Cafun Simulation File Format Specification

Version 1.0

Table of Contents

This document describes the formal requirements any valid Cafun Simulation file has to comply with. It introduces Cafun Simulations as an application of the Extensible Markup Language, or XML for short, and explains the roles and constraints of the individual elements and attributes. It is intended to support the understanding of existing simulations or to serve as a reference for writing custom simulations. Basic knowledge of cellular automata and the Extensible Markup Language is assumed.

1. Introduction

1.1. Concept

A Cafun Simulation file contains all information for simulating a specific cellular automaton. Most file formats describe cellular automata as a set of states and rules. However, Cafun Simulations pursue a similar but object oriented approach. In Cafun each cell is an instance of a certain cell type, which decides about its look and behavior. The latter arises from a set of mutations that is assigned to the cell type. In short, cell types take over the role of states and mutations take over the role of rules. According to the object oriented paradigm a cell corresponds to an object and a cell type corresponds to a class. Besides, Cafun Simulations offer so called “abstract cell types”, which correspond to abstract classes or interfaces. Abstract cell types enable the construction of inheritance relationships between cell types.

The use of the object oriented paradigm is supposed to ease the formulation of complex cellular automata. Defining behavior in the context of cell types makes it obvious which behavior options cells of a certain cell type have. With the help of abstract cell types shared behavior of different cell types can be defined centrally and - as the name suggests - even with ignorance of certain details. It was proven that the file format of Cafun Simulations allows the description of many different kinds of cellular automata. While before each kind of cellular automata required its own file format, a uniform file format can be used now.

In Cafun a simulation always takes place on a two-dimensional lattice of cells, which is called “universe”. For this reason, a Cafun Universe unifies all cell states at a certain time. Different from other file formats of cellular automata Cafun Simulations do not store any cell states. Cafun Universes are stored independently from Cafun Simulations to give the user the opportunity to perform one simulation with different universes. That means that a Cafun Simulation file only contains information that does not change during simulation.

1.2. XML

The foundation of the Cafun Simulation file format is the so called “Extensible Markup Language”, or XML for short. XML has the advantage of being standardized and widespread and that it can be used on practically all computer platforms. Besides it is relatively easy to be read and written by humans. XML describes just basic rules for the markup of structured content. The specifics have to be defined individually for each application. In case of the Cafun Simulation file format, the specifics were designed with the intention of being easy to comprehend and to be written manually.

The specification of the Cafun Simulation file format is carried out in the style of XML Document Type Definitions, or DTDs for short. Document Type Definitions are an established standard for defining structural constraints of XML files. However, there are constraints to valid Cafun simulations that cannot be expressed by Document Type Definitions. Additional constraints or requirements that cannot be expressed by Document Type Definitions are mentioned in the comments of the respective elements and attributes they apply to.

2. Basic Definitions

2.1. Character Encoding

Cafun Simulation files must be encoded in the UTF-8 character encoding scheme. UTF-8, which is XML's default encoding scheme, covers the characters of many languages of the world and can be edited by most modern text editors. Cafun assumes any Cafun Simulation file to be encoded in UTF-8, no matter whether a different encoding was specified in the XML declaration.

2.2. Namespaces

Cafun does not handle XML namespaces. While it is allowed to include elements of foreign namespaces, for what reason whatsoever, the elements and attributes of this specification always must be defined in the default namespace.

2.3. Schema Languages

There are plenty of schema languages available for defining constraints of XML files. The most popular ones are Document Type Definition (ord DTD for short), XML Schema and Relax NG, to name just a few. However, at the time of writing this specification there is no known XML schema language that can cope with all the contraints that apply to Cafun Simulations.

Internally, Cafun uses its own validation scheme, so there's no official schema to use for validating Cafun Simulations. Generally, it is discouraged to include references to any schema in a Cafun Simulation file, even if it is performed via the standard <!DOCTYPE> declaration. Cafun Simulations are intended to be self-contained. Including a reference to a schema, however, can make a XML file dependend on it. This, for example, happens when a XML files contains custom entities defined in a DTD. If, nevertheless, a schema is referenced in a Cafun Simulation file it is ignored and not used for validation.

Discouraging referencing schemas does not mean that they're discouraged to be used for validation. Even if a schema fails to cover all constraints of Cafun Simulations it might be better to be able to validate some constraints than none at all. As far as available, the use of schemas is encouraged, since many XML processors provide functionality to validate a XML document by explicitly specifying a schema without the need to include a reference in the XML file.

