pp. 593–605.

[19] M. Becchi and P. Crowley, "Efficient regular expression evaluation: Theory to practice," in *Proceedings of Architectures for Networking and Communications Systems*, ser. ANCS '08, 2008, pp. 50–59.

[20] D. Kramp, J. Wadden, and K. Skadron, "Automata Lab: An open-source automata visualization, simulation, and manipulation tool," University of Virginia, Tech. Rep. CS2017-03, 2017.

[21] V. Betz and J. Rose, "VPR: A new packing, placement and routing tool for FPGA research," in *Proceedings of the International Workshop on Field Programmable Logic and Applications*. Springer, 1997, pp. 213–222.