Document Oriented Modeling of Cellular Automata

Christian Veenhuis, Mario Köppen
Fraunhofer Institute for Production Systems and Design Technology
Department Pattern Recognition
Pascalstr. 8-9, 10587 Berlin, Germany
`Email: {veenhuis|mario.koeppen}@ipk.fhg.de`
Key Words: cellular automata, cellular automaton document, modeling language, algorithm design

Abstract.

This paper proposes a document-oriented modeling concept for cellular automata (CA), which supports the simple and rapid design of a huge variety of cellular automata. This modeling concept is realized as a domain-specific modeling language derived from XML (e**X**tensible **M**arkup **L**anguage). XML is in general considered as the future for internet documents and data exchange. The main concept behind XML is to separate the content of a document from its layout (its appearance). The presented modeling concept uses a document for describing a whole cellular automaton. Like the content of a document is separated from its layout, the abstract cellular automaton is separated from a concrete implementation and programming language. Everyone can create and use XSL(T) stylesheets for translating cellular-automaton-documents into ready to use source-code (covering the adequate cellular-automaton-functionality) as well as for documentation and exchange of the realised CA.

1 Introduction

A cellular automaton (CA) is a system consisting of units called cells whose discrete states change over time. These cells are arranged by a specific type of lattice, as depicted in figure 1. There is a variety of possible lattices. Lattice type `(a)` defines a hexagonal grid, where every cell has six neighbors. Lattice type `(b)` defines a one dimensional square lattice with each cell having two neighbors. The square lattices `(c)` and `(d)` can either have a five-neighborhood (*von Neumann neighborhood*) or a nine-neighborhood (*Moore neighborhood*).

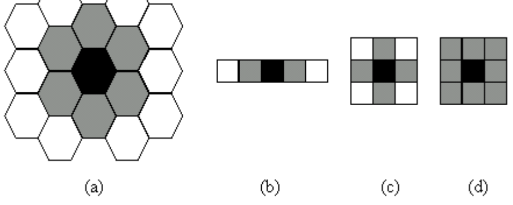

Figure 1: Different lattices for cellular automata

A neighborhood is assigned to each cell, according to the given lattice type. Furthermore, every cell possesses a rule (or rule set) which determines the state of this cell in dependence on its neighbors. According to [9] and [3] a one dimensional CA can be defined by equation 1, whereby r defines the range for the neighbors. In figure 1 the lattice type `(b)` uses $r = 1$.

$$a_i^{(t+1)} = \Phi(a_{i-r}^{(t)}, \ldots, a_i^{(t)}, \ldots, a_{i+r}^{(t)}) \tag{1}$$

The state of the cell for the next generation $t+1$ can be computed by a function Φ. For computing the next state, the function Φ uses the appropriate neighbors.

Analogous to equation 1 a two dimensional CA with a five-neighbor square can be defined as in equation 2. CAs with any number of neighbors and even with multiple dimensions can be defined similarly to equation 2.

$$a_{i,j}^{(t+1)} = \Phi(a_{i,j}^{(t)}, a_{i,j+1}^{(t)}, a_{i,j-1}^{(t)}, a_{i+1,j}^{(t)}, a_{i-1,j}^{(t)}) \tag{2}$$

Cellular automata are used in various fields, for example, biology, chemistry, and physics to simulate (non-linear) phenomena like computational epidemics, diffusion of particles or the flow of fluids. CAs are also used for computational, cryptographic, and image processing tasks.

In the past, some activities were directed at the implementation of software libraries covering CA functionality. Furthermore, there already exist languages like *cellang* for the development of CAs [2]. These languages are specifically designed for the domain of CAs and are often interpreted. A lot of software systems are avaible to simulate and explore CAs. Unfortunately, the created and defined CAs can't be exchanged between these software systems. If one wants to use a created CA with another simulator, the CA has to be re-programmed or re-defined. Furthermore, if one has written a CA in a programming language (e.g. C,C++,Java) and needs to use this CA on another platform or in another programming language, the CA has to be re-programmed, too.

When developing or applying cellular automata it is useful to handle a specific CA on a more abstract level than the level of programming languages or function-calls. To overcome the mentioned problems and to revive efforts in exchange and standardization, a concept for a cellular-automaton-document is proposed, which supports the modeling, description and documentation of CAs at once. The cellular-automaton-document (CAD) should be system-independent and separated from an explicit implementation. It should also be easy to understand and to learn.