2.4. Cafun Color Format

All attributes in a Cafun Simulation file that refer to a color have to meet a uniform color format. In this format, a color is expressed by three color components, red, green and blue. The proportion of each component is measured as an integer number from 0 to 255 and specified in decimal notation. This corresponds to the RGB color representation of most image editing programs. A valid Cafun Color Format value consists of the three components written one after the other, each of them separated by a single blank. The colors black and white, for example, are written as “`0 0 0`” and “`255 255 255`”, respectively. The color yellow, which is composed of the components red and green to equal proportions, is written as “`255 255 0`”.

2.5. Element `simulation`

Every Cafun Simulation file contains exactly one `simulation` element. The `simulation` element wraps the whole definition of the simulation.

```
<!ELEMENT simulation (description?, (cell-type |
          abstract-cell-type)*, chart?, make-up?)>
```

2.5.1. Attribute `name`

The attribute `name` gives the simulation a name.

```
<!ATTLIST simulation name CDATA #REQUIRED>
```

2.5.2. Attribute `author`

The attribute `author` gives the author the opportunity to introduce himself. The value is supposed to contain merely the full name of the author and no other information, like the email address for example.

```
<!ATTLIST simulation author CDATA #IMPLIED>
```

2.6. Element `description`

The element `description` summarizes the description of the simulation. A description can be subdivided into several sections which are represented by `section` elements below the `description` element.

```
<!ELEMENT description (section*)>
```

2.7. Element `section`

The text of a description section is marked up with a `section` element. The text inside a `section` element is supposed to be specified as plain text, i. e. without any formattings of markup languages like HTML for example.

```
<!ELEMENT section (#PCDATA)>
```

2.7.1. Attribute `caption`

The optional attribute `caption` assigns a caption to a description section.

```
<!ATTLIST section caption CDATA #IMPLIED>
```

3. Cell Types

3.1. Element `cell-type`

Each cell in a Cafun universe is an instance of a cell type, which determines its color and how it behaves in the context of other cells. For every cell type in a Cafun Simulation there exists exactly one `cell-type` element which wraps its definition.

This kind of cell types is also called “concrete” cell types to distinguish it from so called “abstract” cell types which represent common behavior shared by multiple cell types. A concrete cell type can implement up to 16 abstract cell types, each of which is represented by an `implementation` element below its `cell-type` element.

The unique behavior of a cell type is defined by a set of `mutation` elements which are subordinated to the `cell-type` element.

```
<!ELEMENT cell-type (implementation*, mutation*)>
```

3.1.1. Attribute `id`

Each cell type in a Cafun Simulation has a unique id. With its id a cell type can be referenced elsewhere in the simulation. The id of a cell type is specified by the attribute `id` which has to comply with the following requirements:

- The id must start with a capital letter.
- The id may only contain the upper case letters `A-Z` and the lower case letters `a-z` of the English alphabet, as well as the numerals `0-9` and the special characters `-` and _.
- The id may be at most 256 characters long.

```
<!ATTLIST cell-type id ID #REQUIRED>
```

3.1.2. Attribute `color`

The color in which the cells of a cell type are drawn is specified by the attribute `color`. Its value has to comply with the Cafun Color Format. Since the cell types of cells in a Cafun Universe are identified by means of colors, colors of different cell types must differ from each other.

```
<!ATTLIST cell-type color CDATA #REQUIRED>
```

3.1.3. Attribute `active`

If the `active` attribute of a cell type is set to `true`, its cells are designated as “active”. If it is set to `false`, its cells are designated as “passive”. At simulation time passive cells are evaluated only if there is an active cell in their neighborhood. Active cells are evaluated principally always.

Since the evaluation of cells consumes time, the simulation speed can be increased if you set the `active` attribute of certain cell types to `false`. However, this should be done with special caution. The `active` attribute of a cell type is supposed to be set to `false` only if it is assured that its cells being active or passive has no affect on the course of the simulation.

```
<!ATTLIST cell-type active (true | false) "true">
```

3.2. Element `abstract-cell-type`

Besides “concrete” cell types there is another kind of cell types, so called “abstract” cell types, which are specified by `abstract-cell-type` elements. Abstract cell types represent common behavior shared by multiple cell types, that's why they cannot be instantiated directly. Accordingly, abstract cell types define no look but only behavior. Just like concrete cell types, the behavior of abstract cell types is specified by a set of `mutation` elements, subordinated to the `abstract-cell-type` element. Abstract cell types can be implemented by concrete cell types, whereby the concrete cell types inherit the mutations of the abstract ones they implement.

Although cells are always instances of just one concrete cell type, they are also regarded as instances of all the abstract cell types their actual cell type implements. This is especially instrumental in referencing cell types, for example in the definition of mutation conditions. Thus, many different concrete cell types can be referenced by one commonly implemented abstract cell type.

