CAML

(**C**ellular **A**utomata **M**odelling **L**anguage)

Version 0.4

Christian Veenhuis

Christian Veenhuis:
CAML (Cellular Automata Modelling Language)

Contents

1 Introduction

This document presents all CAML elements with their definitions in form of a language reference. Thereby, it assumes that the reader is already familiar with Cellular Automata (CA) and the Extensible Markup language (XML). The single elements are categorized to the appropriate groups and presented with the following notation:

< Name of Element

TypeOfAttribute `NameOfAttribute = "` **Default**`"`

>

TypeOfContent `[frequ] <ContentElement>` **Default content** `</ContentElement>`
TypeOfContent `[frequ] [... textual explaination of the content ...]`

</ Name of Element>

Name of Element
The name of the element according to its task e.g. *CA*, *Rule* or *Neighbor*.

Attribute
An element can possess any number of attributes. Beside its name (`NameOfAttribute`) a single attribute possesses a default value (`Default`) and a typing (`TypeOfAttribute`). The default value is taken, whenever this attribute is not defined explicitly. The typing of the attributes represent a special feature. Values, which are used in XML elements as attribute values, are used in form of a character string. A typing, how it is used in CAML (see section 2), is not known by XML (if DTDs are used). This is a weak point: Is an attribute of type `Int`, the author of the CAML description has to be careful that the value of the attribute can be represented by an `Int`. In the very last instance a CAML compiler or interpreter would reject an incorrect used attribute.

Contents
An element can have as many as desired elements as subordinated contents. Everyone of these subordinated elements is an independent element and possesses also its own `Name of Element`. Similar to the attributes it possesses also a content as default (`Default content`) and a typing (`TypeOfContent`). The default content is used, if no other content was explicitly defined. With the typing one has essentially the same situation as with the attributes. The specification `[frequ]` indicates how many of the appropriate elements are allowed to occur. A number for `frequ` means exactly this number of occurences. A `+` means at least one, a `?` zero or one, and an `*` any number of occurences (including 0). Sometimes the content is optionally presented by a simple explaination.

To clarify the notation a sample element with the name `Person` (not existent in CAML) is described:

< Person

String `name REQUIRED`
Int `age = "` **19**`"`
Float `weight = "` **70.0**`"`

>

String [*] <Hobby> " " </Hobby>
String [+] <Residence> " **Berlin**" </Residence>
String [1] <ID> " " </ID>

</ **Person**>

In the description of the element *Person* all valid description elements were used for the definition of the elements (up to the different data types). There are the name of element (*Person*), some attributes and two content elements. The attributes are all typed and up to the name attribute they all possess default values. To assign a name a default value is not meaningful; instead one marks such attributes with the note: REQUIRED. Thus this attribute is marked as mandatory necessary. The content elements are the elements for describing the hobbies if in existance (with no default value), a single ID number, and at least one residence (with `Berlin` as default). A valid *Person* element in accordance with the above description would be:

< **Person** name = " **Meier**" weight = " **82.4**" >

<Hobby> **”Parachuting”** </Hobby>
<Hobby> **”Sailing”** </Hobby>

<Residence> **”Munich”** </Residence>

<ID> **”IDP31415926”** </ID>

</ **Person**>

In the following section all permitted data types of CAML are listed.

2 Data Types

The single attributes of the CAML elements possess a typing. How mentioned in the introduction, such a typing is not provided by XML, if a DTD is used (as in CAML). One has to pay attention for the correct typing while creating a CAML description. On uncertainty a CAML compiler or interpreter can be used for validating the CAML description. In the following table all available types are listed. Beside the type name, a valid literal and a type-describing regular expression can be found in this table.

Type	**Sample**	**Regular Expression / Explanation**
Bool	true	`true|false`
String	”This is a string”	`\"[^\"\n]*\"`
Long	1024	`[0-9]+`
Float	3.1415	`[0-9]+|([0-9]*\.[0-9]+)`
Condition	<OR> ...further nested elements... </OR>	One of the condition elements
Operator	<Current />	One of the operator elements
Free	<Rule> ...further nested elements... </Rule>	Some elements have no type e.g. the rule elements

Starting from the following section all CAML elements are categorized according to their usage within a CAML description. Each individual CAML element is described with the notation as presented in the introduction.

3 Root element: CA

The root element is situated on the highest level and encloses all other CAML elements. With its attributes one can determine the number of dimensions, the range for the cell states, and the project name.