Looking for a suitable concept, different approaches have been studied. Among them are the E**x**tensible **M**arkup **L**anguage (XML) [1] [8] and **E**volutionary **A**lgorithm **M**odeling **L**anguage (EAML) [4] [6], whereby EAML is derived from XML. Both languages are not traditional programmer-languages that specify a computation flow. They are developed to specify and characterize elements of an environment. These languages abstract several elements by adapted constructs and support their hierarchical order within a document. As an example EAML is used to model Evolutionary Algorithms (EA) like Genetic Algorithms, Genetic Programming, Evolutionary Strategies, and so on. Every EA is built up by a hierarchical order of elements, which represent the different (genetic) operators and methods. This way a semantic model is constructed: the kind of order and the nesting of the elements represent the specific EA.

The authors propose a concept for a **C**ellular **A**utomata **M**odeling **L**anguage (CAML) [5], which is derived from XML and works similar to EAML: Several elements are ordered and nested to describe any set of rules and global properties on an abstract level. Finally, the proposed element-hierarchy realizes a complete CA. Using XML-notation documentation and exchange of a CA could be nice. Because CAML is derived from XML, it 'inherits' the platform independency, readability (for humans (text-based) and software (structured)), and the more general and abstract handling because it is independent of the implementation (programming language). Therefore a CAML document can be exchanged between all software simulators, which possess a CAML interpreter. Furthermore, a CA described by a XML notation can be translated into various source-codes (C,C++,Java etc.) by using stylesheets (see figure 6). For this, XSL(T) (**X**ML **S**tylesheet **L**anguage (**T**ransformations)) can be used. By using stylesheets, one does not need to be familiar with compiler technology to translate CAs into source-code. In addition, CAs modeled with CAML can be translated or transformed into every text-based format. Thus, documentations could be generated automatically by using an appropriate stylesheet. It might even be possible to translate CAML documents into hardware description languages like VHDL (but this point has to be examined in the future).

Section 2 presents the current state of CAML. To find out whether CAML is suitable for practical applications or not, a CAML-interpreter was implemented concomitantly and will be introduced briefly in section 3. Finally, section 4 gives an outlook on to further developments. The authors would like to motivate further contributions for the extension of CAML as well as for standardization and exchange within the CA-community.

2 Derivation of the Cellular-Automaton-Document

2.1 Components of a cellular automaton

To be able to model a CA, its specific components have to be recalled. In general, a CA mainly comprises the following components:

- A *lattice*, which arranges all cells.
- At least one *state* with a valid range of values defined by an interval $[k_{min}, k_{max}]$.
- A range for the local *neighborhood* (the r in equation 1).
- A set of at least one *rule* (the Φ in equation 1 and 2).
- A set of logical functions for the use with the neighbor cells as *conditions* for rules.
- *State-readers*, which can read the state of neighbors and the current cell.
- A *state-writer*, which can set the current cell to a new state. Only the current cell should be manipulable. If we could change also the neighbors, we wouldn't have a CA.

Although upcoming components might not be predictable, hence, the modeling concept for a **C**ellular-**A**utomaton-**D**ocument (CAD), that describes CAs, has to consider the huge variability of behavioural and structural design of a CA. The different components of a CA, rules and computations, could be handled as building blocks. So, it will be possible to fiddle about with those building blocks, being simply called *elements*. Finally, a complete CA could be considered as a nested arrangement of elements. For user convenience the most common computations should be pre-defined.

2.2 Notation and basic concepts of CAML

The notation and the basic concept of CAML follows the XML-standard [1]. In general, the structure of CAML consists of several elements, and an ordered collection of such elements can be called a CA-description. Together with some administrative XML elements, such a CA-description is stored in an ASCII-file. This ASCII-file is called CAML document and can be created with a simple text-editor or even with every XML editor by using the Document Type Definition (DTD) of CAML. A DTD is a XML description (grammar) for the structure of a document (language), which is derived from XML [1].

The different elements are applied by the following general notation:

```
< NameOfElement
  Attribute[i].name = "Attribute[i].value"
>
  content (...further nested elements...)
</ NameOfElement>
```

Each element is introduced by a so-called start-tag (<*NameOfElement*>) and finished with an end-tag (</*NameOfElement*>). Enclosed in these both tags the element-content is written. Depending of the element-type an attribute-list containing further parameters can be specified. It has to be considered that attributes are able to carry default-values. Therefore, they haven't to be specified explicitly. Some elements might also contain further elements. Therefore, it is spoken of nested elements or an element-hierarchy.

