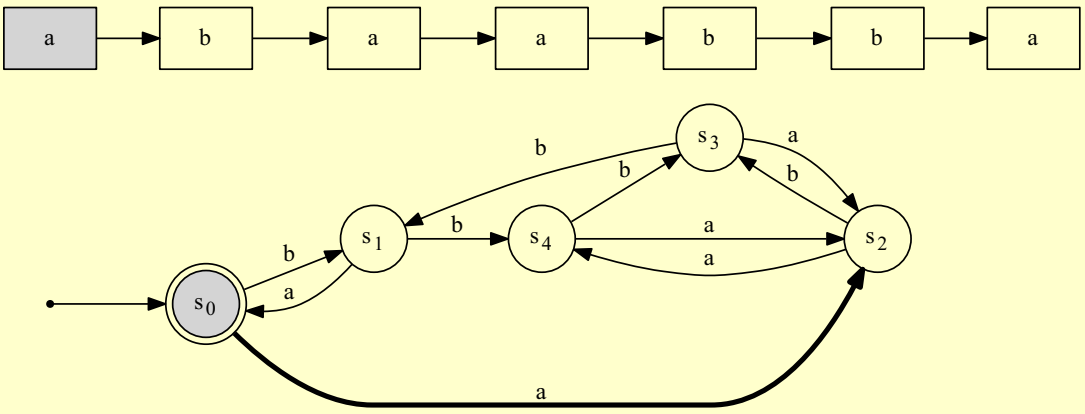
XWizard: The Online Informatics Toolbox

– Handbook for Teachers –

Lukas König, Friederike Pfeiffer-Bohnen



Script ID-10700

XWizard: The Online Informatics Toolbox

– Handbook for Teachers –

Contents

1 What is XWizard? 3

2 Access and Short History 3

3 Basic Workflow: Script Processing 4

4 Conversion Methods 7
4.1 Conversion methods which create a new script 8
4.2 Conversion methods which create a plain text output 9
4.3 In-Script Application of Conversion Methods 10

5 The Exercise Mode and Encrypted Scripts 11
5.1 Creating an exercise 11

6 Hyperlinks to XWizard Scripts 15
6.1 Long URLs 16
6.2 Short URLs, Script IDs and the XWizard Database 16

7 PDF Processors and the Conversion Method 'Plain PDF generator code' 17

8 More Complex Objects: Using Pre-Processors and Sub-Scripts 19
8.1 Sub-Scripts in LaTeX 20
8.2 Pre-Processors 23
8.3 Pre-implemented Examples With Complex Objects 24

9 Known Bugs, Shortcomings and 'Pitfalls' 28

10 Legal Note 28

1 What is XWizard?

XWizard is a free (web) tool for the automatic **visualization, manipulation and PDF generation** of many types of objects from theoretical computer science (such as Turing machines, push-down and finite automata, Chomsky grammars etc.). A broad range of algorithms can be applied to the objects, producing intuitive and customizable views. XWizard is well-suited for students' self-studies, and it is powerfull in aiding teachers at the creation of course material such as exercises (the X in XWizard stands for "eXercise" – and also for "anything"). This handbook explains the most important features of XWizard from a teacher's perspective. For more general information, read the document "XWizard: for Students" or look at the help pages on the XWizard website.

Readers interested in the basic workflow and the overall functioning of XWizard can skip the next section and move on to Sec. 3.

2 Access and Short History

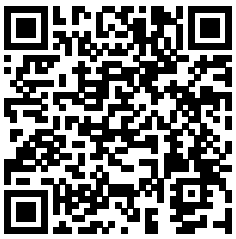
XWizard was created in 2013[1], and the original version was called **Very Fast PDF generator (VFP)**. Its purpose was to simplify the creation of course material, meaning that it was orginally solely used by teachers. Today, XWizard is the name of the **web version of VFP**, created in 2015 after students requested an easy-to-use version of VFP for themselves.

XWizard can simply be accessed via:

`www.xwizard.de`

or by clicking (or scanning) any of the script links in this document, such as:

Script ID-10700



Today, this web version suits well for most purposes. To get the general idea behind XWizard, feel free to play around on the website and apply algorithms to example objects by clicking the "conversion methods" (for example, the conversion method "Simulate one step" of the above script). Note that script IDs such as the one given above can as well be typed into the script field on the website.

The download version VFP, which is still the backbone of XWizard, can be retrieved from:

`https://sourceforge.net/projects/easyagentsimulation`

VFP (as opposed to XWizard) has to be installed on a personal computer, and it requires additional software to run[2]. Both versions have essentially the same range of functionality, however, VFP has unlimited computational power while XWizard will interrupt very long or memory-intensive calculations. Nevertheless, **from a teacher's perspective using the web version, which requires only a browser, will in most cases be the best choice**, at least for starters. As both versions are mostly innerchangeable, only the term XWizard will be used in the following, except where a difference between the two is explicitly addressed, such as this:

XWizard can be switched between English and German language; so far, VFP is only available in English.

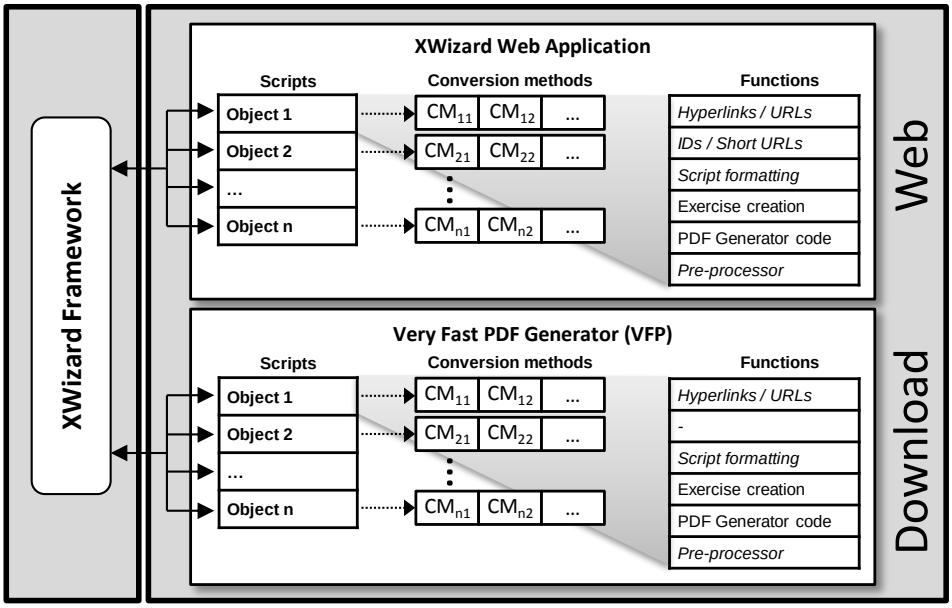
[1] By the teaching team of the course "Foundations of Computer Science II" at the Karlsruhe Institute of Technology, taught by Prof. Dr. Hartmut Schmeck, assisted by Lukas König, Friederike Pfeiffer-Bohnen, Micaela Wünsche and Marlon Braun.

[2] That is, minimally Java 8, LaTeX (preferably miktex), Graphviz, and SumatraPDF.

XWizard and all implementations related to it are *free software*[3]. They are allowed to be run for any purpose (except commercial), and the sourcecode can be studied, changed, and further distributed in accordance with this legal notice: `http://www.xwizard.de:8080/Wizz?impressum&lang=eng`. (Cf. Sec. 10 of this document.)

3 Basic Workflow: Script Processing

The below figure shows the overall structure of the XWizard components for both the web and the download version (XWizard and VFP; as mentioned above, they are basically equal). XWizard objects are given by **scripts** which can be manipulated by **conversion methods**. Therefore, the main functionality of XWizard is established by the conversion methods. However, conversion methods (and basically everything else) can be encoded into scripts, therefore, in the end, everything boils down to scripts.



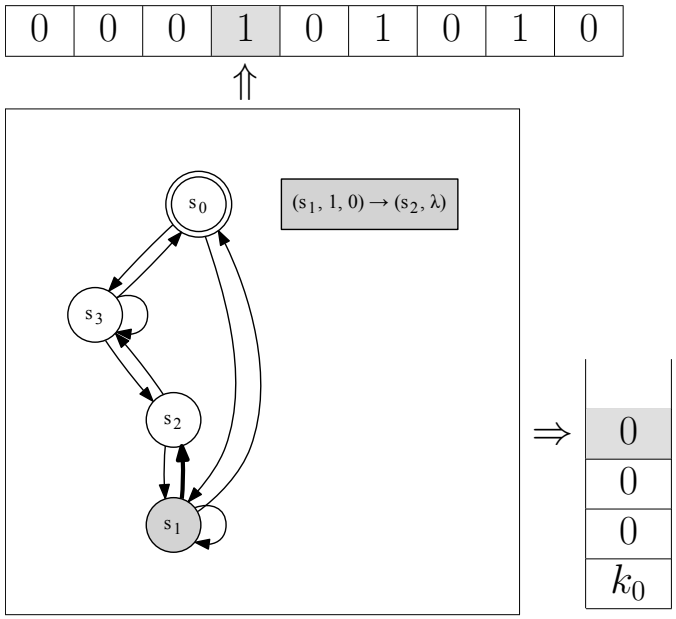
XWizard's basic workflow is simply to process a script, translate it into a PDF image and display this image. A script is usually built up of three parts (or else four, when encoding a script conversion at the bottom, see below), as illustrated in the following example of a PDA:

```
pda:                                                    ⇐ Script preamble
    (s0, 0, k) => (s1, 0k);
    (s0, 1, k) => (s3, 1k);
    (s1, 0, 0) => (s1, 00);
    (s1, 1, 0) => (s2, lambda);
    (s1, lambda, k) | (s3, lambda, k) => (s0, k);
    (s2, lambda, 0) => (s1, lambda);                    } Main script part
    (s2, lambda, k) => (s3, bk);
    (s3, 0, 1) => (s3, b);
    (s3, 0, b) => (s3, lambda);
    (s3, 1, 1) => (s3, 11);
    (s3, 1, b) => (s3, b1);
--declarations--
    e=#n#;
    s0=s0;
    F=s0;                                               } Variable declarations
    kSymb=k;
    inputs=000101010;
    simSteps=3;
--declarations-end--
```

[3] `https://en.wikipedia.org/wiki/Free_software`

All scripts include the following three parts: a **preamble** determines the type of object defined by the script; a **main part** defines the actual structure of the object; variables in a **declarations part** can be used to assign many types of additional properties (both the complete declarations part and some of the variables can be omitted, in which case the according variables are set to standard values).