< CA
String `project = "` **CA**`"`
Long `dimensions = "` **2**`"`
Float, Long `min = "` **1.7e-307**`"`
Float, Long `max = "` **1.7e+307**`"`
Bool `discrete = "` **true**`"`
>
Free `[+] <Rule> </Rule>`
</ CA>

Attribute: project
The content of this attribute serves as base name e.g. for the created source code files generated by a CAML compiler or as project name for a CAML interpreter.

Attribute: dimensions
The number of dimensions can be specified with this attribute (1D, 2D, 3D etc.).

Attribute: min
The lower value of the interval for valid cell states. An underrun clamps the cell state to `min`. This parameter has several types. It can take on values of type `Float` and `Long`.

Attribute: max
The upper value of the interval for valid cell states. An overrun clamps the cell state to `max`. This parameter has several types. It can take up values of type `Float` and `Long`.

Attribute: discrete
This attribute specifies whether the values of the cell states are discrete or continuous.

Content: Rule
Every CA possesses at least one Rule element. These Rule elements need no typing.

4 Elements for Rules

CAML provides the following elements for defining the rules of the CA.

4.1 Rule

A single rule can be defined with the following element. Thereby a rule contains conditions and operators for determining the new state of the cell. It has no attributes.

< Rule>
Free `[+] <Cond> </Cond>`
Free `[*] <Action> </Action>`
Free `[*] <Otherwise> </Otherwise>`
</ Rule>

Content: Cond
Every rule needs some conditions which specifies whether the rule is active or not. A rule can have any number of `Cond` elements. But only one `Cond` element needs to be active to activate the rule. A `Cond` element contains any number of CAML elements of the type `Condition`.

Content: Action
The operations (actions) which have to be executed if the rule is active are defined within this element. This element contains any number of CAML elements of the type `Operator`.

Content: Otherwise
The operations (actions) which have to be executed if the rule is **not** active are defined within this element. This element contains any number of CAML elements of the type `Operator`.

4.2 Cond

Every rule should possess at least one condition for determining whether the rule is active or not. For this any number of conditions can be defined within the *Cond* element. If every of these conditions is **true**, the *Cond* element is set to active.

< Cond>
Condition `[*] ...every of the condition elements...`
</ Cond>

Content
This element contains any number of CAML elements of type `Condition` as presented in section 5. The meaning of the single elements is presented in the appropriate sub-sections.

4.3 Action

Every rule should possess at least one *Action* element for changing the cell state if the rule is active. For this any number of operators can be defined within the *Action* element.

< Action>
Operator `[*] ...every of the operator elements...`
</ Action>

Content
This element contains any number of CAML elements of type `Operator` as presented in section 6. The meaning of the single elements is presented in the appropriate sub-sections.

4.4 Otherwise

For changing the cell state if the rule is **not** active, a rule could possess an *Otherwise* element. Any number of operators can be defined within the *Otherwise* element.

< Otherwise>
Operator `[*] ...every of the operator elements...`
</ Otherwise>

Content
This element contains any number of CAML elements of type `Operator` as presented in section 6. The meaning of the single elements is presented in the appropriate sub-sections.

5 Elements for Conditions (type: Condition)

Every rule possesses conditions for determining whether the rule is active or not. The definition of conditions is done by using the following CAML element in a hierarchical manner.

5.1 EVER

This element returns **true** with every call. Every *Cond* element which consists only of this element returns **true**, too. This way the rule will be active for ever and is executed every time.

< EVER>
</ EVER>

This element possesses no attributes and no content.

5.2 OR

The logical **or** function on the results of any number (at least one) of condition elements can be realized with this element.

< OR>
Condition `[+] ...every of the condition elements...`
</ OR>

Content
This element contains one or more CAML elements of type `Condition`.

5.3 AND

The logical **and** function on the results of any number (at least one) of condition elements can be realized with this element.

< AND>
Condition `[+] ...every of the condition elements...`
</ AND>

Content
This element contains one or more CAML elements of type `Condition`.

5.4 NOT

The logical **not** function on the result of one condition element can be realized with this element.

< NOT>
Condition `[1] ...every of the condition elements...`
</ NOT>

Content
This element contains one CAML element of type `Condition`.

5.5 Pattern

Sometimes one wants to specify the occurence of a specific pattern directly. This element allows the definition of any pattern with wildcards and any centered cell position.

