# On-line Debugging Methods and Tools in Movie-based Programming

DMITRY VAZHENIN, ALEXANDER VAZHENIN
Graduate School Department
University of Aizu
Tsuruga, Ikki-machi, Aizu-Wakamatsu, Fukushima
JAPAN
d8052102@u-aizu.ac.jp, vazhenin@u-aizu.ac.jp

*Abstract:* - Movie-based programming focuses on a representation of computational process similar to a movie demonstration. It is possible by correlating animation frames with solution steps. Like in conventional movie frame is representing a part of a process. Typically, one frame corresponds to logically completed part of an algorithm like one iteration of iterative method. By its nature the animation frame is an image, and the execution frame is a source code snippet, both are produced by corresponding component of a system. Both a movie and program can synchronously be generated and debugged. This allows that debugging operations can be implemented in any stage of the movie/program design. In this paper, we discuss main stages of the movie/program design and propose the new visual debugging schemes allowing to implement debugging process at all design stages. We demonstrate how some typical programming mistakes can be easily avoided or discovered/fixed by means of the movie-based programming software that integrate traditional debugging attributes such as breakpoints with multimedia formulas backtracking methods.

*Key-Words:* - Visual Programming, Movie-based programming, Debugging of Algorithms and Programs, Matrix Computing, Formula Backtracking

## 1 Introduction

Debuggers are universal tools for understanding what is going on when a program is executed. Using a debugger, one can execute the program in a specific environment, stop the program under specific conditions, and examine or alter the content of the program variables or pointers. Traditional command-line oriented debuggers allowed only a simple textual representation of the program variables (program state) [1]. Textual representation did not change even when modern debuggers came with a graphical user interface. Although variable names became accessible by means of menus, the variable values were still presented as text, including structural information, such as pointers and references. Likewise, the program execution is available only as a series of isolated program stops. In contrast to traditional textual programming languages, where multi-dimensional structures are encoded into one-dimensional strings according to some intricate syntax, Visual Programming essentially remove this layer of abstraction and allows the programmers to directly observe and manipulate the complex software structures.

Compared to traditional debuggers, the techniques of visual debugging allow quicker exploration and understanding of what is going on in a program [2]. As shown in [3], the visual languages variety corresponding environments may be classified according to the types and extents of visual expressions used including possibility to involve visual debugging mechanism.

The animated visual 3D programming language SAM (Solid Agents in Motion) for parallel systems specification and animation was proposed in [4]. A SAM program is a set of interacting agents synchronously exchanging messages. The SAM objects can have an abstract and a concrete, solid 3D presentation. While the abstract representation is for programming and debugging, the concrete representation is for animated 3D end user presentations.

Tanimoto [5] states that “Data Factory” indicates visual dataflow environment. In this model, users can control icons prepared with mathematical operations in the layout where mathematical methods are connected to others like belt conveyers in the factory. JAVAVIS was developed as a tool to support teaching object-oriented programming concepts with Java [6]. This tool monitors a running Java program and visualizes its behavior with two types of UML diagrams, which are de-facto standards for describing the dynamic aspects of a program, namely object and sequence diagrams. We can characterize most of the mentioned systems as very special. They are mostly focused on solving specific problems.

A declarative and visual debugging environment for Eclipse called JIVE presented in [7]. In contrast with traditional step-by-step debugging procedures, authors present a declarative approach consisting of a flexible set of queries over a program's execution history as well as over individual runtime states. This runtime information is depicted in a visual manner during program execution in order to aid the debugging process. The current state of execution is depicted through an enhanced object diagram, and a sequence diagram depicts the history of execution. Authors present details of the JIVE architecture and its integration into Eclipse.

To conclude this brief review, we would like to mention that approaches described are mostly focused on improving the run-time debugging operating with the executable code. The embedding the debug operations at the earlier stages of the software design still needs to be supported by new programming tools.

