

MODULAR SOFTWARE ARCHITECTURE FOR AUTHORING MATHEMATICAL CONTENT

STĂNICĂ JUSTINA LAVINIA ^{1*}

ABSTRACT:

The present paper intends to highlight the advantages of choosing a modular architecture in the development of a software platform for teaching math. The software mostly focused on implementing the mathematical functionalities related to an important math concept – functions. Therefore, three main independent components were developed: an equation editor, a mathematical compiler, and a graph plotter. Each component was implemented through an independent user control. The idea behind this was to have a high degree of independence, but yet to assure that the components can work together in order to create a coherent mathematical lesson.

KEYWORDS: *e-Learning, learning component, modular architecture, mathematical compiler*

1. Introduction

Most e-Learning systems are developed without regard of reuse or extension. The modular architecture allows a system to be built from separate independent components that can be combined together. The advantages of using such a design is that components can be removed, replaced, and added without affecting the rest of the system.

The idea of developing a component based software emerged from the necessity to increase the efficiency of educational content authoring. The modular design brings two important advantages: allows the components to be reused, updated, replaced, added, in order to create new learning content, and also satisfies the need to customize training according to some specific user options.

The functionality of the software was divided into independent modules, so that each component implements only one aspect of the mathematical functionality required. So, the mathematical compiler is responsible for functions analysis, evaluation and calculation, the mathematical formulas are input using the equation editor, while the graphics component is used to draw functions graphs.

2. The modular architecture

Developing a software for mathematics raises some difficulties, since it should implement certain functionalities required to illustrate the mathematical concepts. The training application presented here intended to cover three important math functionalities: equation editing, formula interpretation and evaluation, and graph plotting. Consequently, the following software components have been defined:

^{1*} Corresponding author. Lecturer, PhD, School of Computer Science for Business Management – Romanian-American University;

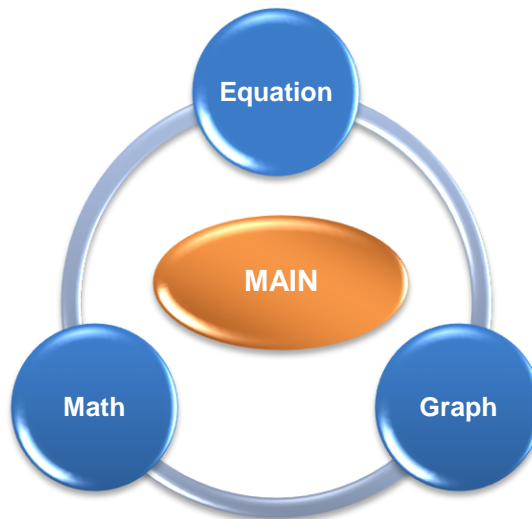


Figure 1. The software modules

The *Main* module implements the core of the application, and includes the graphical interface that allows the user to access the software's functionalities. The application interface is designed as a board (or a panel) on which the user can place different types of objects. Each component is implemented through an independent user control that defines its visual appearance and its functionality. The user can combine the objects in different ways using simple drag and drop operations.

For example, if the user wants to draw the graph of a function, he can first use the equation editor to input the function formula, next he will have to use the mathematical compiler in order to analyze and evaluate the function, and last (once the function is evaluated) he can use the function plotter to draw the graph.

The *Equation* component implements a simple equation editor, thus offering support for editing mathematical formulas and displaying equations. The formulas are written in a scripting language similar to Latex, which is later translated and displayed in visual equational form.

The *Math* module is the most complex component of the application. It implements a syntactic analyzer and a mathematical compiler. It can be used to perform the validation, compilation, and evaluation of any real-valued function. It includes a computation engine as well, which is used to calculate the function values over its definition domain. An additional role of calculating a function's derivative was also implemented.

The *Graphics* component is responsible for calculating, generating, and drawing the graphical representation of real-valued functions in a two-dimensional coordinate system. Besides this, other tools were included, allowing the user to add other graphical elements related to a function graph.

3. The Equation editor






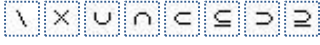



This module offers support for mathematical formulas visual editing. The component uses a scripting language resembling Latex, which is *translated* and displayed in equational visual form. The Latex format was chosen because it provides a platform-independent way for writing mathematical equations. The textual language it uses has many advantages in terms of storing, interpreting, handling, or compressing mathematical formulas. Especially for technical subjects, Latex offers a way of writing complex formulas, and is commonly used in documents with rich mathematical content.




