Automata Processor

Tobias Markus
Computer Architecture Group, University of Heidelberg

***Abstract*—This paper gives a brief overview over non-deterministic automata and the Automata Processor an architecture implemented by Micron inside their DDR3 SDRAM technology to directly map NFA designs. The Automata Processor can be programmed via Regular Expressions or the Automata Network Markup Language (ANML) an XML-like language to describe an automaton. Big data search and analysis are problems which are a good fit for the Automata Processor.**

I. Introduction

In the past, the performance of traditional CPUs have increased by frequency scaling. Frequency scaling stopped because of the heat wall since the power in CMOS devices increases linear to the frequency $P = C * V^2 * f$. With the heat wall in mind, the new goal is to increase performance with holding the power consumption constant in each generation.

According to Moore's law, the technology complexity is doubling regularly. Moore's law is not a physical law. It is more an observation, nonetheless industry is following this trend. This means we are able to use more transistors since with decreasing gate capacity the power drops. To increase performance in modern architectures, parallel systems and concepts are the solution.

Massive parallel systems like GPUs or CPU clusters are a good fit to work on structured data. But there are some problem loads like random access, graph problems and pattern matching which are not fitted for these common architectures.

Some of these difficult problem loads can be directly mapped into NFAs. The Micron Automata Processor is an approach in this direction trying to implement a RegEx engine or rather a NFA engine.

This paper gives an introduction in the Automata Processor. The paper begins with an introduction in automaton theory in section 2. In section 3, the basic elements of the Automata Processor are described. The Automata Processor can be programmed with regular expressions or with ANML an XML based language. This two workflows are described in section 4. In section 5, different application areas are given as examples and are examined. In the last section, a conclusion with an evaluation and opinions are given.

II. Automaton Theory

In general, there are two types of automata. One is the deterministic finite automaton (DFA) and the other the non-deterministic finite automaton (NFA). The main difference between these two is that the DFA can only have one active transition in each state. There is a transition function. The NFA does not have this limitation. Each state can have many transitions to a set of other states. This is realized by a transition relation and not a transition function.

The NFA has some advantages compared to the DFA. Complex tasks can often be implemented with less states and less transitions. In some cases, the number of states of a DFA can increase exponentially in comparision with an equivialent NFA.

To examine the NFA further, we need to define a mathematical model. The NFA can be described with a 5-tuple $< Q, \Sigma, \delta, q_0, F >$ where Q is the set of automaton states, Σ is the input alphabet, δ is the transition relation, q_0 is the start state, and F is the set of the final states. The transition relation depends on the current state q and the input symbol α $\delta(q, \alpha)$, and defines which states can be reached next.

The automaton accepts a word w when each character results in a new state or a set of states and the last character get matched with an active final state. The following conditions must be met for accepting the word w:

- $r_0 = q_0$
- $r_{i+1} \in \delta(r_i, a_{i+1}), i = 0, ..., n-1$
- $r_n \in F$

All the strings (w) accepted by the automaton (M) are called the accepted language ($L(M)$). We can define $L(M)$ as the set of w where the intersections of the set of transition relations and the set of fineal states is not empty.

$$L(M) = \{w \mid \delta^*(q_0, w) \cap F \neq \emptyset\} \tag{1}$$

Another important definition is the ϵ-transition. An ϵ-transition allows the automaton to do an transition without consuming an input.

An example of a NFA which accepts binary strings ending with "01" with the example input "00101":

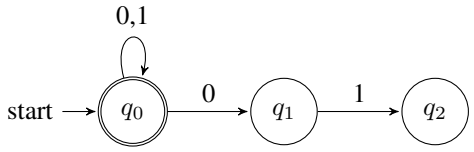

Fig. 1. NFA example detecting binary strings ending with "01"

- $\delta^*(q_0, 0) = \{q_0, q_1\}$
- $\delta^*(q_0, 00) = \delta(q_0, 0) \cup \delta(q_1, 0) = \{q_0, q_1\} \cup \emptyset = \{q_0, q_1\}$
- $\delta^*(q_0, 001) = \delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$
- $\delta^*(q_0, 0010) = \delta(q0, 0) \cup \delta(q_2, 0) = \{q_0, q_1\} \cup \emptyset = \{q_0, q_1\}$
- $\delta^*(q_0, 00101) = \delta(q0, 1) \cup \delta(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$

In this example the current state set for each new input character is calculated incrementally, with the knowledge of the states activated in the step before. The transition relation can be directly taken from the graph.