Multimedia approach for interactive specifications of applied algorithms and data representations is based upon a collection of computational schemes represented in the "film" format proposed in [8]. In [9,10], we presented an extension of this approach called the Movie-Based Programming. The programming process is in manipulating with special movie-program objects (MP-objects) generating *automatically* a part of an executable code as well as producing frames, which are *adequate* to the code generated. Both movie and program can *synchronously* be generated and debugged. This allows that debugging operations can be implemented in any stage of the movie/program design. In this paper, we discuss main stages of the movie/program design and propose the new visual debugging schemes allowing to implement debugging process at any such a stage. We demonstrate how some typical programming mistakes can be easily avoided or discovered/fixed by means of the movie-based programming software that integrate traditional debugging attributes such as breakpoints with multimedia formulas backtracking methods.

The rest of the paper is organized as follows. In Section 2, we discuss a concept of the Movie-based Programming and show main elements of Movie-based Multimedia Environment for Programming and Algorithms Design. The third section describes debugging movie-based algorithms and programs together with the main user's design. In Section 4, the movie-program execution environment and run-time debugging tools are demonstrated. The last section contains conclusion and future research topics.

# 2 Movie-based Programming Concepts

## 2.1 Main Elements and Definitions

***Movie-based programming*** is mainly focucing on a representation of computational process similar to a movie demonstration by correlating **animation frames** with solution steps. Like in a conventional movie, a frame is representing a part of a process. Typically, one frame corresponds to a logically completed part of an algorithm like one iteration of iterative method (Fig.1).

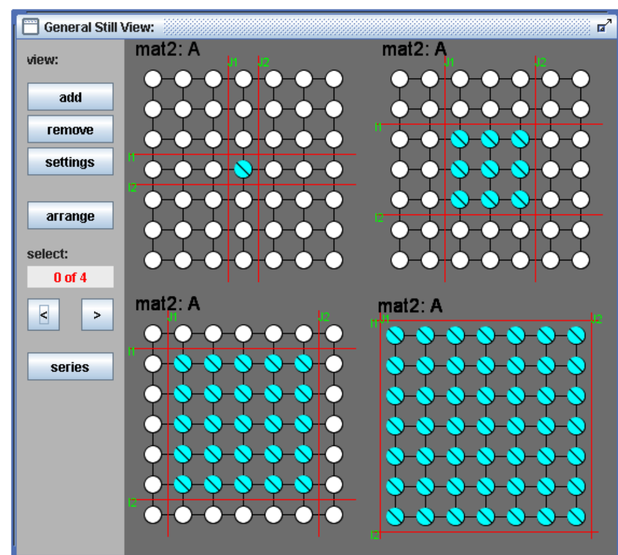


Fig.1 Algorithmic movie example

According to its nature, we distinguish two types of frames. The ***animation frame*** is actually an image that can be animated in order to improve the user'r perception. According to matrix structures, Each frame highlights and flashes some elements of parameterized matrix structures defining operations or formulas. Different operations can be coded by different colours. Special ***Control Lines*** (I1, I2, J1, J2 in Fig. 1) are used to reference these areas of activities. They can change their placement inside matrix during frame transitions.

The ***execution frame*** is a source code snippet. As shown below, both are produced by a corresponding component of system. We are calling such a basic component a Movie-Program (MP) ***Metaframe*** because of presence of these dualistic features.

A ***Film*** (or ***MP-film***) is a sequence of metaframes. It is possible to have a film collection in which each film is independent from other films, but films can be nested. Any film is able to produce animation and execution frames. The Fig. 2 shows an example of the animation frames producing process.

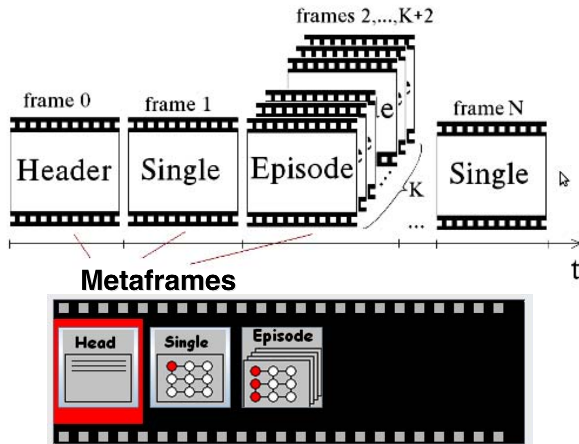

Fig.2 Animation frames producing process

A metaframe is a basic producer of executing and animation frames depending on the type and specifications. Table 1 contains a classification of main metaframes.