The definition-notation for the element-types and their attributes were derived from XML [1] and will not be explained further at this point.

Each CA-functionality, like conditions, actions or computations, required for the description of a CA, has to be abstracted by the mentioned elements. So, up to now, every CA-functionality might be handled as complete parameterized building blocks. To give an example, an expected one dimensional pattern can be described by the following notation:

```
< Pattern dimx="3" centerx="1" > 1 0 1 </ Pattern>
```

The element is named after its meaning and comprises two attributes. With the `dimx`-attribute the length of the pattern can be determined. The `centerx`-attribute defines the position of the current considered cell within the pattern. As mentioned before, the attribute-values might be changed in a specific CA-description. Another example that should be provided is a *Neighbor*-element that provides the reading of the neighbor cells (now for a two dimensional CA):

```
< Neighbor x="-1" y="-1"></ Neighbor>
```

and shortened according to the XML-standard:

```
< Neighbor x="-1" y="-1" />
```

This element possesses two attributes, which contain the offsets to the neighbor cell. This way any number of neighbors with any range can be realized (the *r* in equation 1).

By using the presented notation all components of a CA can be modeled, no matter whether they are rules, state-readers, state-writers, conditions, and so on.

2.3 The element-hierarchy

A complete CA-description consists of diverse elements arranged in a nested order. So, these elements form a hierarchy, which is mentioned as element-hierarchy. This element-hierarchy also represents the affiliation of an element, whereby each element of a lower level is contained in an element of the higher level. Currently, on the top-level there are the following elements permitted:

CA	Covers the complete description of one CA
Rule	Describes a (single) rule

As suborder-elements of the *CA*-element, up to now, the following elements were provided. Moreover, some of these elements contain further elements as content.

Conditions	EVER, OR , AND , NOT , Pattern , GT , LT , EQ , UNEQ
Operators	Break , Literal , Min , Max , Add , Sub , Mul, Div, Sum
State-readers	Neighbor , Current
State-writers	Current
Structure	Cond , Action , Otherwise

The number of elements is not restricted. From a *CA*-element hierarchically downwards all components of a particular CA are specified.

Figure 5 depicts the Document Type Definition (DTD) of the current state of CAML. This DTD defines the grammar of valid CAML documents. For this it defines the affiliations of the elements and their attributes.

In figure 2 a simple example of a valid CAML document is given. There, Conway's *Game of Life* is described with CAML. This is a very old application of CAs, but it's well-known and therefore a good example for a CAML document. The description of a CA is realized by several sub-trees with the root being a *Rule*-element. A single sub-tree contains e.g. a condition (for activation of a rule) or an action which computes and sets the new state for the current cell. The example in figure 2 contains two rules for describing the rules of the *Game of Life*:

1. If a cell is dead, it will become alive if it has exactly 3 neighbors which are alive

2. If a cell is alive, it will keep alive if it has 2 or 3 neighbors which are alive (otherwise it will die)

Every *Rule*-element can contain any number of *Cond*-, *Action*- and *Otherwise*-elements. Each of these elements can contain several sub-trees of theoretically infinite depth for describing the rules in all complexity. This way it is possible to design a wide variety of different CA-descriptions.

Equation 3 shows the definition of a CA described with CAML. A `rule` (R_n) consists of `conditions` (C_n), `actions` (A_n) and `otherwises` (O_n) realized by the appropriate CAML elements. A whole CA Φ contains several `rules` (R_n).

$$
\begin{array}{rcl}
S_k & := & k^{th} \text{ sub-tree} \\
C & = & \{S_1, \ldots, S_{ntrees}\} \\
A & = & \{S_1, \ldots, S_{ntrees}\} \\
O & = & \{S_1, \ldots, S_{ntrees}\} \\
R & = & \{C_1, \ldots, C_{nconditions}\} \cup \{A_1, \ldots, A_{nactions}\} \cup \{O_1, \ldots, O_{notherwises}\} \\
\Phi & = & \{R_1, \ldots, R_{nrules}\}
\end{array}
\tag{3}
$$

These rules are executed in the order of their position within the CA-description. For every rule the activity has to be computed to decide whether the action or the otherwise part must be executed. The activity for a given rule R_n can be computed like in equation 4: At least one *Cond*-element (C_i) of the *Rule*-element (R_n) should evaluate to 1. For this, each of the n_{trees} sub-trees of this *Cond*-element must evaluate to 1, too.

$$
\exists C_i \in R_n \left(\bigwedge_{k=1}^{n_{trees}} S_k \in C_i \right) \tag{4}
$$

If a rule R_n of the CA fulfills equation 4, all *Action*-elements (otherwise all *Otherwise*-elements) of this rule are executed.