More detailed information about automaton theory can be found in [2], [1] and [3].

III. Implementation

The Micron Automata Processor (AP) is implemented in Microns standard DDR3 SDRAM memory array technology. The AP is capable of processing 8 bit input symbols at a rate of 1Gbps. The row address is used as the input symbol for the AP and with that given to the memory array. The AP than invokes operations with its State Transition Elements (STE) and the routing matrix, see figure 2. This architecture can directly implement an NFA in a parallel manner. In this section, each element in the architecture will be explained in detail. Most informations from this section are taken from [8] and [9].

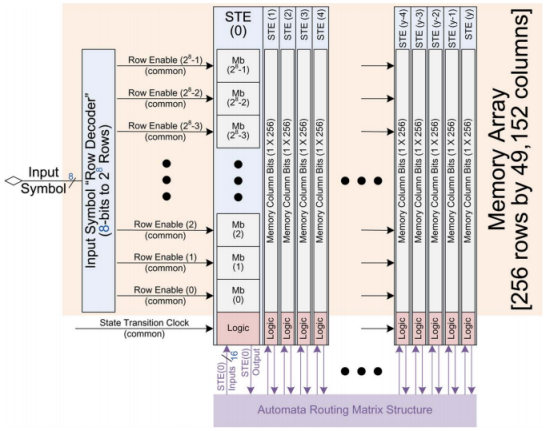

Fig. 2. Automata Processor Memory Array

A. State Transition Element

The State Transition Element (STE) is the basic element of the AP. It consists of a state memory which gets the input symbol as row address. The state memory contains the matching character class and the state transitions logic to determine when the state is active and when an output is given.

The state memory is basically a look up table which is programmed to recognize a character or a character class. As mentioned before, the input signal addresses the 256 Bits of state memory. The one bit output indicates if a match happened and is forwarded to the STE logic. This concept allows it to do every match on a single character with only one STE independent from complexity. For example to match with the character ”a” which is 0x61, we have to set the memory bit at address 0x61 to one and every other memory bit to zero. The other case is that we want to match a character class. In this case, we have to set every matching case to one and every other case to zero. The matching characters are for example ”0”-”9” (0x30 - 0x39). The memory bits from address 0x30 to address 0x39 have to be set to one and every other memory bit to zero.

The additional logic contains the state bit which indicates whether the state is active ”1” or inactive ”0”. There are

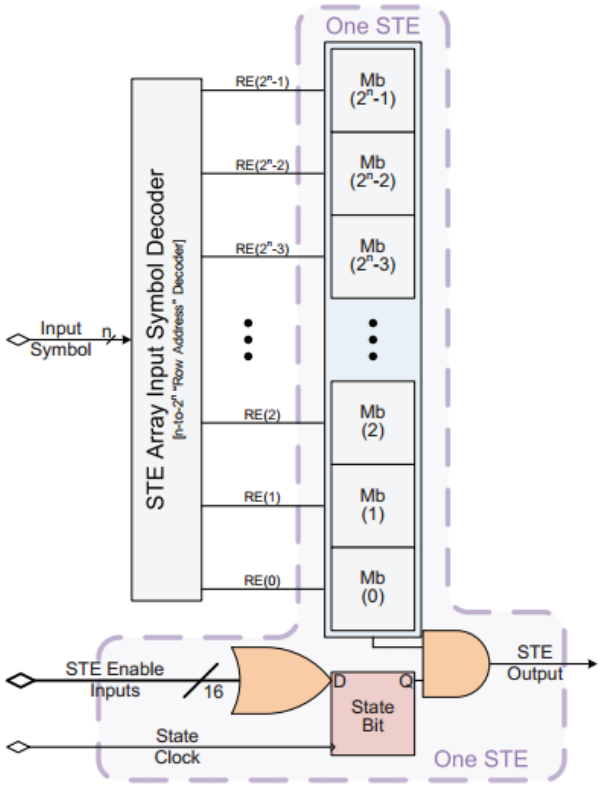

Fig. 3. State Transition Element

16 state enable inputs coming from other STEs which are forwarded to a logical OR. The output of the OR determines if the state bit is set or reset. The state output is set when the state bit is set and the output of the state memory is one. An important feature is that every state bit can be preloaded. This allows every state to be a start state. Multiple start states allow the implemention of multiple independent automata inside the AP. But it also allows the implemention of ϵ-NFAs since with multiple start states we are capable of rewriting them to a normal NFA. See the example in figure 4. Here the ϵ-transition activates the state q_1 before any input is consumed. When there is an ”a” as input character q_1 and q_0 get activated again. This can be rewritten to an NFA with two start states an slightly other transitions.