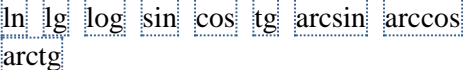
The Equation engine reads and interprets the Latex syntax and generates an image of the mathematical formula in a temporary file. The interface of the component is implemented through a user control in which the formula is edited and the corresponding image is displayed.

The generated images can also be used in other independent training situations. Basically, any software application that allows the display of graphical content, can integrate them as autonomous components; this supports the reuse of learning objects in other platform-independent learning contexts.

However, a software that only allows inputting the mathematical content in Latex format has limited functionality, since not all users are expected to know the Latex syntax. Consequently, the Equation component was equipped with an intuitive easy-to-use interface. The control has a simple editing window that allows the user to either directly write the Latex text or use the toolbar to generate the Latex syntax. The preview image of the formula can be updated continuously or upon request, according to the author's choice. These features make editing formulas easy, since the user doesn't need any knowledge of the Latex syntax.

Table 1. The Equation toolbar

Symbol	Description
	arithmetic and relational operators
	subscripts and superscripts, fractions, radicals, limits
	summations, products, integrals
	common sets of numbers, the empty set
	existence and membership symbols
	sets operations
	lowercase Greek letters
	lowercase Greek letters
	uppercase Greek letters

	implication arrows
	resizable brackets
	editing one- and two-dimensional arrays
	common mathematical and trigonometric functions

4. The Math compiler

The Math module includes an analyzer and a mathematical compiler. This component is used for analyzing, parsing, and compiling the analytic forms of real-valued functions, and for calculating the values of such a function on the definition intervals. The choice of implementation for the graphical interface was to develop an independent user control, which can be reused and included in other software applications.

The control allows the user to directly input the function formulas or to provide the function in Latex format. The expressions are syntactically analyzed, parsed and compiled, and as a result an object of class *Function* is going to be generated. A mathematical function needs to be compiled only once, after which the associated object can be used for repeatedly calculating the function values.

Depending on the input format chosen by the user, the function is analyzed and interpreted by one of the following methods:

- *EvaluateFunction* – it syntactically analyzes the multiple expressions defining the function, validates the domain intervals and checks so they don't overlap; if everything is correct, it generates the *Function* object;
- *EvaluateLatex* – first, the method does the conversion from the Latex format, by decomposing the latex expression into items; next, it analyzes and uses the items to create the associated *Function* object.

The *Function* class covers an important mathematical concept – *the real-valued functions*. Each function can be defined by multiple expressions; also, another essential attribute of a mathematical function is its domain. Each expression has its own definition domain, whose values will be interpreted and tabulated. Consequently, a *Function* object includes other objects that implement the analytical expressions and their associated intervals. For each of the mathematical function's forms, a *Function* object will contain one *ExprPol* object that implements the expression, and one *Interval* object that defines the associated interval.

```
public class Function
{
    public int n; // number of expressions
    public ExprPol[ ] expresia = new ExprPol[N]; // expressions
    public Interval[ ] intervalX = new Interval[N]; // variable X intervals
    public string[ ] s_expresia = new string[N]; // latex string expressions
}
```

```

        public string s_latex;                // function's latex string
    // .....
}
    
```