In figure 2 the conditions are defined by combining relational and logical operators with arithmetic functions. This resembles to syntax-trees of programming languages, where expressions are managed by tree-structures. Defining conditions this way can be very flexible but also bloating. The obtained CAML-document could be somewhat unreadable for human beings. Therefore, one is able to define conditions directly by specifying patterns like in figure 3. This way one has a more concise style than by using arithmetical computations. Figure 3 describes the Sierpinsky Triangle as shown in figure 4. Both styles (like in the figures 2 and 3) can be combined to describe as much CAs as possible.

3 Realisation

Concomitant with the CAML-concept, an interpreter was implemented for developing and testing of CAML descriptions. This was necessary in order to prove whether the proposed document-oriented model is meeting practical expenses.

The main concept behind XML is to separate the content of a document from its layout (its appearance). The layout is defined by a separate stylesheet language (CSS or XSL(T)). Like the content of a document is separated from its layout, the abstract cellular automaton is separated from a concrete implementation and programming language by a CAML document.

If one wants to use the CA behind the CAML document, one needs to combine the CAML document with an appropriate stylesheet. With this combination, the CAML document will be translated into a 'layout' defined by the stylesheet. This 'layout' is the concrete implementation of the CA. Figure 6 shows the whole environment for CAML documents: The CAML document represents one side of the input to a XSL processor. A XSL processor is a software tool which translates XML documents with respect to a XSL stylesheet. Such a XSL processor is mainly used within web servers to provide XML on web sites for producing HTML

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE CAML SYSTEM "caml.dtd">

<!--
  Game of Life: Assumes, that a dead cell has a 0 and a cell alife has a 1 as state.
-->

<CA  project="gol"  dimensions="2"  min="0"  max="1">

  < !-- ////// RULE 1 /////// -->

  <Rule>
    <!-- A dead cell becomes alive with exactly 3 living neighbors -->

    <Cond>
      <EQ>
        <Current/>
        <Literal value="0"/>
      </EQ>
      <EQ>
        <Sum>
          <Neighbor x="-1" y="-1"/> <Neighbor x="0" y="-1"/> <Neighbor x="1" y="-1"/>
          <Neighbor x="-1" y="0"/>                            <Neighbor x="1" y="0"/>
          <Neighbor x="-1" y="1"/>  <Neighbor x="0" y="1"/>  <Neighbor x="1" y="1"/>
        </Sum>
        <Literal value="3"/>
      </EQ>
    </Cond>

    <Action> <Current> <Literal value="1"/> </Current> </Action>

  </Rule>

  < !-- ////// RULE 2 /////// -->

  <Rule>
    <!-- A living cell keeps alive with 2 or 3 living neighbors, otherwise it dies -->

    <Cond>
      <EQ>
        <Current/>
        <Literal value="1"/>
      </EQ>
      <OR>
        <EQ>
          <Sum>
            <Neighbor x="-1" y="-1"/> <Neighbor x="0" y="-1"/> <Neighbor x="1" y="-1"/>
            <Neighbor x="-1" y="0"/>                            <Neighbor x="1" y="0"/>
            <Neighbor x="-1" y="1"/>  <Neighbor x="0" y="1"/>  <Neighbor x="1" y="1"/>
          </Sum>
          <Literal value="2"/>
        </EQ>
        <EQ>
          <Sum>
            <Neighbor x="-1" y="-1"/> <Neighbor x="0" y="-1"/> <Neighbor x="1" y="-1"/>
            <Neighbor x="-1" y="0"/>                            <Neighbor x="1" y="0"/>
            <Neighbor x="-1" y="1"/>  <Neighbor x="0" y="1"/>  <Neighbor x="1" y="1"/>
          </Sum>
          <Literal value="3"/>
        </EQ>
      </OR>
    </Cond>