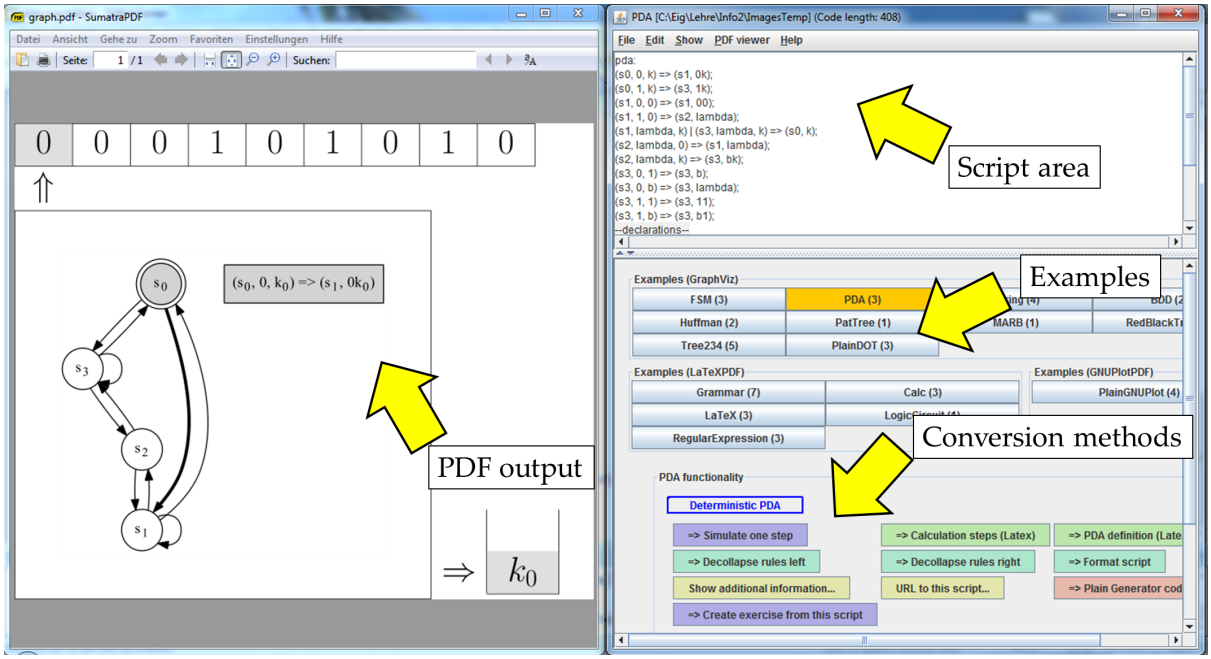
This example script will create the following image (using Graphviz and LaTeX in the background):



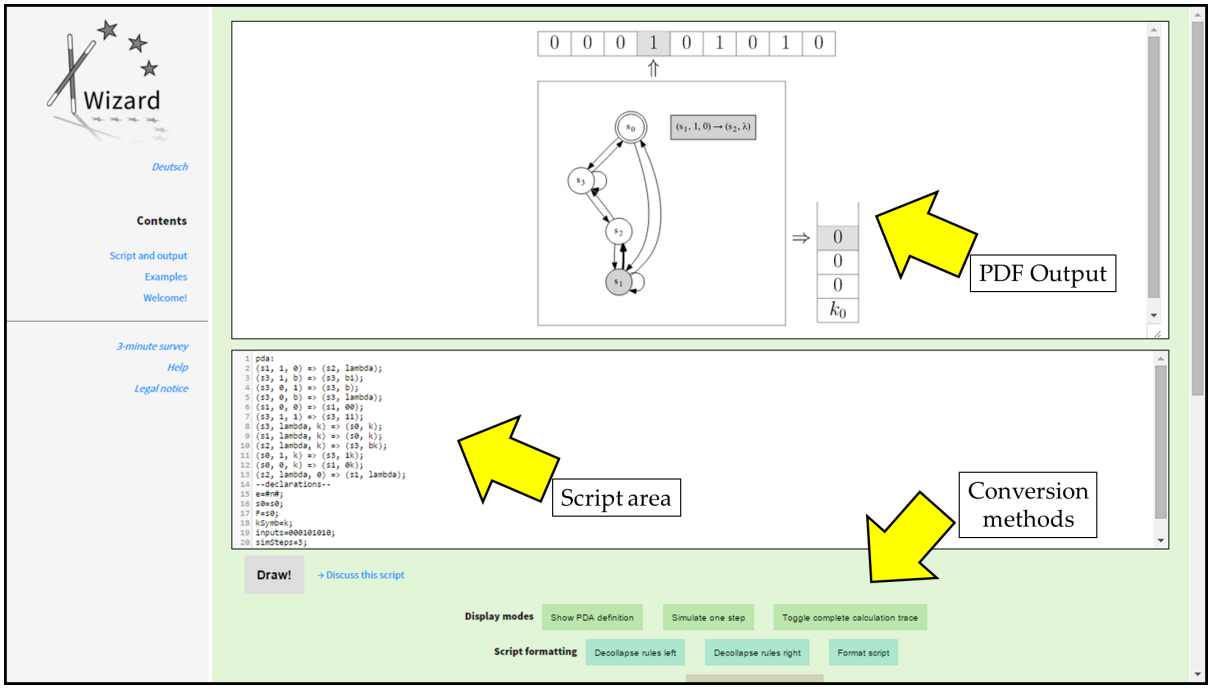
Script ID-13417

The script can be pasted into the script areas of both VFP and XWizard to produce the given output, see screenshots below (clicking or scanning the above QR code will open XWizard and automatically execute the script).

VFP:



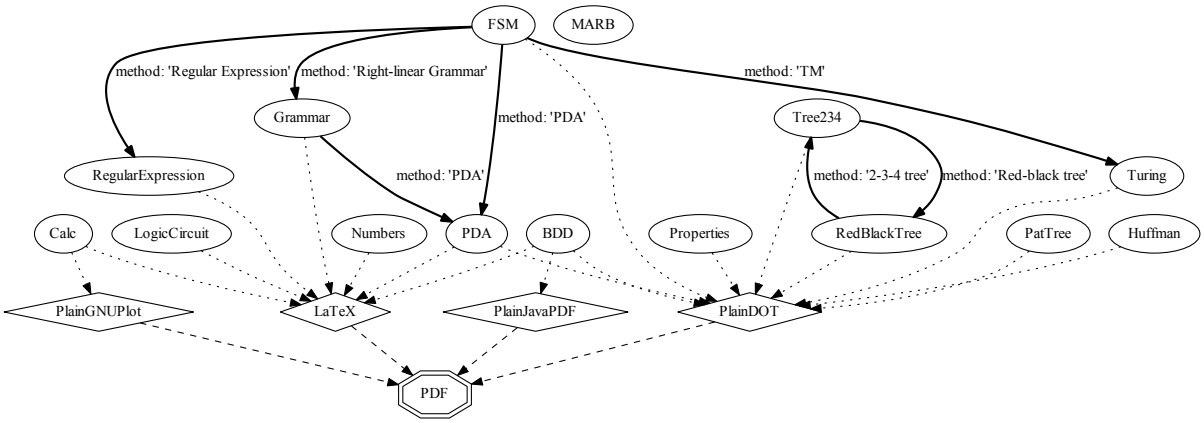
XWizard:



The calculation of the displayed PDA can be carried on by increasing the "simSteps" variable in the script or by repeatedly clicking the **conversion method** "Simulate one step" (cf. description of conversion methods below).

When using VFP, the PDF output is automatically updated each time the script area changes. When using XWizard, the image output is created after clicking the "Draw!" button on the web page. A PDF can be retrieved by clicking the "Download PDF" link; it appears when scrolling down below the PDF output.

An overview of the script types available at creation time of this document are depicted in the following figure (solid and dotted arrows denote script type transitions by conversion methods; dashed arrows denote the PDF creation process; diamond nodes represent plain PDF generator script types – see below for details on these terms).