Table 1 Classification of Metaframes

| Icon | Metaframe Definition |
|---|---|
| Head | ***Head Metaframe*** is the first (topmost) metaframe in any Film containing description of data structures and variables used in a current film. |
| Single | ***Single Metaframe*** is a metaframe which represents the one algorithmic step and generates one animation frame. |
| Episode | ***Episode Metaframe*** produces a series of frames. The main feature of episode is that the same operations should be implemented in all frames until episode condition is true. It is primarily used to specify iterative process since it produces instances of the same frame with different parameters. |
| If...else a>0 and ... 1 2 | ***IF-Metaframe*** is to skip or process selected groups of stills. The user should specify a logical conditional expression as well as mark stills that will be processed for true and false cases correspondingly. |
| While a>0 and ... 2 | ***WHILE-Metaframe*** is to repeat the processing of stills marked while a condition is true. |
| Call Film 3 | ***CALL-Metaframe*** is to pass processing to other MP-film. |

## 2.3 System Architecture

Fig. 3 shows main components the Movie-based Multimedia Environment for Programming Algorithm Design. To design the movie-based algorythms/programs, the user should work with the ***Film Editor*** in order to specify MP-film metaframes and their parameters. The ***Movie-Generator*** and **Player** generates a series of animation frames as shown in Fig. 2. The ***Executable Code Generator*** produces an executable code in two modes. The first mode called a Movie-based Program has a structure exactly corresponding to the algorithmic movie. In the second mode EC-generator produces the final executable program according to the target machine requirements. ***Program Executor, Debugger and Data Visualizer*** are to support executing and debugging the Movie-based program. The Manager controls all system operations and data access procedures. The user can design his/her algorithm using metaframes and films stored in the ***Metaframe*** and ***Film Database***. It is also possible to customize this library by adding user-designed components.

## 2.3 Algorithm/Program Design Stages

The general scheme of the movie-based algorithm/program design includes the following four stages:

1. Creating the ***prototype MP-film*** which is a shape of computing. It defines only a distribution of computations over metaframes. In other words, the it defines only a distribution of activities on structure elements in time.
2. Creating the ***complete MP-film*** according to the concrete application requirements. For each metaframe, it is necessary to attach ***computational formulas*** to active structure nodes defined in a prototype film. To provide debugging procedures, these formulas are extended by special breakpoint operators and references allowing to monitor structure data as well as check correctness the index expressions defining references to structure elements.
3. ***Generating, Executing and Run-time Debugging*** of Movie-based Program using special monitor controlling the program execution and visualizing matrix data.
4. ***Exporting algorithmic movie and program*** to the target machine.