    <Action> <Current> <Literal value="1"/> </Current> </Action>

    <Otherwise> <Current> <Literal value="0"/> </Current> </Otherwise>

  </Rule>

</CA>
```

Figure 2: Conway’s *Game of Life* written in CAML

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE CAML SYSTEM "caml.dtd">

<!-- Sierpinsky Triangle (cell-values: 0 = dead, 1 = alive) -->

<CA  project="sierpinsky"  dimensions="1"  min="0"  max="1">

  <Rule>

    <Cond>
      <OR>
        <Pattern dimx="3" centerx="1" > 0 0 0 </Pattern>
        <Pattern dimx="3" centerx="1" > 1 1 1 </Pattern>
      </OR>
    </Cond>

    <Action>    <Current> <Literal value="0"/> </Current> </Action>

    <Otherwise> <Current> <Literal value="1"/> </Current> </Otherwise>

  </Rule>

</CA>
```

Figure 3: The Sierpinsky Triangle written in CAML

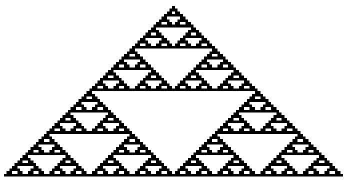

Figure 4: Sierpinsky Triangle

pages. Additionally, a XSL processor can be used autonomously. Therefore, we can use a XSL processor as 'compiler' to translate the CAML document e.g. into programming languages like C. The other input side of this XSL processor has to be the XSL stylesheet. After execution the XSL processor creates the output file, which will be the implementation of the CA. This output file could be compiled with a main-program or be integrated into another software project. This way, even people not familiar with compiler technology can realize a translation of CAML documents simply by defining XSL stylesheets. They can use every XSL processor as 'compiler' and every XML editor as development environment.

Furthermore, a CAML document can be interpreted by software systems. A CAML document is a general representation of a CA. CAML can be used as representation for the exchange of CAs as well as for the description and development of CAs. One representation allows a varied usage of the CA.

4 Conclusion and future work

Within this paper a document-oriented model following the XML-notation (E**x**tensible **M**arkup **L**anguage) was presented that enables the modeling of cellular automata. In this way it is possible to handle CAs with their components at the abstraction-level of building blocks. The document-oriented model applicable by CAML (**C**ellular **A**utomata **M**odeling **L**anguage) enables the modeling as well as the documentation of a CA simultanously and ensures the interchange of CAs modeled once.

```
<!ENTITY % bool       "true | false"      >

<!ENTITY % dispatch_operators
  "(Break | Literal | Min | Max | Add | Sub | Mul | Div | Sum | Neighbor | Current)" >

<!ENTITY % dispatch_conds
  "(EVER | OR | AND | NOT | Pattern | GT | LT | EQ | UNEQ)" >


<!ELEMENT  CA (Rule+)*  >
  <!ATTLIST  CA project     CDATA    "CA" >
  <!ATTLIST  CA dimensions  CDATA    "2" >
  <!ATTLIST  CA min         CDATA    "1.7e-307"  >
  <!ATTLIST  CA max         CDATA    "1.7e+307"  >
  <!ATTLIST  CA discrete   (%bool;)  "true"  >

<!ELEMENT  Rule (Cond+ | Action+ | Otherwise+)* >

<!ELEMENT  Cond (%dispatch_conds;)* >

 <!ELEMENT  EVER         EMPTY                             >
 <!ELEMENT  OR (%dispatch_conds;                    )+ >
 <!ELEMENT  AND (%dispatch_conds;                    )+ >
 <!ELEMENT  NOT (%dispatch_conds;                    ) >
 <!ELEMENT  Pattern     (#PCDATA                      ) >
   <!ATTLIST  Pattern dimx     CDATA  "3" >
   <!ATTLIST  Pattern dimy     CDATA  "0" >
   <!ATTLIST  Pattern dimz     CDATA  "0" >
   <!ATTLIST  Pattern centerx  CDATA  "1" >
   <!ATTLIST  Pattern centery  CDATA  "0" >
   <!ATTLIST  Pattern centerz  CDATA  "0" >
 <!ELEMENT  GT (%dispatch_operators; , %dispatch_operators;) >
 <!ELEMENT  LT (%dispatch_operators; , %dispatch_operators;) >
 <!ELEMENT  EQ (%dispatch_operators; , %dispatch_operators;) >
 <!ELEMENT  UNEQ (%dispatch_operators; , %dispatch_operators;) >

<!ELEMENT  Action (%dispatch_operators;)+ >

<!ELEMENT  Otherwise (%dispatch_operators;)+ >

<!ELEMENT  Break  EMPTY >
<!ELEMENT  Literal  EMPTY >
  <!ATTLIST  Literal value  CDATA  #REQUIRED >
<!ELEMENT  Min  (%dispatch_operators;                     )+ >
<!ELEMENT  Max  (%dispatch_operators;                     )+ >
<!ELEMENT  Add  (%dispatch_operators;                     )+ >
<!ELEMENT  Sub  (%dispatch_operators;                     )+ >
<!ELEMENT  Mul  (%dispatch_operators;                     )+ >
<!ELEMENT  Div  (%dispatch_operators;                     )+ >
<!ELEMENT  Sum  (%dispatch_operators;                     )+ >

<!ELEMENT  Neighbor  EMPTY >
  <!ATTLIST  Neighbor x  CDATA  "0" >
  <!ATTLIST  Neighbor y  CDATA  "0" >
  <!ATTLIST  Neighbor z  CDATA  "0" >

<!ELEMENT  Current  (%dispatch_operators;)? >
```