A current version of the figure can be retrieved via this link (note that it is itself generated by a simple XWizard script):

Script ID-15812

For any of these object types, there are example scripts on the XWizard web page:

`http://www.xwizard.de:8080/Wizz?lang=eng&hide#Examples`

Note that the syntax of scripts will not be discussed here in detail; for this, see the XWizard help pages:

`http://www.xwizard.de:8080/Wizz?help&lang=eng&hide`

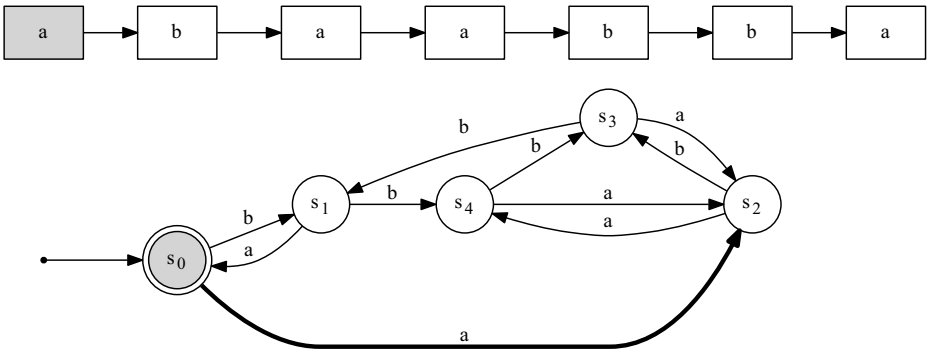
To sum up this short section, the basic workflow of XWizard is to take an input script and to create a PDF image from it. Nearly all GUI-based abbreviations and simplifications described below can be substituted by creating an according script; more precisely, all the GUI does is inducing the creation of appropriate scripts in the background. A major benefit of this structure is that every action can be archived by simply storing the according script, cf. Sec. 6.

4 Conversion Methods

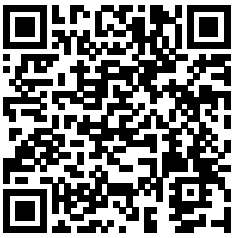
Besides creating and displaying objects, a main functionality of XWizard is to apply algorithms to scripts, e. g., to visualize the stepwise computation of a PDA or to minimize an FSM. Algorithms are applied to objects by using conversion methods, i.e., methods that transform one script into another. Conversion methods provide a simple user interface for applying algorithms to XWizard objects.

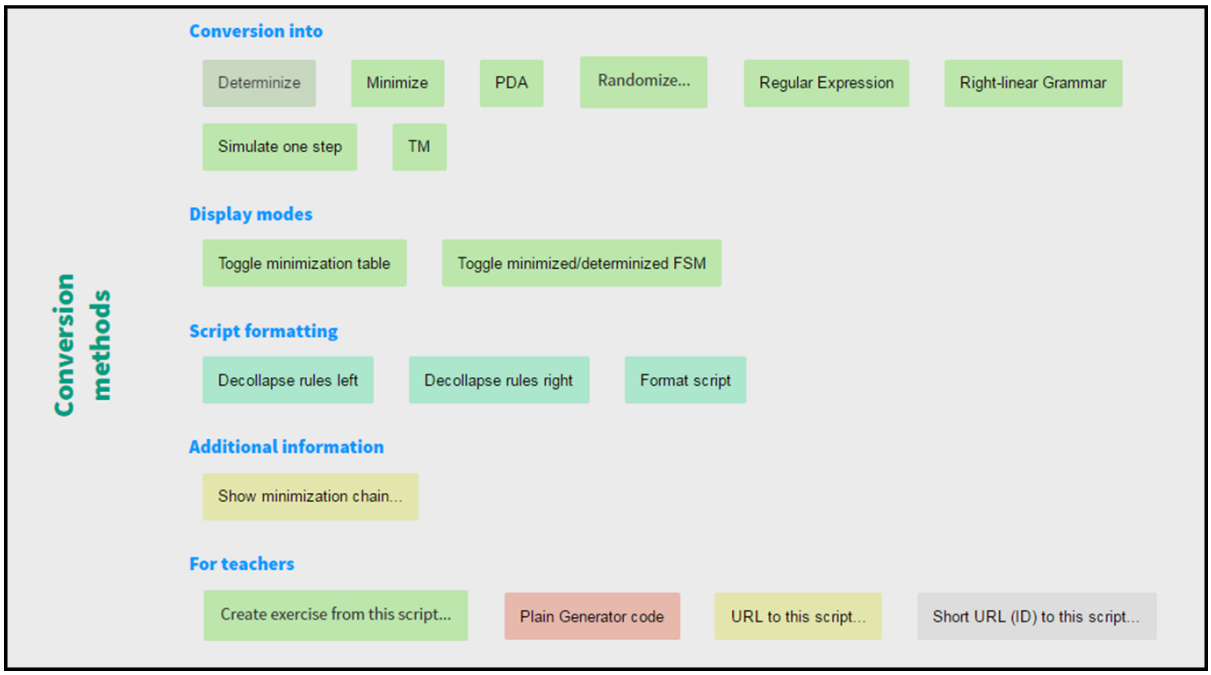
When applying a conversion method, the script belonging to the current object is replaced by the new script created by the conversion method, and the object defined by the new script is created and displayed. (Note, however, that there are also conversion methods that produce a plain text output rather than a new script.)

The easiest way to apply a conversion method to a script is to click the according button in the "Conversion methods" area of the graphical user interface. For example, a finite state machine (FSM), such as the following, comes with a set of conversion methods as shown below.



SCRIPT ID-10700





4.1 Conversion methods which create a new script

Conversion methods which convert a script into a new one are the usual – and by far most-frequent – ones. The conversion methods are displayed below the script area; the type of the current script defines which conversion methods are applicable, and only these are displayed. In the above example, the following conversion methods are shown for an FSM script, grouped along the categories printed in blue:

- **Conversion into** – Methods that will create a "new", i. e., semantically different object:
  - **Determinize** will use the well-known powerset algorithm to create a script that represents a deterministic FSM equivalent to the original one. (This method is not enabled in the above example, because the FSM is already deterministic.)

  - **Minimize** will create an equivalent FSM with a minimal number of states, using the Myhill/Nerode equivalence statement.
  - **PDA**, **Regular Expression**, **Right-linear Grammar** and **TM** will create an equivalent push-down automaton, regular expression, right-linear grammar or Turing machine script, respectively.
  - **Simulate one step** will let the FSM consume one more character of the given input, or ask the user to enter an input word if none is given.
  - **Randomize...** will create a new random FSM, by asking the user first to enter a number of states and a boolean value indicating if the new FSM should be deterministic. (Note that the resulting FSM will always be minimizable, to facilitate the creation of exercises.)

- **Display modes** – Methods that will change the view on the current object, but not its semantics:
  - **Toggle minimization table** will switch between views showing only the FSM, both the FSM and its minimization table, and only its minimization table.
  - **Toggle minimized/determinized FSM** will switch between views showing or hiding an additional view of a minimized and a determinized version of the current FSM.

    
    Showing different equivalent versions of the same FSM allows to simulate them simultaneously, which can be interesting both for students and at exercise generation.

- **Script formatting** – Methods that will change the script appearance, but not the output:
  - The two methods **Decollapse rules left** or **right** are available for all scripts based on rules such as `A => B`. Such rules can be collapsed on the left side (`A | B => C`) and on the right side (`A => B | C`) for better readability. The "Decollapse" methods will undo such collapsing on the left or right side, respectively.
  - The **Format script** method will collapse the rules where possible, and do some more formatting.

    
    Particularly, the methods **Format script** or **Add declarations to script** (which one depends on the current script type) will add the declarations part to a script including all available variables.

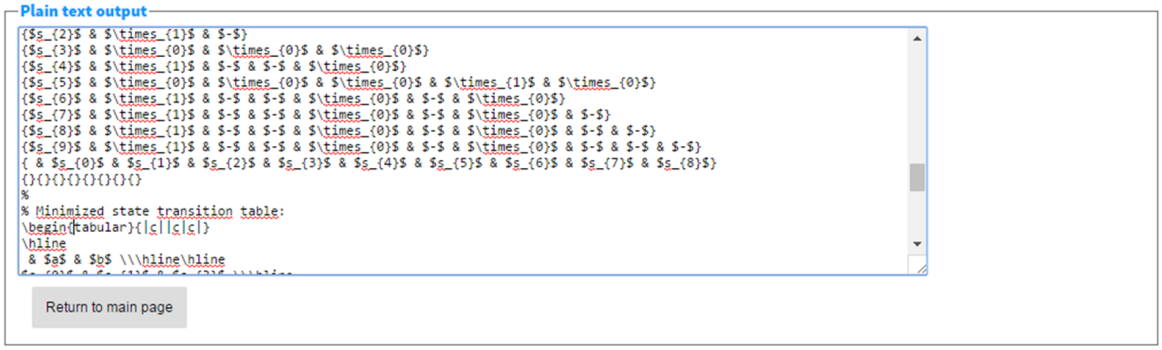
- **Additional information** – The single method **Show minimization chain** from this category is a plain text conversion method. It produces an output ready to be copied into a LaTeX document, showing information about making the current FSM deterministic first and minimizing it afterwards.

- **For teachers** – The methods from this category are available for all scripts. As they all relate to major topics of this handbook, they are explained in detail in Secs. 5, 6 and 7.

Three dots (...) at the end of a button name indicate methods that require some user interaction. Some methods need parameters to be executed, these methods will ask for those parameters before starting the conversion, cf. the "Randomize..." method explained above. The remaining methods with dots are those that create a plain-text output. They will just open a new window to show their output, asking the user to press a confirmation button.

4.2 Conversion methods which create a plain text output

Besides the above-described, regular conversion methods, there are those which create a plain-text output. For example, the method "Show minimization chain" of an FSM script produces a LaTeX code output:



In this case, the output is a LaTeX code that includes information about the minimization and determinization of the current FSM. Besides this, the main conversion methods which produce plain text output are:

- "URL to this script..." and
- "Short URL to this script..."