In the identification of commonalities individual details often must be abstracted away. Therefore, abstract cell types give the possibility to express references to certain cell types symbolically. In Cafun Simulations such symbolic references are called “abstractions”. Each abstraction used within the definition of an abstract cell type must be declared by an subordinated `abstraction` element.

```
<!ELEMENT abstract-cell-type (abstraction*, mutation*)>
```

3.2.1. Attribute `id`

Just like every “concrete” cell type, every abstract cell type has a unique id, which enables it to be referenced elsewhere in the simulation. The id is specified by the attribute `id` and must comply with the following requirements:

- The id must start with a lowercase letter. Thus, abstract cell types can be distinguished from concrete cell types by the first letter of their id.
- The id may only contain the upper case letters `A-Z` and the lower case letters `a-z` of the English alphabet, as well as the numerals `0-9` and the special characters `-` and _.
- The id may be at most 256 characters long.

```
<!ATTLIST abstract-cell-type id ID #REQUIRED>
```

3.3. Element `abstraction`

Abstractions are symbolic references which are used inside definitions of abstract cell types instead of actual references to existing cell types. Before an abstraction may be used it has to be declared by an `abstraction` element. If a concrete cell type implements an abstract cell type, it has to specify an existing cell type for every abstraction declared in the `abstract-cell-type` element.

```
<!ELEMENT abstraction EMPTY>
```

3.3.1. Attribute `id`

The attribute `id` specifies the name of the abstraction. This name must be unique within the scope of the abstract cell type the abstraction applies for and comply with the following requirements:

- The id must start with a dollar sign `$`. Thus, abstractions can be distinguished from actual cell types references by the first letter of their id.
- The id may only contain the upper case letters `A-Z` and the lower case letters `a-z` of the English alphabet, as well as the numerals `0-9` and the special characters `-` and _.
- The id may be at most 256 characters long.

```
<!ATTLIST abstraction id CDATA #REQUIRED>
```

3.4. Element `implementation`

An `implementation` element makes a concrete cell type implement a certain abstract cell type. A concrete cell type contains exactly one `implementation` element for each abstract cell type it implements.

If the abstract cell type which is implemented by an `implementation` element declares any abstractions, exactly one `concretion` element must be subordinated to the `implementation` element for each of these abstractions.

```
<!ELEMENT implementation (concretion*)>
```

3.4.1. Attribute `cell-type`

The attribute `cell-type` specifies the abstract cell type to be implemented by the `implementation` element. Its value has to equal the id of an abstract cell type defined in the simulation.

```
<!ATTLIST implementation cell-type CDATA #REQUIRED>
```

3.5. Element `concretion`

If an abstract cell type declares any abstractions these must be put in concrete form at implementation. This is done by specifying a `concretion` element for every abstraction the abstract cell type declares. A single `concretion` element defines which cell type to substitute for a certain abstraction.

Principally both concrete cell types and abstract cell types can be specified with `concretion` elements. However, special circumstances have to be considered when specifying abstract cell types. Within the definition of an abstract cell type an abstraction can be referenced at sites, where only concrete cell types are allowed. Such a site, for example, is the attribute `cell-type` of a `mutation` element. Here the abstraction may only be substituted by concrete cell types, the specification of an abstract cell type results in an error message.

```
<!ELEMENT concretion EMPTY>
```

3.5.1. Attribute `abstraction`

The attribute `abstraction` determines which abstraction the `concretion` element puts in concrete form. Its value must equal the id of the corresponding abstraction, including the prefixed dollar sign.

```
<!ATTLIST concretion abstraction CDATA #REQUIRED>
```

3.5.2. Attribute `cell-type`

The attribute `cell-type` determines which cell type to substitute for the abstraction. Its value must equal the id of a concrete or an abstract cell type defined in the simulation.

```
<!ATTLIST concretion cell-type CDATA #REQUIRED>
```

4. Mutations

4.1. Element `mutation`

During the simulation of cellular automata the cells change their state stepwisely. In Cafun cells don't change their state but their cell type, which is called “mutation”. A cell of a certain cell type can only mutate into certain cell types, as defined by a set of `mutation` elements subordinated to its `cell-type` element. Each `mutation` element represents a possible mutation into one cell type.

Mutations can be dependent on certain conditions, which are defined by `condition` elements subordinated to the `mutation` element. Only if a cell complies to all conditions of a mutation it can change its cell type accordingly. If a change into a certain cell type is supposed to occur under different conditions which exclude each other mutually, it can be defined by multiple `mutation` elements each of which with individual conditions.