B. Counter

The counter element in the AP is a 12 bit counter. A count enable signal lets the counter count up by one when it is asserted. If the counter equals the target value, an output is set. The counter has some additional features. Up to four counters can be cascaded, a synchronous reset function and the ability to choose different row signal inputs for count or reset functions.

The counter element can simplify the automaton. Typical use cases are if subexpressions need to be counted or for regular expressions with other quantifications. Counters in automata offer the abilities to implement a subset of pushdown automata. Furthermore, it is possible to implement Turing machines.

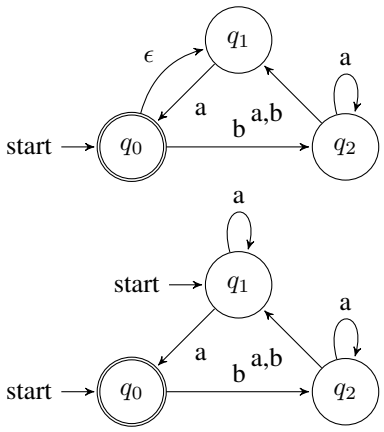

Fig. 4. ϵ-NFA conversion to NFA

One practical example for the utilization of counters is the use as an one shot timer to run through a circled NFA only once.

C. Boolean Element

The boolean element is a combinatorial element which can be programmed as OR, AND, NAND, NOR, sum of products or product of sums. They have no state in contrast to counter elements or STEs. Like counter elements this elements are also used to simplify the designs. To implement the evaluation at the end of data, boolean elements can be used. These are often used in regular expressions (rigth-anchored expressions).

This feature allows the AP to implement functions beyond the formal definition of an NFA.

D. Routing Matrix

To implement an NFA, there must be programmable connections between the programmable operations of the STEs. This is implemented as a programmable routing matrix connecting the Elements within the AP. The routing matrix consists of programmable switches, buffers, routing lines and cross point connections. In the formal NFA model, the state can be connected to each other element. Due to the performance issues regarding clock speed, power, and propagation delay, a trade-off is made.

The routing matrix is implemented hierarchically. Various AP elements are grouped to a row and rows then are grouped into blocks. The blocks are implemented in a grid of block rows and block columns, see figure 5.

A signal can be routed from any given point to several other points of the nearby hierarchy.

E. Reconfigurability and Inter Rank Bus

The AP is a programmable device. The operations of the element arrays and their connections can be programmed. Another feature of the AP is that it is partially reconfigurable. The operation of the elements can be reprogrammed at runtime. To reprogramm the connections between the elements, a place and route step has to be done. But it is possible to load a structure incrementally inside the AP.

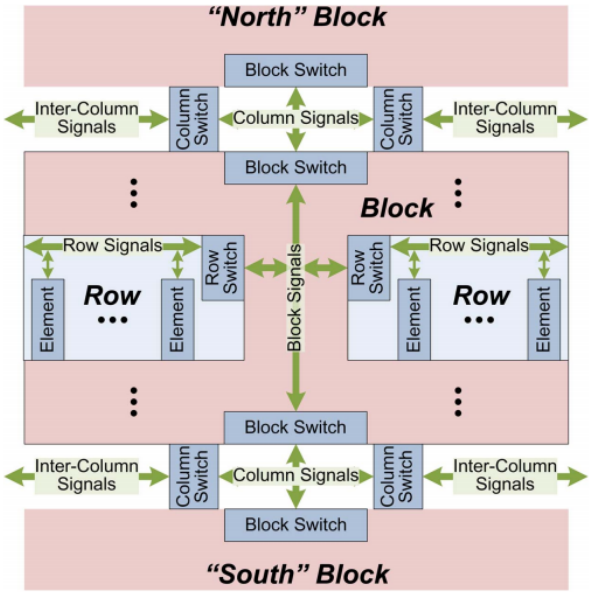

Fig. 5. routing matrix hierarchy

The scalability of the AP is important to work on huge datasets. To achieve scalability, an inter rank bus is available which allows input symbols to be distributed over several APs. This increases the amount of states and the throughput.