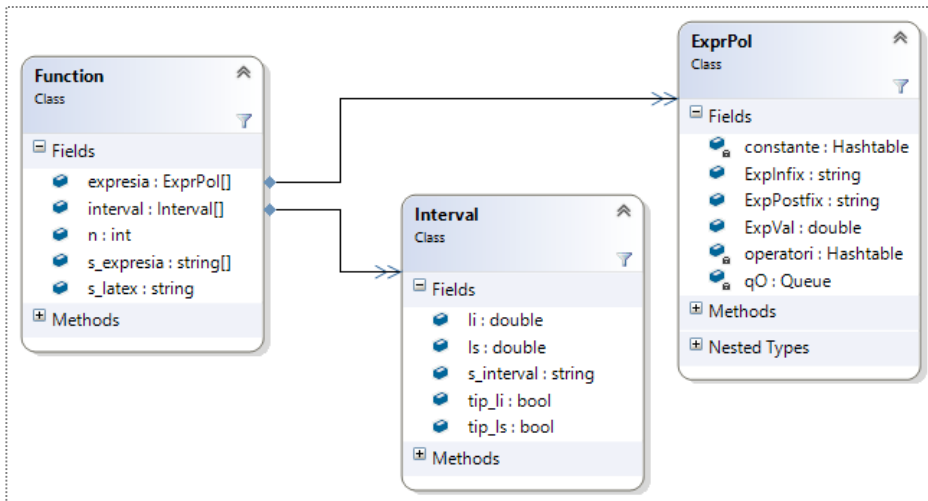


Figure 2. The Function class

The first stage of the analysis includes verifying the definition intervals and making sure they don't overlap; this check is performed by the methods belonging to the *Interval* class. Then, a syntactical examination of each analytical expression is done using the *ValidateExpr* method of the *ExprPol* class. This method carries out an initial evaluation of the expression's correctness in terms of the mathematical functions, operators, variables and constants, and properly balanced pairs of parentheses and braces used.

In the next phase each expression is parsed, compiled, and then calculated and tabulated. Most compilers and interpreters use an evaluation process. The evaluation engine will first check if the expression is in full compliance with some basic rules, and if so, move on to the mathematical analysis. The process consists of using a set of rules to break down the analytical expression into its constituent elements, called items or *tokens*. The analyzer identifies each token and includes it into one of the following categories:

Table 2. Tokens used in expressions

Type	Token	Description
Operand	numerical operand	is a number (an integer or real signed or unsigned value)
	constant operand	defines a mathematical constant and its value; two commonly used constants were defined: π and e
	variable operand	the operand is a variable
Operator	arithmetic operator	this category is specific to the common arithmetic

	function operator	operators that were implemented: +, - (unary and binary), *, / and ^ (power)
	parentheses operator	it specifies that the analyzer has identified one of the implemented math functions
		they are used to indicate a function call or to group expressions (when is necessary to change precedence)

The compiler can analyze and evaluate complex mathematical expressions of real-valued functions, but it can be extended to work with more variables. In fact, the evaluation engine was designed to analyze functions with up to four variables, but tabulating or plotting such a function in more than three dimensions would be difficult. Consequently, the current implementation supports the analysis and compiling of one- or two- variables functions. The syntactic analyzer identifies the variables, and when calculations are needed, the compiler replaces the variables with their corresponding values from the function domain. Any adjustment of one interval will require to automatically recalculate the function for the new values.

The computation engine is responsible for calculations with basic arithmetic operators, which are evaluated according to their precedence. It also supports complex calculations involving mathematical functions that can be combined in any manner. Therefore, the function operators can integrate or be integrated into other mathematical functions or in any arithmetic expressions.

The compiler is programmed to identify a number of common mathematic and trigonometric functions that will be considered function operators. These were defined as internal elements of the evaluation engine. This can be a disadvantage if new mathematical functions are needed, but adding new functions would not require a significant programming effort. Initially, the idea emerged to implement the operators as external components, so that new math functions can easily be added. But the complexity of the evaluation process and the heterogeneous ways mathematical functions behave, motivated the current approach.

Table 3. Function operators

Function			Description
sqrt			calculates the square root of a positive value
root			calculates the n th root of a real number
abs			calculate the absolute value of a number
sin	cos	tg	evaluates the corresponding trigonometric function
arcsin	arccos	arctg	evaluates the corresponding inverse trigonometric function
ln	lg	log	calculates the logarithm (natural, decimal, or to another base) of a positive number

The compiler has two main components: the analyzer and the evaluator. The analyzer first verifies if the expression complies with the basic syntax rules and then generates the collection of tokens the expression contains.

Basically, the expression is written in the infix notation and is converted to the postfix form (reverse Polish notation) using Dijkstra's Shunting Yard algorithm. The algorithm can be used for evaluating and calculating expressions on the fly, but also for converting an infix expression to the Polish postfix form. The idea of the algorithm is to use a stack to temporarily keep each operator until both its operands are processed; after the operands are added to the output queue, the corresponding operator is placed in that queue as well. The output queue will contain the reverse Polish form of the expression. The method that implements the Shunting Yard algorithm and calculates the postfix form is the *CalcInfixToPostfix* method of the *ExprPol* class.