These methods create URLs to the current script to facilitate script exchange between users, cf. Secs. 6.1 and 6.2.

4.3 In-Script Application of Conversion Methods

The application of conversion methods via scripts is an advanced feature, added to this handbook for interested readers. It is not mandatory to understand the usual workflow of XWizard, so the rest of this section may well be skipped.

As mentioned above, scripts control XWizard completely; this particularly means that even the application of conversion methods can be initiated via a special script type. Such a **conversion script** starts with the regular three parts, which determine the script to be converted. As a fourth part, a **conversion command** is written as the last line of the script. A conversion command looks like this:

```
**>CM-NAME<**
```

where `CM-NAME` is the English name of the conversion method to be applied or, if the conversion method requires parameters:

```
**>CM-NAME[p1, p2, ...]<**
```

where `p1, p2, ...` are the method parameters (they can be put in quotes if they are supposed to include special characters such as white spaces or commas: `["p, a, r, 1", "p, a, r, 2", ...]`). For example, the "Simulate one step" conversion method can be applied to a PDA script by adding the red-colored last line to it:

```
pda:
    (s1, 1, 0) => (s2, lambda);
    (s3, 1, b) => (s3, b1);
    (s3, 0, 1) => (s3, b);
    (s3, 0, b) => (s3, lambda);
    (s1, 0, 0) => (s1, 00);
    (s3, 1, 1) => (s3, 11);
    (s3, lambda, k) => (s0, k);
    (s1, lambda, k) => (s0, k);
    (s2, lambda, k) => (s3, bk);
    (s0, 1, k) => (s3, 1k);
    (s0, 0, k) => (s1, 0k);
    (s2, lambda, 0) => (s1, lambda);
--declarations--
    e=#n#;
    s0=s0;
    F=s0;
    kSymb=k;
    inputs=000101010;
    simSteps=3;
    maxNondetCalcDepth=12
--declarations-end--
**>Simulate one step<**   /* This is a conversion command. */
```

When entering this script, the result will be exactly the same as after clicking the according button on the script without the conversion command.

5 The Exercise Mode and Encrypted Scripts

XWizard has a distinguished mode called **Exercise Mode** (which is currently only available in the web version). In this mode, the user (typically a student) is asked to solve a task displayed at the top of the page. To get to the solution, the user is allowed to utilize a predefined subset of the XWizard functions. This subset can be flexibly defined by the creator (a teacher) of an exercise, and it can involve both an adjustment of the available conversion methods and a limitation of the script utilization. More precisely, the Exercise Mode differs in the following regards from the regular mode:

(1) The XWizard website prompts a question, displayed over the script area, and requests the user to answer it (cf. screenshot below).

(2) Some of the conversion methods may be hidden to prohibit undesired shortcuts to the solution.

(3) The script may be partially or completely encrypted to prohibit users from cheating by changing it or reading the exercise definition (see below).

(4) When answered correctly, the user is offered a "badge". (This is so far just a secret code word, displayed to the user; in future, the implementation of personal "portfolios" is planned, allowing users collect badges, experience points or similar things.)

(5) When answered correctly, an additional explanation may be displayed to the user if provided by the creator.

5.1 Creating an exercise

The exercise mode is defined by a variable called "`e`" (or "`exercise`") in the declarations area of a script. Therefore, this variable has to be declared correctly in order to create an exercise. A conversion method **Create exercise from this script** (which is available for every script type in the "For teachers" category) facilitates this process. When executing the method, the user is asked for the following parameters (all except the last two are string values; optional parameters can be simply left empty):

- titleString: A title displayed above the detailed exercise description.

- explanationHTML: A detailed description of the exercise which may contain HTML.
- solutionString: A string which represents the solution to be entered by the user. This parameter is optional, if left empty, the user is not prompted to enter a solution.
- codeToEarn: The "badge", i. e., a string displayed to the user as a reward if the answer is correct.
- regexForAllowedMethodNames: An optional regular expression (as used in Java) to restrict which conversion methods are displayed. The regular expression is applied to the English method name and displays only matching methods. For example the regular expression

  ```
  .*inimiz.*
  ```

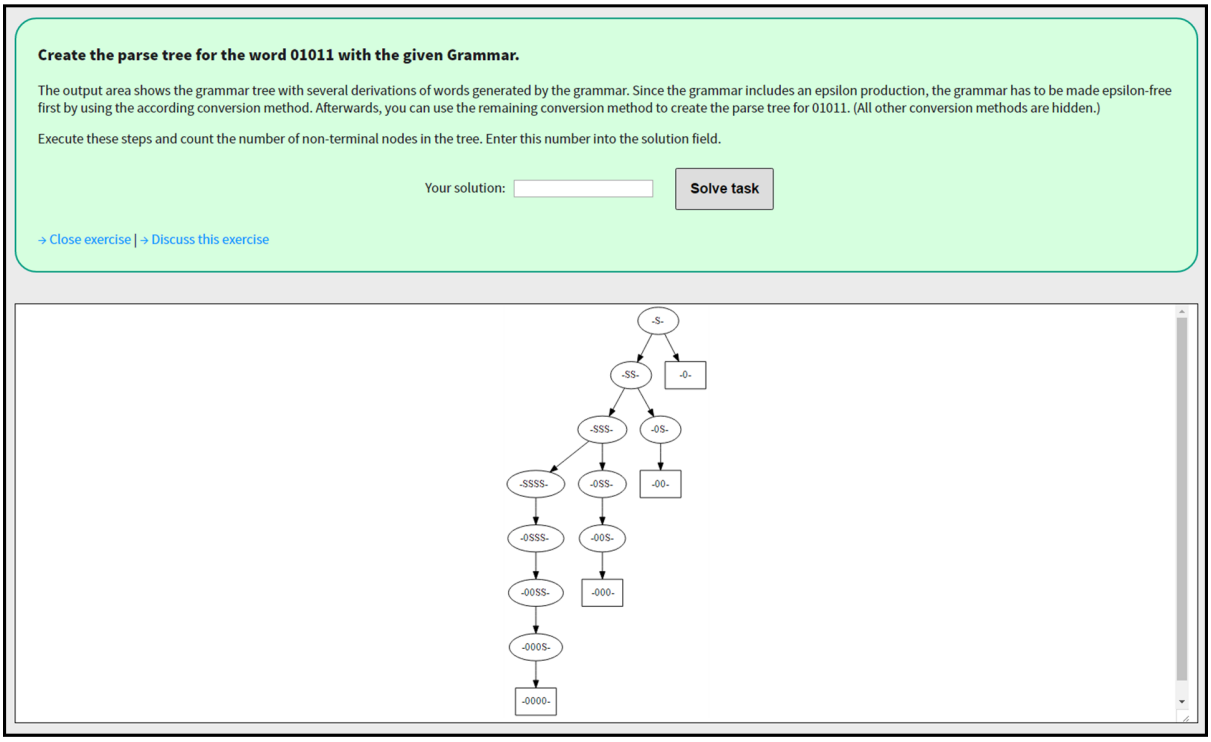
  will only display the two methods **Minimize** and **Show minimization chain...**
- regexForAllowedClassNames *(for expert use only, can usually be left empty)*: Another optional regular expression to restrict the display of conversion methods. It is applied to the class name of the base class which provides the method.
- regexForAllowedTargetClassNames *(for expert use only, can usually be left empty)*: A third regular expression to restrict the display of conversion methods. It is applied to the class name of the target class, i. e., the class of the converted script.
- solExp: Optional explanation to be displayed with the solution after a correct answer has been given (may contain HTML).
- exEncrypt *(boolean)*: Set this to true if the code of the exercise (not the complete script code) is to be encrypted (see below).
- encrypt *(boolean)*: Set this to true if the complete script code should be encrypted (see below).

After the input of these parameters, an exercise definition is added to the current script, and this causes XWizard to enter the exercise mode. (Note that "e=n" or "e=null" is the code for "regular mode".)

The following script shows an example of how an exercise is encoded in the declarations part of a script. The output is displayed in the screenshot right below it. Click the script link to see the example in a browser.

```
grammar:
    A => A, A | 0 | epsilon;
    E => A, 1, A;
    S => E, E, E | S, S | 0;
--declarations--
    e=#tit=~Create the parse tree for the word 01011 with the given Grammar.~,
       exp=~<P>The output area shows the grammar tree with several derivations
            of words generated by the grammar. Since the grammar includes an epsilon
            production, the grammar has to be made epsilon-free first by using
            the according conversion method. Afterwards, you can use the remaining
            conversion method to create the parse tree for 01011. (All other
            conversion methods are hidden.)</P><P>Execute these steps and count the
            number of non-terminal nodes in the tree. Enter this number into the
            solution field.</P>~,
       sol=~6~,
       cod=~parser-guru~,
       met=~.*Epsilon.*|.*Parse.*~,
       cur=~.*~,
       tar=~.*~,
       crypt=~false~,
       excrypt=~false~,#;
    N=S,E,A;
    T=0,1;
    S=S;
    ...
--declarations-end--
```

In the declarations part, the symbols # and ~ indicate the beginnings and endings of strings which may include special symbols such as "= , ;" etc. (however, the symbols # or ~ themselves, respectively, and some reserved words cannot be used unconditionally). To completely secure the string, it can be put within the following bracket combination: `[~(~{ some text }~)~]`. This allows to include all character combinations including subordinate instances of the bracket combination itself, until the secure part is finished by the matching closing bracket combination (cf. further explanations in Sec. 8).