The change of their cell types is the only behavior option cells in a Cafun Simulation have, which means that the behavior of one cell type is completely determined by the set of mutations subordinated to it (and the abstract cell types it implements). The task in the design of Cafun Simulations is to define mutations in a way that they give rise to complex behavior.

```
<!ELEMENT mutation (condition*)>
```

4.1.1. Attribute `cell-type`

The attribute `cell-type` specifies into which cell type a cell changes in the mutation. Because cells are primarily instances of concrete cell types, its value must equal the id of a concrete cell type defined in the simulation.

```
<!ATTLIST mutation cell-type CDATA #REQUIRED>
```

4.1.2. Attribute `priority`

A cell type can have multiple mutations assigned, whose conditions overlap each other. In such a case it is ambiguous which mutation takes place, i. e. into which cell type a cell changes. If several mutations compete with each other in this way, the attribute `priority` allows to specify exactly which mutation to prefer. Thereby, mutations with higher priorities are preferred principally. However, chance decides what mutation takes place if two mutations with equal priorities compete with each other.

```
<!ATTLIST mutation priority (top | very-high | high | medium |
          default | low | very-low | lowest) "default">
```

4.1.3. Attribute `probability`

To simplify the simulation of phenomena with statistical properties, mutations can be influenced by chance. The attribute `probability` defines the probability that a mutation takes place if a cell complies to its conditions.

If a cell complies to the conditions of multiple mutations, the evaluation of probabilities is performed always before the selection of a mutation by means of the priorities. The probability that a mutation takes place at a certain cell is independent in the mathematical sense from whether the same mutation took place before at another cell.

The value of the attribute must be stated as a decimal number between `1.0` and `0.0`, whereas `1.0` corresponds to the certain probability and `0.0` corresponds to the impossible probability. The decimal separator is the dot `.`.

```
<!ATTLIST mutation probability CDATA "1.0">
```

4.2. Element `condition`

A `condition` element defines a certain condition a mutation depends on. As usual with cellular automata the behavior of a cell in a Cafun Universe is only affected by the eight neighbor cells that surround it. That's why conditions always relate to the presence or absence of certain cell types in the neighborhood of a cell.

Strictly speaking, a `condition` element states how many cells of a certain cell type must be present minimally or may be present maximally in the neighborhood of a cell. In the evaluation of a condition both the concrete cell types of the neighbor cells as well as the abstract cell types they implement are taken into account.

```
<!ELEMENT condition EMPTY>
```

4.2.1. Attribute `cell-type`

The attribute `cell-type` defines the cell type the condition relates to. Its value must equal the id of a concrete or abstract cell type defined in the simulation.

```
<!ATTLIST condition cell-type CDATA #REQUIRED>
```

4.2.2. Attribute `min`

The attribute `min` defines the minimum count of neighbor cells that have to be instances of the specified cell type.

```
<!ATTLIST condition min (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8) "0">
```

4.2.3. Attribute `max`

The attribute `max` defines the maximum count of neighbor cells that may be instances of the specified cell type.

```
<!ATTLIST condition max (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8) "8">
```

4.2.4. Attribute `scope`

The attribute `scope` specifies which neighbor cells are evaluated for the condition. The selection is based on the point of the compass. For example, the upper left neighbor cell is selected by the direction `north-west`. The value must be a list of the following key words, each of them separated by exactly one blank: `north-west`, `north`, `north-east`, `east`, `south-east`, `south`, `south-west`, `west`. This attribute is optional. The condition relates to all neighbor cells if it is omitted.

```
<!ATTLIST condition scope CDATA #IMPLIED>
```

5. Chart

5.1. Element `chart`

Cafun offers functionality to record the quantitative development of certain cell types over simulation time. Therefore, the count of their respective cells is measured in every simulation step and displayed as a chart.

All settings regarding the recording and display of the chart are summarized below the `chart` element. The recording and display of the chart is performed only if the `chart` element is defined. For every cell type that is to be recorded an `indicator` element is subordinated to the `chart` element. Altogether, up to four `indicator` elements may be subordinated to the `chart` element.

```
<!ELEMENT chart (indicator+)>
```

5.2. Element `indicator`

A single `indicator` element represents a certain cell type whose development is to be recorded.

```
<!ELEMENT indicator EMPTY>
```

5.2.1. Attribute `cell-type`

The attribute `cell-type` determines which cell type is to be recorded. Its value must equal the id of an existing cell type defined in the simulation. The specification of both concrete and abstract cell types is allowed. If an abstract cell type is specified the count of all those cell types is measured that implement the abstract one.

```
<!ATTLIST indicator cell-type CDATA #REQUIRED>
```