For example, let's consider a simple expression such as $(2 + 5) * 3 - 6 * 4$; obviously, calculating its value is trivial. But for a computer, evaluating the same expression is difficult, since it requires rearranging the items first. The initial expression is given in the infix form and the analyzer will break it down and convert it to the postfix form, so to be easier to evaluate. In fact, for the simple example considered above, the Shunting Yard algorithm will rearrange the expression as $2\ 5\ +\ 3\ *\ 6\ 4\ *\ -$.

The conversion of the infix expression to the reverse Polish form, clearly brings considerable simplification. Starting from the postfix form, the *CalcPostfixToQueue* method (of the *ExprPol* class) generates a queue needed by the algorithm that calculates the value of the expression.

```
protected void CalcPostfixToQueue()
{
    // populates the queue needed for calculations
    string input = ExpPostfix.ToLower();
    string[] token = input.Split(' ');
    for (int i = 0; i < token.Length; i++)
    {
        // populate the queue starting from the postfix form
        if (token[i][0] == 'x' || token[i][0] == 'y')
        {
            // token is a variable
            string var = "";
            if (token[i][0] == 'x') var = "1"; // replace x with x1
            else var = "2"; // replace y cu x2
            TReg o = new TReg(2, var); qO.Enqueue((object)o);
        }
        else if (esteOperator(token[i]))
        { // token is an operator
            TReg o = new TReg(1, token[i]); qO.Enqueue(o);
        }
        else
        { // token is an operand
```

```

        double operand = 0, val;
        if ((val = esteConstanta(token[i])) > 0)
            operand = val;    // is a constant
        else
            try { operand = Convert.ToDouble(token[i]); }
            catch (Exception) { mesaj = "entitate nedefinita: " +
token[i]; }
        TReg o = new TReg(0, operand.ToString()); qO.Enqueue(o);
    }
}
}

```

The need to use a queue for storing the postfix form occurred because the compiler has to operate with variables whose values will be known only when the expression will be calculated.

Next, the engine can easily calculate the value of the expression using a simple algorithm operating with a stack. The algorithm will process all items in the postfix queue: each time an operand is read, it will add it to the stack; and each time an operator comes up, it will extract the required operands from the stack (one or two, depending if it is a unary or binary operator), perform the operation, and push the result back to the stack. At the end, when all items of the postfix queue have been analyzed, the stack should contain a single element representing the result. The algorithm is implemented by the *Evaluate* method of the *ExprPol* class.

This is practically the last stage of the calculation process, which ends with returning the calculated result. The compiler was designed to analyze complex mathematical expressions, and had to take into account many aspects such as tokens variety (functions, operators, variables, and constants), operations and functions precedence and cardinality.

4.1. The derivation engine

Additionally to the main functionalities of analyzing and evaluating functions, the *Function* class also offers support for calculating *functions derivatives*. The derived expressions are built by two auxiliary methods belonging to the *ExprPol* class:

- *CalcDerivString* – first, this method calculates the string associated to the derived expression, according to the derivation rules; at the end the string is also optimized;
- *CreateDerivExpr* – then, this method uses that string to create an *ExprPol* object, which will contain the derived expression.

The *CreateDerivFunction* method of the *Function* class builds the derived function: the derivative will be obtained by deriving all expressions of the initial function on the same definition intervals.

The algorithm for calculating the derivative is similar to the calculation algorithm presented above. It will therefore use the queue that stores the items of the reverse Polish

expression of the initial function. The queue has already been generated when building the initial expression. The derivation procedure additionally uses a stack containing objects of an auxiliary class (*TDeriv*). Each *TDeriv* object is described by three attributes: a flag indicating the item type (operand, operator, variable, or already derived expression), the item being derived, and its corresponding derived element. The algorithms is implemented by the *CalcDerivString* method.

The string returned by the method undergoes an optimization process that will: remove excessive parentheses and brackets; remove the parentheses enclosing single variables or positive values (such as (x) or (1)), if they don't indicate a function call; eliminate unnecessary addition or multiplication operations (especially those with identity elements, e.g. $1 * \dots * 1$, $0 + \dots + 0 \dots - 0$), etc. The optimized string is then passed to the other methods that will first generate the derived expression and finally the derived function.