Script ID-11020

As the example script shows, the exercise definition is, for now, given as plain, human readable text in the script. Particularly, this means that students are able to see the solution as well as change parts of the exercise definition. For example, they can change it to show conversion methods which are supposed to be hidden. As this may be undesired, **two levels of encription** can be applied to the script. The first level is activated by setting the exercise parameter

```
excrypt=true
```

which leads to an encryption of the exercise definition. The second level encrypts the whole script, and it can be activated by setting the parameter

```
crypt=true
```

The two example scripts below show the respective results; the output remains the same as shown above.

Example script with "excrypt=true":

```
grammar:
    A => A, A | 0 | epsilon;
    E => A, 1, A;
    S => E, E, E | S, S | 0;
--declarations--
    e=#scrypt:401s3X3o133k2L2R2g202p2z0x3w1Q1G052n0b040u2K0K0l1y2b1k161t0d0Q1J0I
              1K0A0B2m1c0T1B3G3726250T1q2y1h0c3d3f2B1T1D350z3j2c3L3a3u2M0b102s2l
              061a0J3X0e1K1z3Y430X1v142Z0A0N1T0e1z042831301F1r0O2J2x102O450V1x35
              0Y1G2M1E0Q1C283s3f3718400m3X3F170K1m0d3d1O043O2H1O1a1z3G3p082E2k07
              2G2H3S3a1k3D382j071d1p45271h012w460z1m180k332e3u263y2E3i0h3C1z1G0P
              0H0A1v2X1K3K1n3k042714303346143R063Y3S3C3f3r2P3h0w3F3x0P1g0F0t3w0V
              3e3M3t0k2k3I3i3F1S2a2y0704440S3M0T033h1e1y3q351d1I0d3T0S01463z0I1Y
              1M1h163x1i2a0K2y0D2d0y1g1X2H011m3O3D2g1w2D0p1J1Z471Z2b3j461V0a2O0L
              0k3g2H3f2P1l320Y3Q3J3g3j081Q443m300d0o1S3E0R342b1k1W2J1o1G262V3i0R
              3D3C3d301n301a1F3o2A2V3z252m0D3y3p1n0M3S1Q0C0a0n3k0r450V2L0t0b102P
              1L2y102Z3I1g0B3p15452R0r1v2u0P1r0Y0p1f101g1f3c1j0p1J0I3W1b46322w3B
              2q172j3Y2m0I471I0z3k0s0R442b233Z1U0C#;
    N=S,E,A;
    T=0,1;
    S=S;
    displayMode=2;
    maxdepth=8;
    cutNonTerminalBranches=true;
    cutTerminalDoubleBranches=true;
    maxLengthWords=4;
    multiLetterSymbolsHaveIndex=true;
    parseTreeNum=0
--declarations-end--
```

Example script with "crypt=true":

```
scrypt:401s3X3p1t0a1t2Q2k452j1C2z2a1U0B2h0O0r0k3o3g0X3w3e3x1l2q0S2B1y360v2p3N3E
     2f3G1b1G0i2f2M221U0e1C280W1v2q2w1B0p2T3V1E1e3D1k301j1V14450s1h1o122z3C1f3C
     2h3l0M0A2z2i3A3L3N341f3J1n2p3v2l3e3D0t140a1D071I3M2a3X3M2B3G342L2e090d2q3k
     283H0d2P0a1X2g1X3z3m010f3B2w0a002k3004372Y3l3B2i2l2i1o1D1m1b302d1d1n3U0R02
     0j2e3z1g0R1I082a103j261d0h1z411y1B211U3D3d0o0d3D0825280h021N1K1b3l2x3p1L3a
     0t41322y1j2d2h2m0t303k0y122y1Q2Y2j2N2u26273g1q3B221K2P0x3s0C133J0k0k1i3n0y
     2b3D3Q1g121y0I0c3j2C3H1h3M31373U2M1K2l422B1P3s3A2w1Q141c1X0g1E3b0L3i0r1t3a
     1l0g0s2o2V1o0g3y3l041i3F1G0I2M3D452c1G3k021C2S0U1q2c2h1f0f461k1E0M2r1c200P
     2R1f3R221J402W0W043t3N0J132m1M3A2m1W1C2N3W3g3g2U1D1J1Z2X1t2b3U10303F2n2H3g
     0y1R012W2D352E3X3d2n1Q2A2n1Z331j0h17062W2G1H0Z2S3G110N1b303H3U3j0i3n383H2c
     0B3u1X3x0j2S2E302p400H3F3U0l1e382N1d2U443G0k1C1B1Z0h2V3s1Q2R1N1B0a2W0W2Y2S
     2m1e193U3n3j2W0n0b0D3U0N3j400V0w1I1G2m2W0O2A0K2z0d0X2g3q2z3y163W3o1p1O2A0Q
     0D1W0e0e40153N3s0N3W1G31103G0H3W1O0Y0C0T3q461i1A1Q0j3s1z3R2J1M173v1A2k3022
     01260O3H3F1f1M0W1N3J2H3i392y1P2U3H383R1i3t2v1V2u232O2A0K3d2U3P1G460B151C1N
     3o2K1I0w2y2R0g3h1T0B1x3x1L110b2D222Q2m1h0T0K1K140e1u1u160u2P3B201X1i1a0y1W
     2h3i3q3r08
```

Note that encryption is only meant to make cheating difficult – not to make it impossible. It is achieved by a combination of zipping the text and converting the result to alpha-numeric values. As the XWizard sourcecode is free, ambitious students can download it and use it to decrypt a script. Nevertheless, be aware that after applying encryption in XWizard, **you will not be able to undo this easily**, therefore it makes sense to backup a non-encrypted version of each script.

6 Hyperlinks to XWizard Scripts

XWizard can generate URLs which point to specific scripts, to facilitate the exchange of scripts. This

technique has been used to generate the script links in this document. When following such a link, the XWizard website will open and automatically load the according script. There are two types of URLs ("long" or "short") that can be used for this purpose; they are explained in the following.

Short URLs are preferable in most situations to long URLs in terms of convenience and security, as long URLs can get too large to be accepted by a browser. Nevertheless, long URLs, as opposed to short ones, carry the whole information about a script which makes them independent of the XWizard database (see below).

6.1 Long URLs

Every script type comes with a conversion method **URL to this script...** which generates what is here called a "long URL" to this script. For example, the following script leads to the URL shown below:

```
latex:
    \mbox{XWizard long URL}
--declarations--
    formulaMode=true;
    e=#n#;
--declarations-end--
```

```
http://www.xwizard.de:8080/Wizz?template=latex%3A%0D%0A%5Cmbox%7BXWizard+long+
URL%7D%0D%0A--declarations--%0D%0AformulaMode%3Dtrue%3B%0D%0Ae%3D%23n%23%3B%0D%
0A--declarations-end--
```

The URL points to the XWizard website and transmits a parameter "template" which contains the whole script as a URL-encoded string. When such a link is opened, XWizard will decode the script, copy it into the script area and show the according output image.

In principle, every script can be converted into such a long URL, however, if the URL gets too large, a browser may reject parts of it which leads to corrupted scripts. Furthermore, malware blockers may produce alerts due to the unusual form of these URLs. For these reasons, **short URLs** have been introduced as described in the next section.

6.2 Short URLs, Script IDs and the XWizard Database

A script of similar size as the one shown in the previous section, such as this:

```
latex:
    \mbox{XWizard short URL}
--declarations--
    e=#n#;
    formulaMode=true
--declarations-end--
```

can be encoded into a much simpler URL which is generated when the conversion method **Short URL to this script...** is executed:

```
http://www.xwizard.de:8080/Wizz?template=ID-11567
```

Instead of encoding a complete script within the URL, short URLs make use of the XWizard database. XWizard (not VFP!) stores every processed script along with different types of corresponding information into a mysql database. Each script is assigned an ID in the database which can be used to retrieve the script later. If this ID is passed to XWizard using the "template" parameter, it will be looked up in the database and checked if it is "web-free", meaning if it can be displayed to the public. Of course, this is not allowed for all scripts; rather, when executing the **Short URL to this script...** method, a flag is set which makes the script "web-free". All other scripts are protected and will not be shown when the according ID is loaded. Therefore, only the creator of a script can decide to make it publically available via ID. Note that all the script links in this document are based on short URLs.

Although convenient, short URLs set up a pitfall when creating "critical" scripts, e. g., scripts for exam questions, as scripts with public IDs can in principle be viewed by everyone. Despite the unlikeliness of students "guessing" an ID which belongs to a script relevant to their exam, critical scripts should not be made public by creating short URLs for them.

A public script ID can also be typed into the script area of XWizard (not VFP), instead of using it within a short URL.

Adding "&lang=ger" or "&lang=eng" to any of the two URL types can specify the XWizard language to German or English. Adding "#Output", "#Codebox", "#ConversionMethods", "#Examples" etc. to the very end of a URL will let XWizard jump immediately to the respective part of the website.

7 PDF Processors and the Conversion Method 'Plain PDF generator code'

Every XWizard script is associated to a PDF processor which compiles the script into a PDF. The translation process is divided into three steps as follows:

(1) **Script** (user-defined)	⇒	(2) **PDF processor code** (obtainable by **Plain PDF generator code** method)	⇒	(3) **PDF**

There, the PDF processor code is plain source code in the language of the PDF processor, i. e., it can be copied and compiled outside of XWizard as well (which is basically what XWizard does, by storing the code in a file and running, for example, pdflatex.exe or dot.exe from Java). The most important PDF processors used in XWizard are LaTeX and Graphviz; overall the following are implemented so far:

- LaTeX,
- Graphviz,
- GNUPlot (only available in VFP; GNUPlot has to be installed),
- JavaPDF (implemented, but not yet used in productive mode).