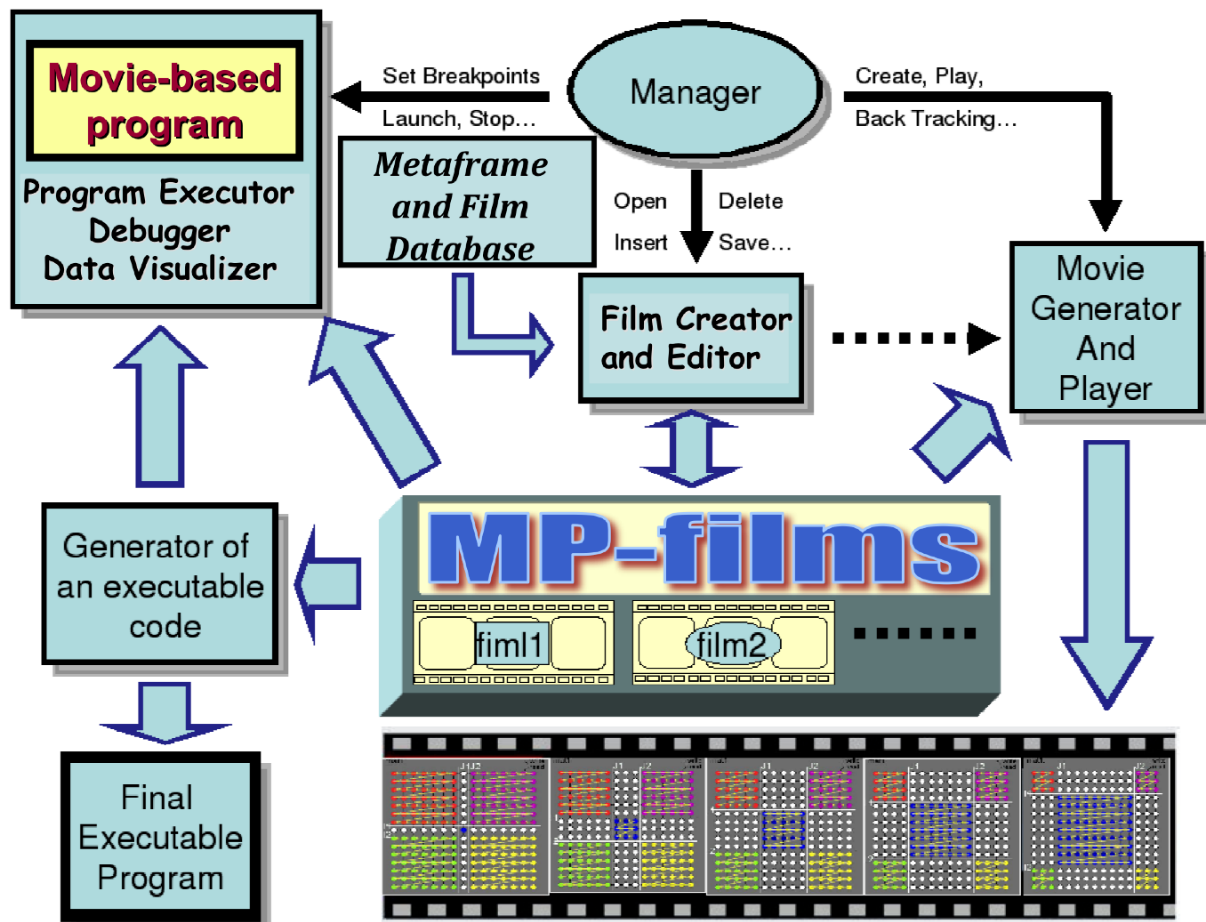


Fig. 3 Movie-based Programming Architecture

## 3 Design/Debugging Metaframes

In this paper we are focused on design and debugging matrix algorithms that operate with matrices and vectors. Therefore, our following analysis will be based on basic features of these 2D-structures.

### 3.1 MP-structures, Control Lines and CF-formulas

Each metaframe contains a set of traversal schemes specifying coloured domains in corresponding structure. Each schema has its own colour and ***formula sequence*** attached to this colour. The same nesting scheme is always reflected in the corresponding program source code. Each still produces one or several static frames representing skeleton steps of computation and hiding formulas.

Actually, any algorithm represents data structures as well as an order of operations or ***formulas*** implementing on structure elements or nodes. Therefore, MP-structures are other important components of each MP-film. Each MP-structure includes the following attributes:

1. The ***unique structure name*** is to identify a structure from other structures.
2. ***Parameters or variables*** are for defining structure size, for example, number of rows and number of columns for matrix structures. Importantly, these parameters have two values. The first value is used for an animation movie, and the second is used for generating program.
3. ***Structure control components*** are used to reference activities areas inside a structure. This means that control objects divide a structure into zones each of which can have individual color. Different colors mean that different operations can be implemented on the corresponding nodes. For matrices, we have a deal with vertical (J-lines), horizontal (I-lines) and diagonal lines (D-lines). They can change their placement inside matrix during frame transitions.
4. Structure variables can be simple doubles, integers, etc. as well as a composite type like strings, complex numbers, etc. Each structure can have several variables.