5. The Graph plotter

The Graphics module is used for drawing graphs of real-valued functions over their definition domain. The mathematical function to be plotted is handled as an object belonging to the *Function* class. The interface of this module is implemented through another independent user control. The plotting algorithm uses a two dimensional coordinate system; the two axes of the coordinate plane has to be calculated first.

The axes are implemented as *FormatAxis* objects; the *FormatAxis* class groups together the characteristics of an axis: minimum and maximum values, scale, unit, and center. For any value on the X axis, the pair $[x, f(x)]$ is calculated on the fly (using the *Evaluate* method of the *Function* class) while the graph is plotted.

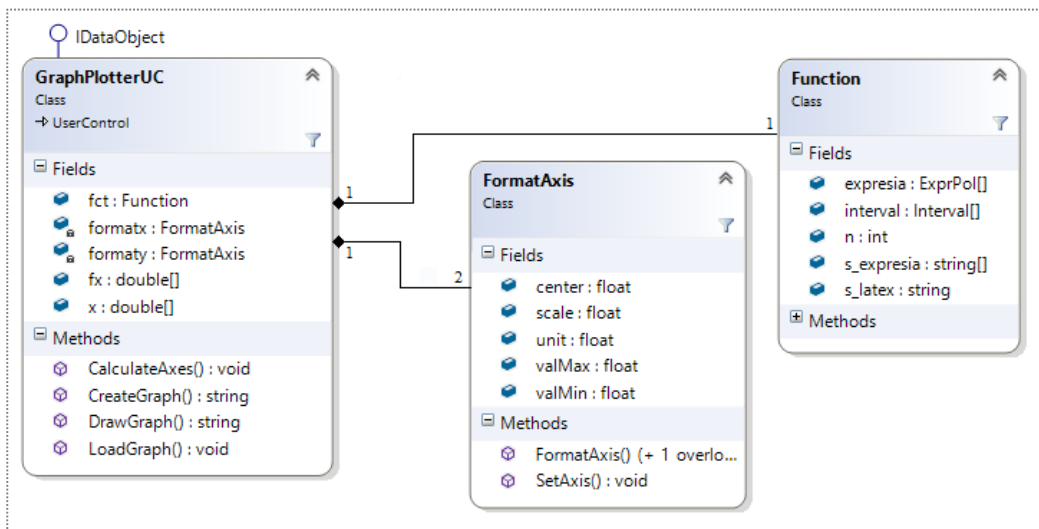


Figure 3. The class implementing the Graphics module

Once the *Function* object is passed to the Graphics module (after being analyzed, compiled, and built by the Math module), the *Evaluate* method is going to be used for calculating the function domain and range. The two axes can now be calculated starting from the domain and range. The *CalculateAxes* method determines the origin, minimum and maximum values, and scale of each axis.

Next, the axes and the function graph are drawn in a panel by the *DrawGraph* method. For the drawing operation, a *System.Drawing.Graphics* object is used; this provides advanced features for elementary graphical operations, text rendering, and handling graphical objects.

The drawing algorithm performs the following steps:

- first a series of graphical objects are instantiated (Pen, Brush, Bitmap, Graphics); these are needed to do the drawing;
- the two coordinate axis are drawn according to the characteristics that were calculated before;
- the function graph is drawn according to the following procedure: for each of the function's formulas, and for each value x in the domain intervals, the value of $f(x)$ is calculated; the pixel corresponding to the $[x, f(x)]$ pair is drawn; in addition, the consecutive pairs belonging to the same domain interval are connected by a line;
- depending on the user's choice, other graphical elements can also be included:
 - the derivative at a point, which is in fact the tangent line to the graph of f at a point x ; the equation of the tangent line is calculated by the methods of the *DerivativePoint* class;
 - the definite integral on an interval, by interpreting it in terms of the area; in this case, the area between the graph and the X axis on the given interval, is going to be shaded;
 - horizontal, vertical or oblique asymptotes of the function; the user is asked to input each asymptote's equation;
- in the final step, the drawing is loaded in the panel and all the unnecessary graphical objects are disposed.