XWizard, in contrast to VFP, performs a fourth step: it converts the output PDF into SVG format which is directly embedded into the HTML code of the XWizard website.

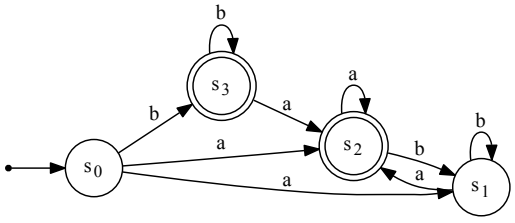
In contrast to students, teachers may frequently be interested in manipulating the PDF processor code – for example, in order to change some specifics in the depiction of an object to point out some peculiarity relevant to the course.

For example, it might be desired to highlight the **nondeterministic transitions** in the FSM depicted below on page 18. However, the FSM script type only allows to define an FSM object in terms of its mathematical properties, while the actual depiction is left to an internal algorithm. To get access to the actual depiction code, i. e., the raw PDF processor code, XWizard offers for each PDF processor a specific script type which allows to use code directly as accepted by the PDF processor. When using this script type, plain LaTeX or Graphviz (or GNUplot) source code can be inserted as the main script part, to be sent directly to the PDF processor.

Note that the plain PDF processor script types **can be** used to just pass code on to the PDF processor, but they can do more than that. First, preprocessors may be added to the code, see Sec. 8. Furthermore, using the declarations, some changes to the interpretation of the code can be achieved, cf. "formulaMode=true" when using LaTeX PDF.

To make use of these plain script types, every XWizard script has a conversion method **Plain generator code**. When executing it, the script will be converted into the according plain PDF processor code, embedded in the according script type. Now, this script can be changed as desired, according to the rules of the language of that PDF processor.

This procedure is illustrated using the following FSM script. As suggested earlier, it might be desired to highlight the non-deterministic transitions (the two outgoing transitions from state s_0 labelled a) in the PDF output in order to point out some peculiarity in a course.



Script ID-11832

The regular script producing the above automaton looks like this:

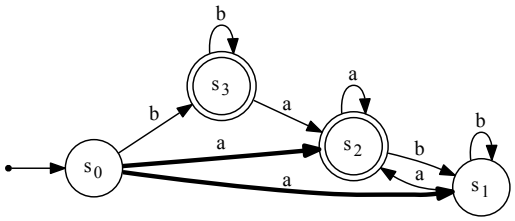
```
fsm:
    (s0, a) | (s1, a) | (s2, a) | (s3, a) => s2;
    (s0, b) | (s3, b) => s3;
    (s1, b) | (s2, b) => s1;
    (s0, a) => s1;
--declarations--
    e=#n#;
    simulateToStep=-1;
    input=null;
    s0=s0;
    F=s3,s2
--declarations-end--
```

Obviously, there is no way of adding information for highlighting transitions to this script. However, by executing the **Plain generator code** method, the script is being converted into raw Graphviz code which looks as follows (first without the text highlighted in red):

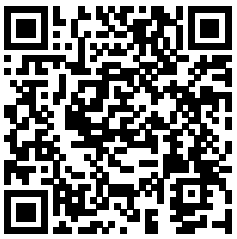
```
dot:
    digraph G {
        rankdir=LR;
        node [shape = point ]; qi
        node [shape = circle];
        s1[label=<s<SUB>1</SUB>>];
        qi -> s0;
        s0[label=<s<SUB>0</SUB>>];
        node [shape = doublecircle];
        s2[label=<s<SUB>2</SUB>>];
        s3[label=<s<SUB>3</SUB>>];
        s3 -> s3 [label="b"];
        s3 -> s2 [label="a"];
        s0 -> s3 [label="b"];
        s0 -> s1 [label="a",penwidth=3];
        s0 -> s2 [label="a",penwidth=3];
        s1 -> s1 [label="b"];
        s1 -> s2 [label="a"];
        s2 -> s1 [label="b"];
        s2 -> s2 [label="a"];
    }
```

Converting a script into the plain PDF processor code will never change the PDF output. (However, the above code has been cleared out a little for better readability which slightly changes the output.)

Based on this script, the highlighting of non-deterministic edges can be achieved, for example, by adding the red-colored text to the plain Graphviz code. This causes Graphviz to draw the desired edges thicker than the others. The result is shown here:



Script ID-11836



Obviously, all kinds of changes can be made to this script as long as the Graphviz syntax is obeyed. The same is possible with scripts based on LaTeX. An according example is shown in the next chapter.

8 More Complex Objects: Using Pre-Processors and Sub-Scripts

While both Graphviz and LaTeX are powerful programs on their own, it sometimes makes sense to combine the two to allow the creation of more advanced figures. The following two examples illustrate typical use cases:

- The visualization of the stepwise calculation of push-down automata, such as the one on page 5, is a combination of a Graphviz graph and two LaTeX tables. It would be very difficult to achieve the same by using only one of the two programs.
- While it is certainly a common application of XWizard to create PDF images and import them into some external document, it can sometimes be more convenient to directly create small LaTeX documents with XWizard which contain both text and graph parts.

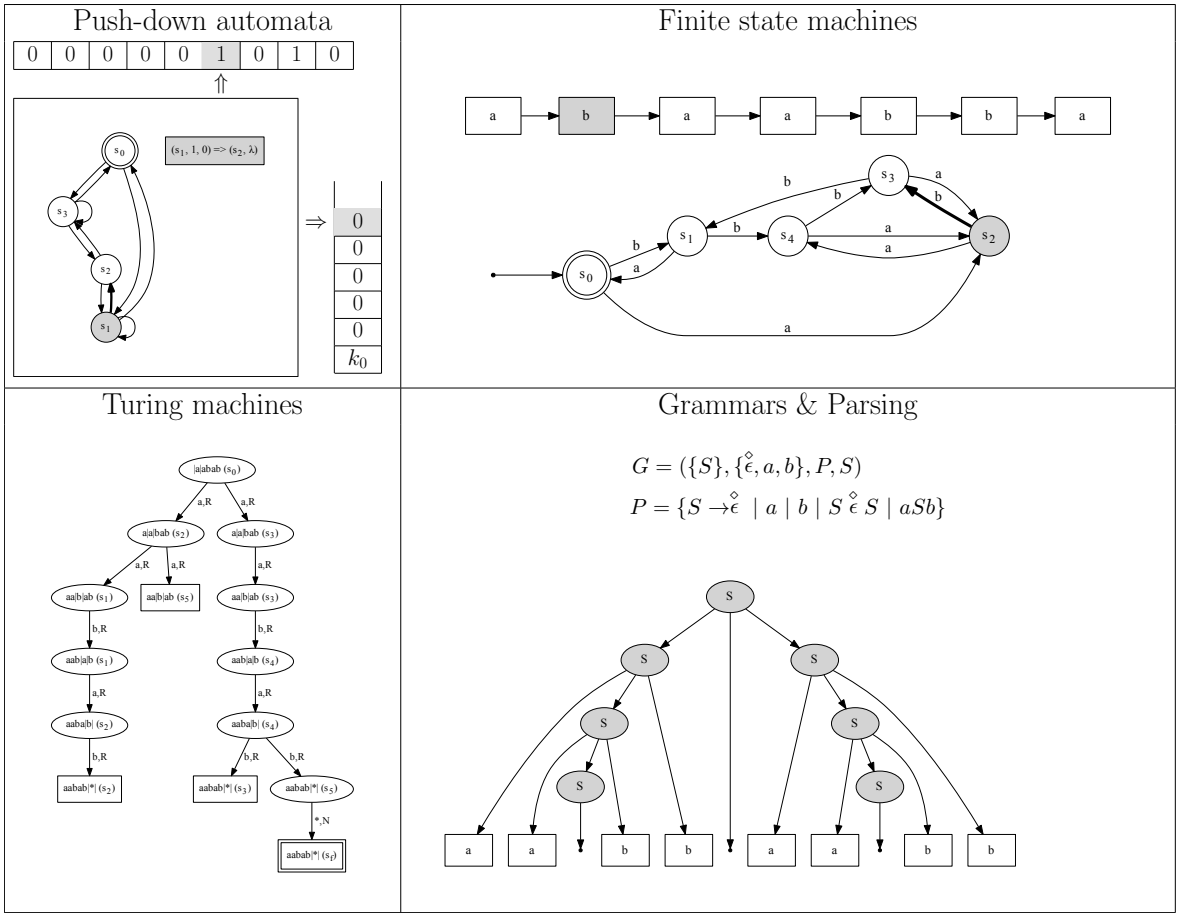
To facilitate both of these requirements, XWizard implements so-called "pre-processors". A pre-processor is a script X which is embedded in another script Y. During the translation of Y, X is translated first, creating the according PDF image P_X. As the actual father script Y is translated subsequently, Y can import P_X, thus combining its own contents with X's output.

For LaTeX, this general mechanism is encapsulated in an easy-to-use structure: the **Sub-Script**. Sub-Scripts are a special case of pre-processors, but they are easy to understand, and therefore described first in the next section. The more general pre-processor mechanism is described subsequently in Sec. 8.2.

8.1 Sub-Scripts in LaTeX

The image below is a complex LaTeX object which contains four XWizard sub-objects (a PDA, an FSM, a Turing machine and a grammar object).