IV. Programming the AP

The AP can be configured via Perl Compatible Regular Expressions (PCRE) or via an XML-based language called the Automata Network Markup Language (ANML).

```
include <ap_compile.h>

ap_automaton_t CreateAutomaton(void)
{
    ap_automaton_t amton;
    ap_exprdb_t db;

    // Create an expression database
    db = AP_CreateExpressionDB();

    // Add an expression with PCRE delimiters
    // and set the i PCRE modifier (caseless).
    // The expression identifier is set to 1
    AP_AddExpression(db, NULL, "/a(b|c)*[de]/i", 1,
        0, AP_GRAMMAR_PCRE_DELIMITED);

    // Add an expression without PCRE delimiters
    // and set to caseless matching
    // The expression identifier is set to 1
    AP_AddExpression(db, NULL, "x.*yz", 1,
        AP_MOD_CASELESS, AP_GRAMMAR_PCRE);

    // Compile the expression database
    // into a automaton
    AP_Compile(db, &amton, 0, NULL, 0,
        DEVICE_FAMILY_FRIO);

    // Clean up
    AP_DestroyExpressionDB(db);

    return amton;
}
```

Listing 1. PCRE implementation and compelition as Automata

ANML is developed by Micron to describe automata. In this section, both variants are introduced. Micron offers an AP SDK in which both tools for compiling regular expressions and tools to create and compile ANML descriptions into automaton are available. The SDK for example offers C libraries for compiling regular expressions and ANML or even for creating ANML from C. The SDK also includes the AP Workbench, a development environment to visually create NFAs, compile and simulate them. For this section more detailed information can be found in the SDK User Guide [5] and the ANML User Guide [4]

A. Definition by Regular Expressions

Perl Compatible Regular Expressions (PCRE) are compatible to the AP. Because not every expression can be directly implemented as an NFA, software post-processing is needed for some constructs. Even with the counter and boolean elements, there are some restrictions on how to implement this constructs in order to assure hardware performance. Regular expressions can be defined within a C/C++ description of the automaton. Listing 1 shows an example that implements the regular expressions /a(b—c)*[de]/i and /x.*yz/i. And then compiles it into an automaton with AP_Compile.

B. Definition by ANML

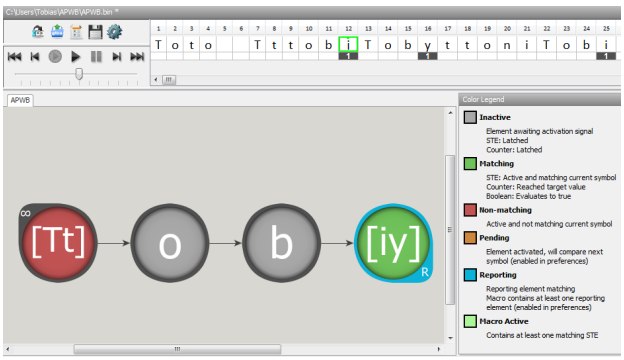

Fig. 6. Example automata in the AP Workbench simulation

With ANML an XML-based language, it is possible to directly describe and implement automaton structures inside the AP hardware. A possible workflow is to write the automaton design directly in ANML then read it into a C program in which we compile the automaton using the in the SDK provided libraries. But there are other ways to implement an atomaton using ANML. The libraries provided by the SDK offer the possibility to implement an atomaton within the C code and than generate an ANML out of it.

The other way to develop an ANML automaton is to use the AP Workbench to implement the NFA graphically. The AP Workbench then generates automatically the ANML representation of the graph and can also compile and simulate the design. A detailed documentation of the AP Workbench can be found at [6].