< Pattern
Long `dimx = "` **3**`"`
Long `dimy = "` **0**`"`
Long `dimz = "` **0**`"`
Long `centerx = "` **1**`"`
Long `centery = "` **0**`"`
Long `centerz = "` **0**`"`
>
Free `[+] ...the vector or matrix of values...`
</ Pattern>

Attribute: dimx
The size of the specified pattern on the x-axis.

Attribute: dimy
The size of the specified pattern on the y-axis.

Attribute: dimz
The size of the specified pattern on the z-axis.

Attribute: centerx
The position of the center of this pattern on the x-axis.

Attribute: centery
The position of the center of this pattern on the y-axis.

Attribute: centerz
The position of the center of this pattern on the z-axis.

Content
The content consists of the pattern which one wants to specify. E.g. one wants to define that whenever the one dimensional pattern `111` occurs the cell has to become `1`. The cell considered shall be the center of this pattern (the midst 1). This means that every cell being 1 having two neighbors of symbol 1 shall be recognized. This could be described by the following *Pattern* element:

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

The attribute `dimx` specifies a length of 3 digits, whereby the mid position shall be the 2nd digit at position 1 (`centerx` starts counting at 0).

Another example searchs for cells being 0 and having a 1 as left neighbor and any symbol at the right side:

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

The * symbol is a wildcard which marks any digit at the appropriate position. Of course, also two dimensional pattern are possible:

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

This pattern recognizes two dimensional patterns with a 0 in the center and the digit 1 in every corner. On the left, right, top, and bottom side any digit is expected.

5.6 GT

The relational **greater than** operator (>) on the results of two operator elements can be realized with this element. It returns true, iff: first operator > second operator.

```
< GT>
   Operator [2] ...every of the operator elements...
</ GT>
```

Content
As binary operator this element contains exactly two CAML elements of type `Operator`.

5.7 LT

The relational **less than** operator (<) on the results of two operator elements can be realized with this element. It returns true, iff: first operator < second operator.

< LT>
Operator `[2] ...every of the operator elements...`
</ LT>

Content
As binary operator this element contains exactly two CAML elements of type `Operator`.

5.8 EQ

The relational **equal** operator (=) on the results of two operator elements can be realized with this element. It returns true, iff: first operator = second operator.

< EQ>
Operator `[2] ...every of the operator elements...`
</ EQ>

Content
As binary operator this element contains exactly two CAML elements of type `Operator`.

5.9 UNEQ

The relational **unequal** operator ($\neq$) on the results of two operator elements can be realized with this element. It returns true, iff: first operator $\neq$ second operator.

< UNEQ>
Operator `[2] ...every of the operator elements...`
</ UNEQ>

Content
As binary operator this element contains exactly two CAML elements of type `Operator`.

6 Elements for Operators (type: Operator)

For the use within conditions and the action and otherwise parts of a rule, operators and operator hierarchies can be specified. For this the following CAML elements can be used.

6.1 Break

All rules are processed in the order of their occurrence within the CAML document. This element is used if one wants to skip all following rules.

< Break>
</ Break>

It possesses no content.

6.2 Literal

This element is used to specify concrete numbers. E.g., the examination whether the result of an operator hierarchy is greater than 5 could be done with the following CAML elements:

< GT>
```
  <an operator hierarchy> ..... </an operator hierarchy>
  <Literal value="5" />
```
</ GT>

The *Literal* element is defined as:

< Literal
Long, Float `value = REQUIRED`
>
</ Literal>

Attribute: value
The concrete number. It is multiple typed with long and float.

6.3 Min

This element determines and returns the minimum of all given operators and operator hierarchies.

< Min>
Operator `[+] ...every of the operator elements...`
</ Min>

Content
This element contains at least one CAML element of type `Operator`.

6.4 Max

This element determines and returns the maximum of all given operators and operator hierarchies.

< Max>
Operator `[+] ...every of the operator elements...`
</ Max>

Content
This element contains at least one CAML element of type `Operator`.

6.5 Add

This element adds together the results of all given operators and operator hierarchies ($operator_1 + operator_2 + \ldots + operator_n$).

< Add>
Operator `[+] ...every of the operator elements...`
</ Add>

Content
This element contains at least one CAML element of type `Operator`.

6.6 Sub

This element subtracts the results of all given operators and operator hierarchies from the first operator ($operator_1 - operator_2 - \ldots - operator_n$).

< Sub>
Operator `[+] ...every of the operator elements...`
</ Sub>

Content
This element contains at least one CAML element of type `Operator`.

6.7 Mul

This element multiplies the results of all given operators and operator hierarchies ($operator_1 * operator_2 * \ldots * operator_n$).