Some of XWizard's basic object types:



Script ID-16124

It can be generated by the following XWizard script (the code of the sub-objects, the "sub-scripts", is highlighted in red):

```
latex:
    \documentclass[tightpage,preview]{standalone}
    \usepackage{varwidth}\usepackage{amsmath}\usepackage[table]{xcolor}\usepackage{graphicx}\usepackage[space]{grffile}
    \begin{document}
    \huge~\par
    Some of XWizard's basic object types:
    \bigbreak
    \begin{tabular}{|c|c|}
    \hline
    Push-down automata & Finite state machines \\
    @{0.75|
        pda:
            (s1, 1, 0) => (s2, lambda);
            (s3, 1, b) => (s3, b1);
            (s3, 0, 1) => (s3, b);
            (s3, 0, b) => (s3, lambda);
            (s1, 0, 0) => (s1, 00);
            (s3, 1, 1) => (s3, 11);
            (s3, lambda, k) => (s0, k);
            (s1, lambda, k) => (s0, k);
            (s2, lambda, k) => (s3, bk);
            (s0, 1, k) => (s3, 1k);
            (s0, 0, k) => (s1, 0k);
            (s2, lambda, 0) => (s1, lambda);
        --declarations--
            e=#n#;
            s0=s0;
            F=s0;
            kSymb=k;
            inputs=000001010;
            simSteps=5
        --declarations-end--
    }@ &
    @{0.7|
        fsm:
            (s0, a) | (s3, a) | (s4, a) => s2;
            (s0, b) | (s3, b) => s1;
            (s1, a) => s0;
            (s1, b) | (s2, a) => s4;
            (s2, b) | (s4, b) => s3;
        --declarations--
            e=#n#;
            simulateToStep=1;
            input=abaabba;
            s0=s0;
            F=s0
        --declarations-end--
    }@ \\
    \hline
    Turing machines & Grammars \& Parsing \\
    @{0.5|
        turing:
            (s0, a) => (s2, a, R) | (s3, a, R);
            (s0, b) => (s1, b, R) | (s4, b, R);
            (s1, a) => (s2, a, R);
            (s1, b) => (s1, b, R);
            (s2, a) => (s1, a, R) | (s5, a, R);
            (s2, b) => (s2, b, R);
            (s3, a) => (s3, a, R);
            (s3, b) => (s4, b, R);
            (s4, a) => (s4, a, R);
            (s4, b) => (s3, b, R) | (s5, b, R);
            (s5, *) => (sf, *, N);
        --declarations--
            s0=s0;
            F=sf;
            blank=*;
            inputs=aabab;
            runStepsScript=120;
            shortTrace=false
        --declarations-end--
    }@ &
    @{1.5|
        grammar parse(a, a, <>, b, b, <>, a, a, <>, b, b)--48:
            S => a, S, b | <> | S, <>, S | a | b;
        --declarations--
            N=S,A;
            T=a,b,c;
            S=S;
        --declarations-end--
    }@ \\
    \hline
    \end{tabular}
    \end{document}
```

The script is based on the LaTeX script type (cf. Sec. 7) which, beyond plain LaTeX code, allows for sub-scripts to be embedded. A sub-script is an arbitrary XWizard script which is put between the

following bracket combination:

```
@{ *some script* }@
```

During the translation, sub-scripts are treated as pre-processors which are translated first and stored in PDF files, say `file*X*.pdf` for sub-script `X`. Subsequently, each sub-script `X` in the LaTeX code is replaced by the following code:

```
\includegraphics{file*X*.pdf}
```

As a consequence, the subsequent LaTeX run will include the pre-compiled PDFs at the positions of the original corresponding sub-scripts, thus, inserting the sub-script's output into the overall script's output.

Obviously, sub-scripts may only be placed at positions where "includegraphics" is allowed in LaTeX.

If the included graphic's size is unsuitable, the sub-script code may be extended by a scaling parameter, like this (scaling to 0.5 as an example):

```
@{0.5| *some script* }@
```

This will lead to the LaTeX code

```
\includegraphics[scale=0.5]{file*X*.pdf}
```

which, in turn, will cause LaTeX to scale the image to half its original size. A negative number -0.5 will lead to

```
\includegraphics[width=0.5\linewidth]{file*X*.pdf}
```

thus adjusting the image width relative to the current line's width in the document (there is no particular reason for this functionality being encoded by negative numbers; it was just an easy way to achieve this).

Sub-scripts embody a powerful mechanism which allows for complex documents to be created quite easily. Note that a sub-script is allowed to be a plain LaTeX script itself, which, recursively, allows it to contain its own sub-sub-scripts, and so on. For example, in the script at the beginning of this section, the pda script is a sub-script which translates to a LaTeX script with an own sub-sub-script.

Note that, in the context of sub-scripts, the conversion method "Plain generator code" works in a slightly different way than described above. Executing it once will show the plain LaTeX script including sub-script notations; executing it twice, will show the actual plain LaTeX code where the sub-scripts have been replaced by the "inculdegraphics" commands.

8.2 Pre-Processors

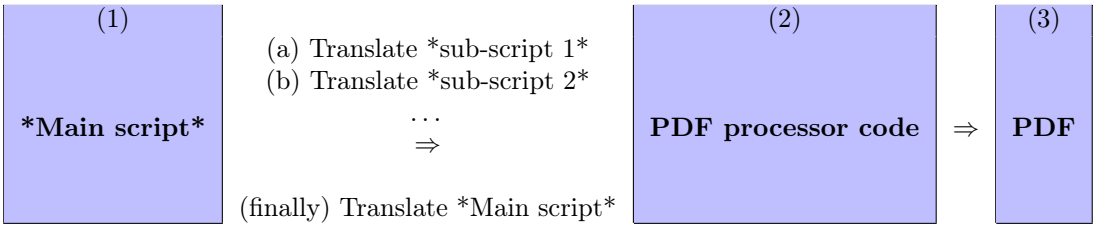
Sub-scripts are implemented on the foundation of the more general mechanism of pre-processors which can be used with every script type (not just plain LaTeX). However, at the time of creation of this document, the only application of pre-processors is the sub-script implementation described above. The rest of this section is written for interested readers, and to indicate possible future applications.

Every XWizard script can make use of pre-processors in the following way:

```
*Main script*
    ...
    * Import filename1.pdf *
    ...
    * Import filename2.pdf *
    ...
--Declarations--
    preprocessor1 = [~(~{filename1.pdf|*sub-script 1*}~)~];
    preprocessor2 = [~(~{filename2.pdf|*sub-script 2*}~)~];
    ...
--Declarations-end--
```

There, the declaration part can define an arbitrary number of pre-processors (the according variable names have to start with the word `preprocessor` and can continue arbitrarily). The pre-processor code begins with a file name to store the PDF output in. Behind that, separated by a `|` symbol, comes the actual script code which can be an arbitrary XWizard script. Note that this code can, recursively, contain further pre-processors in its own declarations part. The declaration of a pre-processor should always be enclosed in the **securing bracket combination** (cf. Sec. 5) to ensure its correct interpretation: `[~(~{ *pre-processor code* }~)~]`

The translation process of a script containing pre-processors is depicted below:



When translating a script which includes pre-processors, first the pre-processors will be translated (which, in turn, means that their sub-pre-processors, if any, will be translated before their own main part – i. e., "depth-first"). After all the pre-processors on the different levels have been translated, the main script is translated which now can assume that the subsequent PDFs exist. Therefore, the main script part can contain code to include the PDF files given in the pre-processors' codes. The result is a PDF file which may contain sub-images generated by arbitrary XWizard scripts.

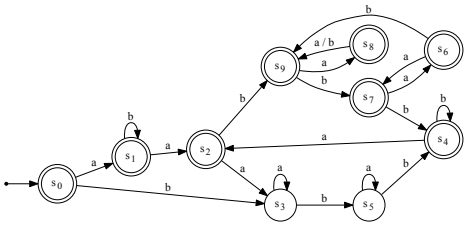
As mentioned before, the only application of pre-processors currently available is the implementation of sub-scripts in LaTeX. However, many interesting things can be imagined to be implemented with this powerful mechanism in future.

8.3 Pre-implemented Examples With Complex Objects

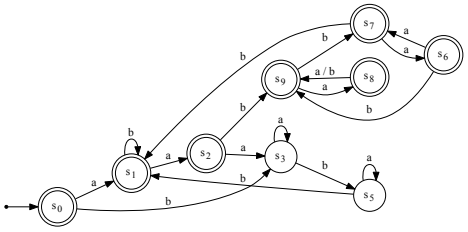
The sub-script mechanism described above is the foundation of several pre-implemented script types. These can be inspected to learn how to use sub-scripts.

For example, FSM scripts can be used to **simultaneously** show (1) an FSM, (2) a minimized version of it and (3) the according minimization table in a single PDF output. (See figure below; this view is very convenient when creating exam questions, as the difficulty of the question can be estimated more quickly if the additional information is automatically shown while the original FSM's definition is entered.)