Fig. 4 shows a screenshoot of the system GUI and illustrates the mentioned above definitions. The main order of operations is defined by metaframes generating computational steps and corresponding movie frames. The Control Flow Formulas or ***CF-formulas*** are introduced to coordinate operations between frames as well as program the control lines behavior. As shown in Fig. 4, attributes and features of the matrix structure can easily be presented. In this example, there are a matrix with sizes of 8x9, nodes arranged into two triangular and two diagonal shapes as well as control lines (I1,I2, J1 and J2). This structure has name "SLAE" with the variable A declared, and two domains composed by a corresponding color. Each domain consists of two sub-domains which nodes have the same color and are aligned to corresponding control lines and structure bounds. The placement of each sub-domain is defined by positions and combination of control line as well as a shape type like triangle, rectangle, line, diagonal, etc.

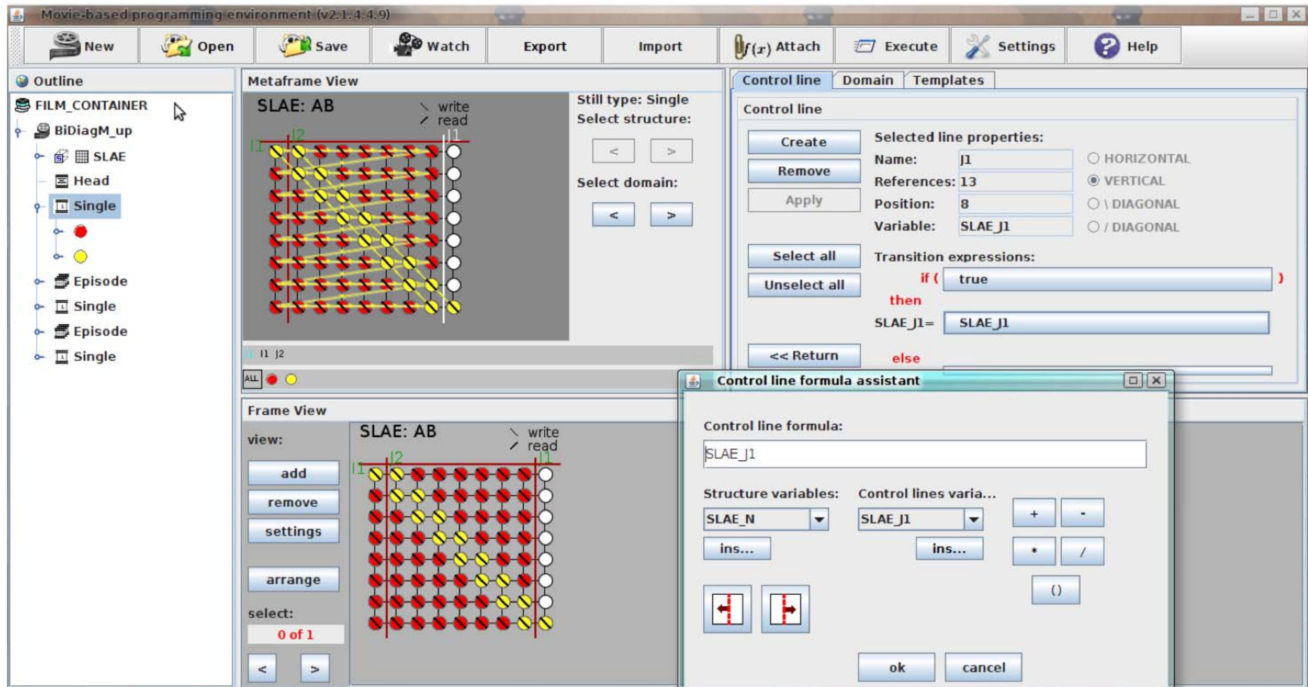


Fig. 4 Metaframe editing window

Involving rules showed in Fig. 5, the user can specify control lines behavior as well as define a corresponding number of MP-frames or program iterations will be generated. The user should specify the Episode Rule that provide condition to finish episode. The system check this condition and trying to calculate the number of frames generated. Is this number is infinite or more than maximum defined by the user the system will stop frame and executable code generation.