Figure 6 shows an example of an NFA which matches the string ”Tobi” either with uppercase or lowercase ”T” and with

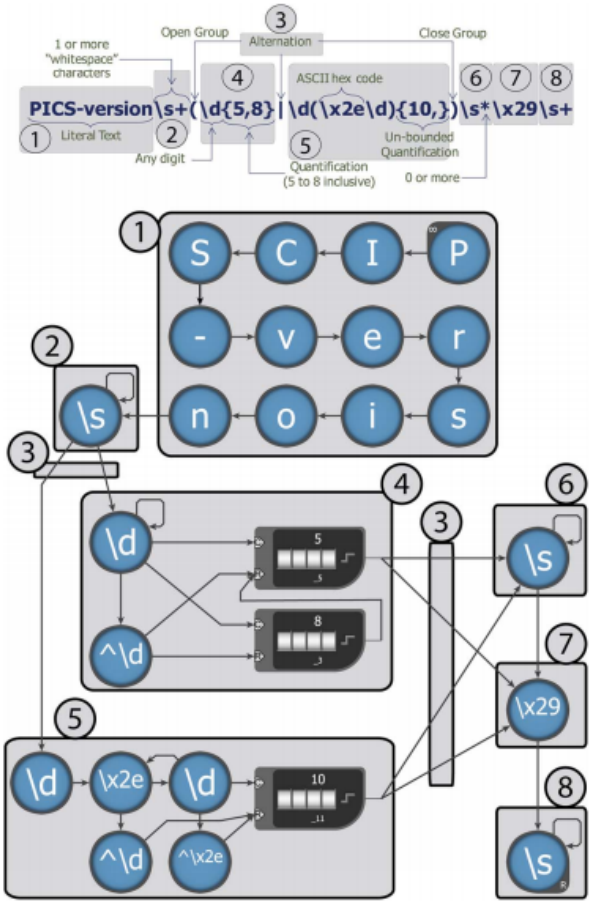

Fig. 7. Example rule to detect buffer overflows in Apache webservers

”i” or ”y” at the end within the AP Workbench. The automaton is graphically implemented with the Automata Workbench. The first state is matched when there is an uppercase or lowercase ”T” and gets every input symbol as indicated by the infinity symbol in the upper left. The next states detect ”o”, then ”b” and the last symbol detects an ”i” or a ”y” and reports when it is active as indicated by the R in the lower right corner. After the implementation step, the automaton is compiled and than simulated with the stimuli seen in the figure. The simulation tools offer to step through the stimuli and visually see how the automaton behaves. A green state means an active state that has a match, a red state means it is active but does not match the current input, a gray state is inactive and blue means the state is reporting.

V. Applications

The APs applications are mainly in the fields of big data research and analysis. The main areas for the AP are Bioinformatics, Network Security, Sigint and Crypto and Finance applications. In this section, we will introduce the fields Network Security and Bioinformatics and examine which task could be efficiently solved with the AP. Informations about every application field can be found in [7].

A. Network Security

The AP is a perfect match for signature-based detection systems which can examine for example network packets for known malware patterns. This method is effective against known malware. An easy approach from the attacker side is to modify or encrypt parts of their code to hide from simple signature-based detection. To detect this modified variants, generic signatures are used. This generic signatures are checking for many variations in known malware patterns. With wild carded regular expressions, even malware with extra, rearranged and partially encrypted data can be detected. The compute load for this many variants is higher than simple pattern matching and network speeds are increasing. The main advantage of the AP for this application is that it can handle complex regular expressions at high bandwidth throughput. Allowing to even try matching possible obfuscated portions themselves.

There were several rulesets taken and simulated for the AP. One specific rule was taken from SNORT and the PCRE was converted to an automaton and simulated. This specific rule shown in figure 7 is used to capture buffer overflow attacks against Apache webservers.

B. Bioinformatics

There are several Bioinformatic calculations and problems in which the AP fits in. Sequencing of DNA is one of this problem fields. Modern sequencing needs a reassemble step where overlapping sequences have to be matched. Finding patterns in the DNA is an ideal task for the AP. Figure 8 depicts a modern sequencing workflow.

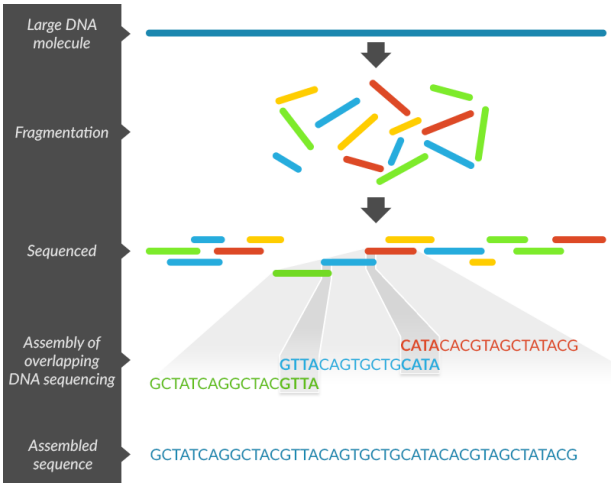

Fig. 8. DNA sequencing workflow

Another task in Bioinformatics is the search for motifs which are sequences across multiple DNA or protein sequences. Motifs give biological informations of the given sequence. Discovering motifs helps in finding variants and cause of genetic diseases, identifying elements in a sequence and their function and in helping to suggest therapeutic drug targets.