FSM:



Minimized:



Minimization table:

s_1	$\times_1$							
s_2	$\times_1$	$\times_1$						
s_3	$\times_0$	$\times_0$	$\times_0$					
s_4	$\times_1$	–	$\times_1$	$\times_0$				
s_5	$\times_0$	$\times_0$	$\times_0$	$\times_1$	$\times_0$			
s_6	$\times_1$	$\times_2$	$\times_1$	$\times_0$	$\times_2$	$\times_0$		
s_7	$\times_1$	$\times_2$	$\times_1$	$\times_0$	$\times_2$	$\times_0$	$\times_3$	
s_8	$\times_1$	$\times_2$	$\times_1$	$\times_0$	$\times_2$	$\times_0$	$\times_4$	$\times_3$
s_9	$\times_1$	$\times_2$	$\times_1$	$\times_0$	$\times_2$	$\times_0$	$\times_4$	$\times_3$	$\times_4$
	s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8

SCRIPT ID-13363

This image is created by the FSM script below. Its main part defines the original FSM only, both the minimized FSM and the minimization table are created from it on the fly. The variable declarations "displayMode=1" and "showMinimizedFSM=true" state that these items should be displayed.

```
fsm:
    (s0, a) | (s1, b) => s1;
    (s0, b) | (s2, a) | (s3, a) => s3;
    (s1, a) | (s4, a) => s2;
    (s2, b) | (s6, b) | (s8, a) | (s8, b) => s9;
    (s3, b) | (s5, a) => s5;
    (s4, b) | (s5, b) | (s7, b) => s4;
    (s6, a) | (s9, b) => s7;
    (s7, a) => s6;
    (s9, a) => s8;
--declarations--
    e=#n#;
    simulateToStep=-1;          /* -1 means don't simulate FSM. */
    input=null;
    s0=s0;
    F=s4,s6,s7,s8,s9,s0,s1,s2;
    displayMode=1;              /* Show minimization table. */
    showMinimizedFSM=true;      /* Show minimized FSM. */
    showDeterministicFSM=false;
--declarations-end--
```

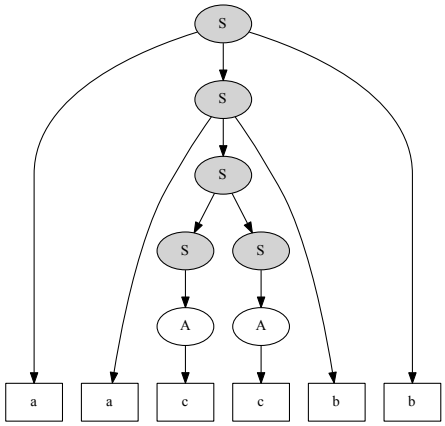
By executing the conversion method "Plain generator code", the according plain LaTeX script can be displayed (this script, as well as the other plain LaTeX scripts in this section, is not shown here due to its large size). The main script part contains LaTeX code which includes the headings and the minimization table (note that the long text in the preamble defines a macro for triangular tables). Furthermore, two sub-scripts of the plain Graphviz type are placed within the LaTeX code; they define the graphs for the original and the minimized FSM, respectively.

Another example are PDA scripts which, as mentioned above, can be used to create a complex PDF image which consists of two LaTeX tables and a Graphiz graph; an according script and image are shown at the beginning of Sec. 3 or online via the following script link (again, the corresponding plain LaTeX script can be retrieved using the conversion method "Plain generator code"):

Script ID-4135

As a last example of a real application, grammar scripts also can produce complex PDF objects. There, a plain Graphviz script (showing a parse tree or different views of the grammar) is embedded within LaTeX code showing the Grammar definition:

$$G = (\{A, S\}, \{a, b, c\}, P, S)$$
$$P = \{S \to A \mid SS \mid aSb,$$
$$A \to c \mid AA\}$$



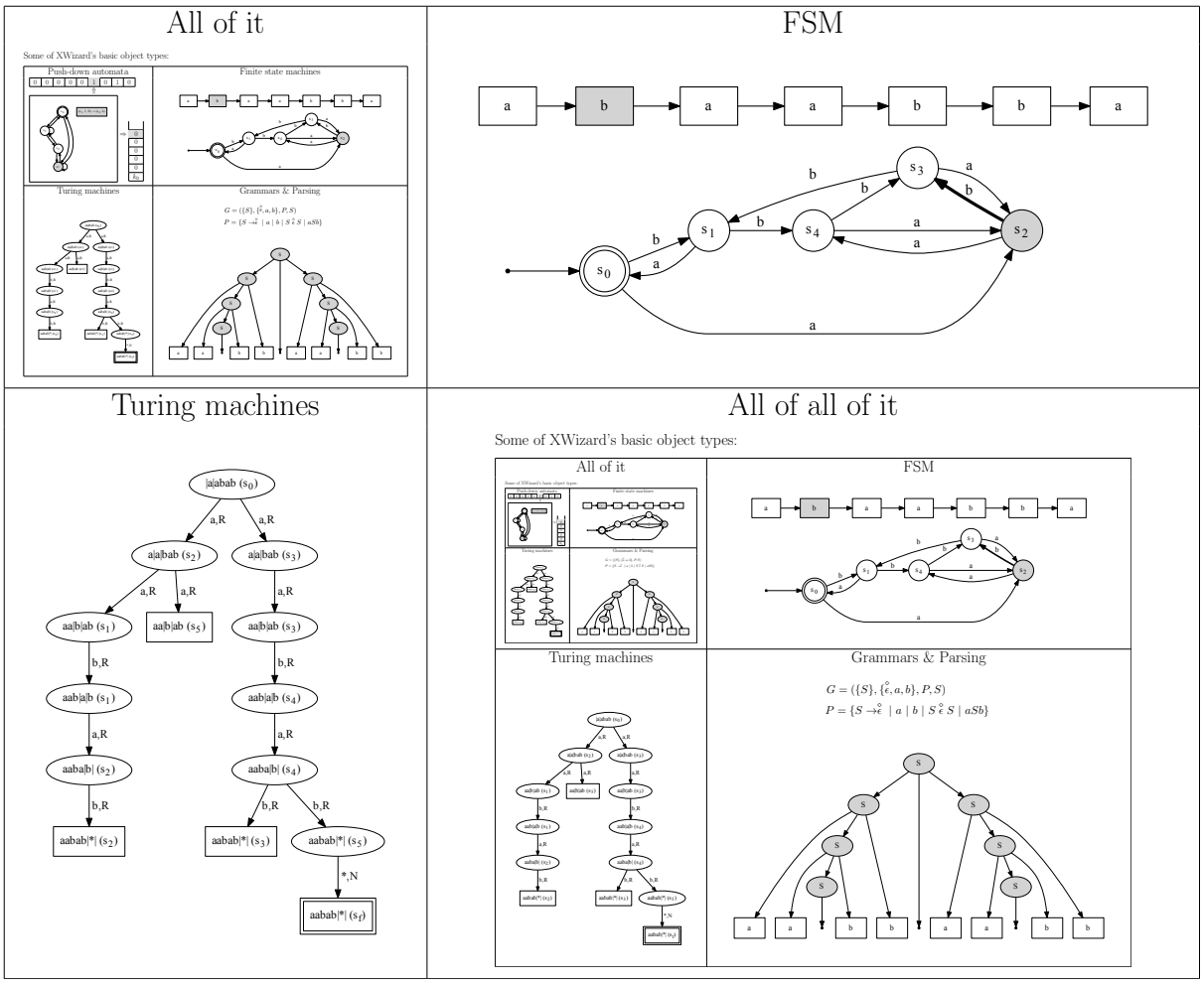
Script ID-11087

The script generating this image is shown below, the according plain LaTeX script can be retrieved as before via "Plain generator code".

```
grammar parse(a, a, <>, b, b, <>, a, a, <>, b, b)--48:
    S => a, S, b | <> | S, <>, S | a | b;
--declarations--
    N=S,A;
    T=a,b,c;
    S=S;
--declarations-end--
```

The following artificial and rather complex example shows how scripts can be embedded recursively to an arbitrary depth within each other (notice that the translation and display of this script can take some 10 to 20 seconds):

Some of XWizard's basic object types:



Script ID-16107

9 Known Bugs, Shortcomings and 'Pitfalls'

- Exam questions and critical scripts should not be made publically available. When doing so, everybody can load the script by typing its ID into XWizard's script area (although somebody guessing correctly the script belonging to his or her exam seems highly unlikely).
- The error messaging system is still being improved. So far, the plain java exception trace is displayed if something goes wrong.
- So far, the script field has no code completion and similar features; they are under construction.
- In the web version of XWizard, an error message may appear without an apparent reason – particularly when many people are working simultaneously with XWizard. We are working on resolving the problem. **A simple workaround is to just repeat the action that caused the problem; usually it will work the second time.**
- After using the "back" button of the browser, downloading the PDF does not work properly anymore. The "Draw!" button has to be clicked first, to reestablish the functionality.
- After using a conversion method, the generated script cannot be made web-free immediatelly, i. e., the "Short URL to this script" method will not work. Workarounds are to either click the "Draw!" button first or to execute the "Short URL to this script" method twice.
- To be continued.

10 Legal Note

The following legal notice is also available (in a possibly more current version) via

`http://www.xwizard.de:8080/Wizz?impressum&lang=eng`

Easy Agent Simulation (EAS) and the embedded Very Fast PDF Generator (VFP; also called PDF XWizard or XWizard desktop version) as well as XWizard, the Web version of the latter, are open source programs; the complete sources, particularly code in Java, SQL, XML, HTML, LaTeX, Graphviz, XWizard-SCRIPT etc., native as well as generated, are protected by the Creative Commons by-nc-sa license, see below.

The complete sources as well as Javadoc for most Java classes are available from Sourceforge:

- EAS (including VFP) on Sourceforge: `https://sourceforge.net/projects/easyagentsimulation`
- XWizard on Sourceforge: `https://sourceforge.net/projects/xwiz`

In a nutshell, you are free:

- to Share – to copy, distribute and transmit the work,
- to Remix – to adapt the work.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following conditions:

- Attribution – You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
- Noncommercial – You may not use this work for commercial purposes.
- Share Alike – If you alter, transform, or build upon this work, you may distribute the resulting work only under the same or a similar license to this one.

No additional restrictions – You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

Have fun with XWizard!

This document has been compiled on September 10, 2016