< Mul>
Operator `[+] ...every of the operator elements...`
</ Mul>

Content
This element contains at least one CAML element of type `Operator`.

6.8 Div

This element divides the results of all given operators and operator hierarchies from the first operator ($operator_1 / operator_2 / \ldots / operator_n$).

< Div>
Operator `[+] ...every of the operator elements...`
</ Div>

Content
This element contains at least one CAML element of type `Operator`.

6.9 Sum

This element summarizes the results of all given operators and operator hierarchies ($\sum_{i=0}^{n} operator_i$).

< Sum>
Operator `[+] ...every of the operator elements...`
</ Sum>

Content
This element contains at least one CAML element of type `Operator`.

6.10 Neighbor

A fundamental aspect of Cellular Automata are the usage of the neighboring cells for determining the new cell state. With this element one can get the state of one of the neighbor cells. For this an offset can be specified relative to the current cell. This way any range for neighbors can be realized.

< Neighbor
Long `x = "0"`
Long `y = "0"`
Long `z = "0"`
>
</ Neighbor>

Attribute: x
The distance to the desired neighbor-cell on the x-axis.

Attribute: y
The distance to the desired neighbor-cell on the y-axis.

Attribute: z
The distance to the desired neighbor-cell on the z-axis.

6.11 Current

This element returns the state of the current considered cell.

< Current>
Operator `[?] ...every of the operator elements...`
</ Current>

Content
This element contains one or no CAML element of type `Operator`. If this element contains one CAML element, the result of this given CAML element is assigned to the state of this current cell.

7 Samples

This section presents some simple examples of CAML documents.

7.1 Game of Life

Conways well-known *Game of Life* realizes the following rules:

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 dies)

Figure 1 depicts the appropriate CAML document. Their are two *Rule* elements (one for every rule described above). The first rule is active if the cell is dead (current equals 0) and the number of living neighbors is 3 (sum of neighbors equals 3). If the first rule is active the current cell is set to alive (current is set to literal 1). The second rule is active if the cell is alive (current equals 1) and the number of living neighbors is 2 or 3 (sum of neighbors equals 3 OR sum of neighbors equals 2). If this second rule is active the current cell is set to alive (1) otherwise to dead (0).

7.2 Sierpinsky Triangle

The Sierpinsky triangle is a one dimensional CA which determines the states with the following table:

Pattern	New State
000	0
001	1
010	1
011	1
100	1
101	1
110	1
111	0

In figure 2 one possible CAML description for this table is given. It uses the fact that every pattern containing 1 or 2 symbols of 1 generates the symbol 1 as result. The left and right neighbor as well as the current state are totalized. If this sum equals 1 or equals 2 the new state is 1 otherwise 0.

In figure 3 another possibility of describing the table is presented. By using the *Pattern* element the both patterns which produce a 0 as new state are described. If one of these patterns occur, a 0, otherwise a 1, is assigned to the state of the current cell.

7.3 Mean Filter

In image processing tasks various filters are used. One among them is the mean filter which replaces a pixel value by the average value of all 8 neighbor pixels inclusive the current pixel. This filter smoothes a given image.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE CA 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 1: Conway's *Game of Life* written in CAML

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

<!--
  Sierpinsky triangle
  Assumes, that every dead cell has a 0 and every cell alife has
  a 1 as value.
-->

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

  <Rule>

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

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

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

  </Rule>

</CA>
```

Figure 2: Sierpinsky triangle written in CAML

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

<!--
  Sierpinsky triangle
  Assumes, that every dead cell has a 0 and every cell alife has
  a 1 as value.
-->

<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: Another Sierpinsky triangle written in CAML

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

<CA  project="mean"  dimensions="2"  min="0"  max="255" >

  <Rule>

    <Cond>
      <EVER />
    </Cond>

    <Action>
      <Current>
        <Div>
          <Sum>
            <!-- ...get all eight neighbors... -->
            <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"/>
            <Current />   <!-- and include the current cell -->
          </Sum>
          <Literal value="9"/>
        </Div>
      </Current>
    </Action>

  </Rule>

</CA>
```

Figure 4: Mean filter written in CAML

In figure 4 such a filter is realized as CA: All eight neighbors are added to the current cell. After this the value is divided by 9. The condition uses the element *EVER*, whereby the rule is applied to every cell. This CA uses gray-value pixels as states for the cells (`min="0" max="255"`). By using CAML this way, various image processing filters can be realized by a CA.