Figure 5: The Document Type Definition of CAML

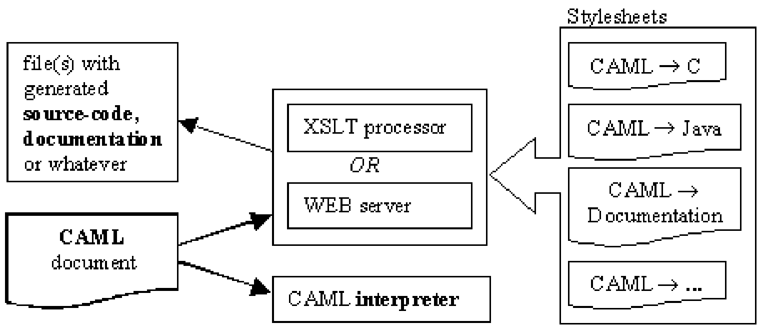

Figure 6: Transforming CAML documents into source code

Up to now, well-known CAs were modeled with CAML. To validate the designed CA documents, a CAML-interpreter was implemented. However, there is still some work to do. For example, it seems to be useful to model movable cells, e.g. for the simulation of particles. For this, every position within the lattice must be able to hold several cells at one time step. Up to now, every cell has only one state. Multiple states per cell could be a nice feature. The possibility of heterogene cells as well, where every cell can have different rules (or rule sets). At present CAML supports only rectangular lattices (e.g. type b, c and d in figure 1). The extension to further lattices would be a nive feature, too. Furthermore, several rule sets, which can be activated alternately, could be an enrichment. And, of course, the definition of mathematical formulars as rules should be possible. For this, it should be examined whether other XML based modeling languages like MathML [7] could be integrated into CAML.

Last but not least, the authors would like to motivate discussions and further contributions for the extension of CAML as well as for standardization and exchange within the CA-community.

5 Acknowledgement

The authors would like to thank Stephanie Wenzel for proof-reading this paper.

References

[1] Henning Behme and Stefan Mintert, *XML in der Praxis: Professionelles Web-Publishing mit der Extensible Markup Language*, Addison-Wesley, Bonn, Deutschland, 1998

[2] The Cellular Automata Simulation System,
URL: http://www.cs.runet.edu/ dana/ca/cellular.html, Dec 22, 2001

[3] N.H. Packard and S. Wolfram, *Two-Dimensional Cellular Automata*, Journal of Statistical Physics, March 1985

[4] C. Veenhuis, K. Franke, M. Köppen, *A Semantic Model for Evolutionary Computation*, Proc. 6th International Conference on Soft Computing (IIZUKA2000), Iizuka, Japan, 2000

[5] C. Veenhuis, *Cellular Automata Modeling Language*, URL: http://vision.fhg.de/ veenhuis/CAML, August 07, 2002

[6] C. Veenhuis, K. Franke, M. Köppen, *Evolutionary Algorithm Modelling Language*, URL: http://vision.fhg.de/ veenhuis/EAML, February 12, 2002

[7] W3C (World Wide Web Consortium), *Mathematical Markup Language (MathML) Version 2.0*, URL: http://www.w3.org/TR/MathML2, Feb 24, 2002

[8] W3C (World Wide Web Consortium), *Extensible Markup Language (XML) 1.0 (Second Edition)*, URL: http://www.w3.org/TR/2000/REC-xml-20001006, Feb 24, 2002

[9] S. Wolfram, *Cellular automata*, Los Alamos Science, Fall 1983