In the example shown in figure 9 we want to find the motif of N-Glycosylation, an attachment of the sugar molecule. The pattern matching rule for the motif is NˆP[ST]ˆP. For this, we need 4 states. We first match the ”N” than not ”P” then we match ”S” or ”T” and in the last state not ”P” again. The last state also reports that the motif was found.

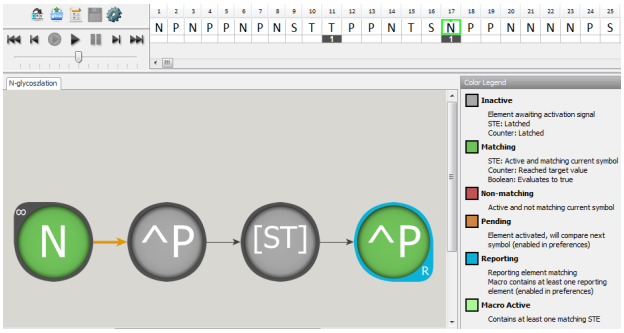

Fig. 9. Automata to match Motif pattern of N-Glycosylation

VI. Conclusion

The Automata Processor is an interesting new architecture which offers a very efficient solution for pattern matching and NFA tasks. A huge problem is that the applications for the Atomata Processor are very specialized, for example in Bioinformatics, network security and finance. For other fields, the Automata Processor does often not offer a complete solution but might accelerate post processing steps or subtasks. On the other hand, it is difficult to estimate which problems fit into the AP. The counters and boolean elements allow the implemention of complex NFA like systems.

The possibility to have a defined description language for automaton like ANML is practical but due to its complexity it is not a format for designing. Despite that it is a good format for interfacing between programs. It could be that most automata will be created from the graphical interface of AP Workbench. It has to be shown that the graphical creation of complex automaton is feasible but there is also the possibility to create ANML out of C or Python code with the libraries offered in the SDK .

The logic for the Automata Processor is implemented into SDRAM technology. The advantage of this is that the process is cheap and a process which is well understood by Micron. So for them, it is a good solution. The disadvantage of implementing the logic inside the SDRAM process is that there are only few routing layers available. Another way would be to implement SDRAM into a normal ASIC. The advantage is that there are a lot of routing layers available and with this a better routing matrix would be possible. The disadvantage of implementing SDRAM into a normal ASIC process is that the SDRAM needs more space and it is not as cheap in production.

All in all, the use of a non von Neumann processor which can directly implement NFAs will fit good into the current accelerators for high performance computing but it may only be a solution for very specialized fields. The other question is whether the first implementation from Micron will be a good one. And what the comprehension with modern FPGA solutions will be.

REFERENCES

[1] Andrew Tolmach Andrew P.Black. Nondeterministic finite state automata. *CS311 Computational Structures*, 2003.

[2] Mirian Halfeld-Ferrari. Finite automata. *Automata Theory, Languages and Computation*, 2003.

[3] Wing-Kai Hon. Lecture 4: Automata theory 2. *CS5371 Theory of Computation*, 2007.

[4] Micron. Anml documentation. http://www.micronautomata.com/documentation/anml_documentation.

[5] Micron. Ap sdk user manual. http://www.micronautomata.com/documentation/ap_sdk/index.html.

[6] Micron. Ap workbench manual. http://www.micronautomata.com/documentation/ap_workbench_documentation.

[7] Micron. Automata applications. http://www.micronautomata.com/#applications.

[8] Paul Glendenning Michael Leventhal Harold Noyes Paul Dlugosch, Dave Brown. An efficient and scalable semiconductor architecture for parallel automata processing. *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS*, 25:11, 2014.

[9] Paul Glendenning Michael Leventhal Harold Noyes Paul Dlugosch, Dave Brown. Supplementary material for an efficient and scalable semicosemicon architectur for parallel automata processing. *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS*, 25:5, 2014.