The plotting algorithm had to consider the fact that the origin of the drawing surface (top left corner of the panel) is not the same as the origin of the coordinate system (center of the panel). For this reason, each point's coordinates needed to be adjusted; the graphical coordinates were calculated depending on the logical coordinates $[x, f(x)]$ and the characteristics of the axes as follows:

$$\begin{aligned} x_G &= C_{OX} + x \cdot S_{OX} \\ y_G &= C_{OY} - y \cdot S_{OY} \end{aligned} \tag{1}$$

where:

- (x, y) – the logical coordinates of the point
- (x_G, y_G) – the graphical coordinates of the pixel
- C_{OX}, C_{OY} – the centers of the X and Y axes
- S_{OX}, S_{OY} – the scaling coefficients of the X and Y axes

The function plotter also provides support for resizing and scaling the graph. When resizing, no additional calculations on the function are needed, only the panel which contains the graph is resized and redrawn. On the other hand, the scaling operation requires recalculating the axes, recalculating the function values, and redrawing the graph according to the new axes.

6. Conclusion

The present paper intended to prove that the modular design can successfully be applied to educational software development. The idea of reusing educational components has many benefits for both developing training platforms and creating learning resources.

For the software development, a C# .Net implementation was chosen. For each module, an independent user control project was developed, so that the components can be reused in other software solutions. In other words, each component that is used in authoring math content will be an object instantiated from a prototype.

While assuring the components independence, the software platform should also guarantee the coherence of a Math lesson built from independent objects. This implies that the components constituting the lesson should be somehow related. More than that, the platform allows them to interact with each other, in order to simplify the process of authoring lessons. Each component's interface is able to identify the other objects it can interact with in a predefined manner.

So as not to affect components independence, all interactions use intermediate objects that handle the communication between source and destination. The interrelated components only work with the mediator object, and don't need to know the implementation details of the object they interact with. This idea guarantees the component decoupling and thus a high degree of reusability, which means a higher efficiency in creating e-Learning content.

7. Bibliography

[STAN14] Stănică J.L., Crișan D.A., *Learning Object Architecture for Dynamic Development of Mathematical Content*, Journal of Information Systems and Operations Management (JISOM), Vol. 8 No. 2 / December 2014, 2014, pp. 297-306, CNCSIS B+ journal, indexed in: Index Copernicus, ProQuest, EBSCO, RePEC, ISSN 1843-4711;

[STAN12] Stănică J.L., Crișan D.A., *Dynamic Development and Assembly of Learning Objects in a Math Learning Environment*, Journal of Information Systems and Operations Management (JISOM), Vol. 6 No. 1 / May 2012, pp., ISSN 1843-4711, 2012

[ALLE10] Allen C.A., Mugisa E.K., *Improving Learning Object Reuse Through OOD: A Theory of Learning Objects*, Journal of Object Technology, vol. 9, no. 6, pp. 51–75, ISSN 1660-1769, 2010, [http://www.jot.fm/issues/issue_2010_11/article3.pdf]

[OVER10] Overton L., Howarth N., Merritt R., Basiel A., Howarth M., *Delivering Results with Learning Technologies in the Workplace*, Towards Maturity Enterprises Ltd, on behalf of BECTA, pg. 76, online, 2010,

[http://www.e-learningcentre.co.uk/Resource/CMS/Assets/5c10130e-6a9f-102c-a0be-003005bbceb4/form_uploads/delivering_results_with_learning_technologies.pdf]

[STAN11] Stănică J.L., Crișan D.A., *Framework for Flexible Reuse and Assembly of Learning Objects – A Pilot Project*, Journal of Information Systems and Operations Management (JISOM), Vol. 5 No. 2.1 – Special Issue / December 2011, pp. 478-484, ISSN 1843-4711, 2011

[ELLI08] Elliott K., Sweeney K., *Quantifying the Reuse of Learning Objects*, Australasian Journal of Educational Technology, vol. 24(2), pp. 137-142, ISSN: 1449-5554, ISSN: 1449-3098, 2008, [<http://www.ascilite.org.au/ajet/ajet24/elliott.html>]

[SMEU08] Smeureanu I., Dârdală M., Reveiu A. *Component Based Framework for Authoring and Multimedia Training in Mathematics*. Proceedings of World Academy of Science, Engineering and Technology, vol. 29, pp. 230-234, ISSN 1307-6884, 2008

[WILE07] Wiley D.A., *The Learning Objects Literature*, in Wiley Blog: “Iterating toward openness”, pg. 10, online, 2007, [<http://opencontent.org/docs/wiley-lo-review-final.pdf>]