5.2.2. Attribute `color`

The attribute `color` determines the color the recording of the cell type appears in. Its value has to comply with the Cafun Color Format, whereas `color` attributes of different `indicator` elements have to differ from each other.

```
<!ATTLIST indicator color CDATA #REQUIRED>
```

6. Make-Up View

6.1. Element `make-up`

Cafun allows to define a special “make-up view” of the universe which is intended to make simulations visually more appealing. In the normal view universes are pictured as lattices of quadratic cells, whereas each cell appears in the color of its cell type. In the make-up view certain cell types appear in alternative colors and optionally, a graphical filter is applied that alienates the view with a special visual effect.

All settings regarding the make-up view are summarized below the `make-up` element. The make-up view is available only if the `make-up` element is defined.

The `make-up` element wraps a set of so called “looks” which assign alternative colors to certain cell types. The cell types being dyed by a look are specified by a set of `recipient` elements, with each `recipient` element designating a single cell type. If a cell type is accidentally referred to by multiple `recipient` elements only the last one is considered.

```
<!ELEMENT make-up (common-look | gradient-look)*>
```

6.1.1. Attribute `filter`

The attribute `filter` determines which filter to apply to alienate the view. If this attribute is omitted or if its value is `none` no filter is applied. The following values are allowed:

`none`	No filter is applied.
`slight-blur`	The view is blurred slightly.
`heavy-blur`	The view is blurred heavily.
`shallow-raised-relief`	The view appears as a shallow raised relief.
`deep-raised-relief`	The view appears as a deep raised relief.
`shallow-sunken-relief`	The view appears as a shallow sunken relief.
`deep-sunken-relief`	The view appears as a deep sunken relief.

`edge`	The edges of the view are emphasized.
`sharpen`	The view is sharpened.
`speed`	The view appears distorted as if the camera is moved.

```
<!ATTLIST make-up filter (none | slight-blur | heavy-blur |
          shallow-raised-relief | deep-raised-relief |
          shallow-sunken-relief | deep-sunken-relief |
          edge | sharpen | speed) "none">
```

6.2. Element `common-look`

The element `common-look` assigns a common color to a set of cell types in the make-up view. Each cell type belonging to the set is represented by a `recipient` element below the `common-look` element.

```
<!ELEMENT common-look (recipient*)>
```

6.2.1. Attribute `color`

The attribute `color` determines the color which is assigned by a `common-look` element. Its value has to comply with the Cafun Color Format.

```
<!ATTLIST common-look color CDATA #REQUIRED>
```

6.3. Element `gradient-look`

The element `gradient-look` assigns a color gradient to a set of cell types in the make-up view. Each cell type belonging to the set is represented by a `recipient` element below the `gradient-look` element.

From a start color to an end color, each referenced cell type is assigned exactly one color of the gradient. The cell types of the first and the last `recipient` element receive the start and end color, respectively, while cell types of intermediate `recipient` elements receive a blend of both colors. The proportion of the blend depends on the position of an intermediate `recipient` element in relation to the first and the last `recipient` element. The impact of the start or end color becomes stronger, the nearer a `recipient` element is positioned to it. The fineness of the color gradient increases with the number of `recipient` elements.

```
<!ELEMENT gradient-look (recipient*)>
```

6.3.1. Attribute `start-color`

The attribute `start-color` determines the start color of the color gradient which is assigned by a `gradient-look` element. Its value has to comply with the Cafun Color Format.

```
<!ATTLIST gradient-look start-color CDATA #REQUIRED>
```

6.3.2. Attribute `end-color`

The attribute `end-color` determines the end color of the color gradient which is assigned by a `gradient-look` element. Its value hast to comply with the Cafun Color Format.

```
<!ATTLIST gradient-look end-color CDATA #REQUIRED>
```

6.4. Element `recipient`

The element `recipient` designates a cell type as a recipient of a certain look. It causes the referenced cell type to be dyed according to the superordinated `common-look` or `gradient-look` element in the make-up view.

Since only concrete cell types can be instantiated and therefore displayed, only those are allowed to be defined as recipients of looks. The reference of an abstract cell type results in an error message.

```
<!ELEMENT recipient EMPTY>
```

6.4.1. Attribute `cell-type`

The attribute `cell-type` determines the cell type which is defined as a recipient of a look. Its value must equal the id of a concrete cell type defined in the simulation.

```
<!ATTLIST recipient cell-type CDATA #REQUIRED>
```