The important debugging components is the tracing of all control line positions. The user can watch animation frames visualizing control line movement during frames transitions, or use the compact representation of these behaviors in graphical form. In the first mode, the user can watch all animations or provide one-by-one scrolling of all frames in the corresponding episode. In the second approach, the system paints graphics reflecting numerical representations of the control line positions during frames transitions.

***CF_ID*** – Control Line Name
1. Initialization Rule (Start position):
***CF_ID = <expression>;***
2. Transition Rule (Next position):
***if (<condition>)***
***then CF_ID = <expression 1>;***
***else CF_ID = <expression 2>;***
3. Episode Rule (How to finish episode):
***If(<condition>) then {Generate Next Frame};***
***else {Finish Episode};***

Fig. 5 Control Flow Rulers

## 3.2 References and Tracing of C-formulas

Actually, CF-formulas are to create a shape of computing and define a distribution of computations over MP-nodes. In other words, an algorithmic movie shows data structures and some activities on these structures. To precisely specify their activities according to the apllication feature, it is necessary to attach arithmetical and/or logical formulas to corresponding nodes. These formulas are called ***Computational formulas*** or ***C-formulas***. We define a C-formula as a subprogram containing a sequence of arithmetical and logical expressions. Each C-formula includes the following components: ***MP-expressions***, ***Control structures, Regular text***. ***MP-expressions*** are to specify data access and operations on structure nodes (Fig. 6). 0

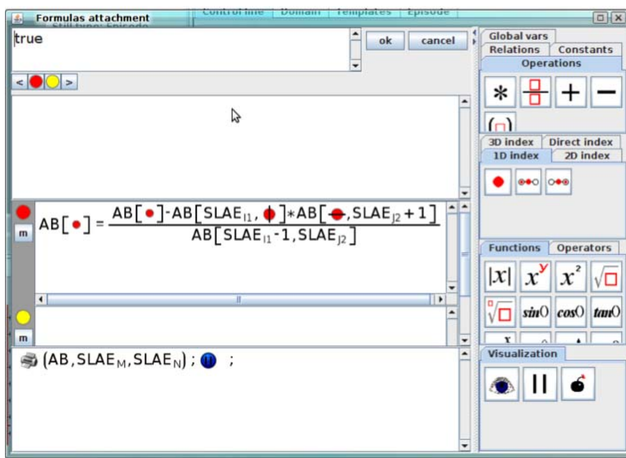


Fig. 7 C-formulas Attachment Interface

The C-formula notation is close to the conventional mathematical expressions. We are enhancing C-formulas by using special multimedia attributes like images, symbols and tables in order to improve the formula perception (Fig. 7).

$$A[\bullet] = \sum A + \frac{A[\cdot]}{2}; \quad A[\bullet]=0.5; \quad B[\bullet]= \frac{1}{4}\Sigma A[\cdot]; \quad A[\bullet]= \frac{1.0}{1.0+\bullet+\bullet};$$

Fig. 8 C-formulas Examples

Special attention in our approach is devoted to the debugging of references that define placement or coordinates of elements in the matrix structure. To define references, it is necessary to specify index expressions that make difficulties for debugging for complex index expressions. Accordingly, we propose understandable references by visual specification of index expressions that are useful to specify complex index expressions in compact and accessible forms.

As shown in Fig. 9, visual references are formed according position of the current matrix element that can be divided into three groups. Group 1 contains basic references like take column number of current element. Group 2 contains reference operators for specifying placement neibors of element on vertical, horizontal, and diagonal positions. Group 3 contains local group operations on neiboring elements like sums, products, etc. The operators proposed have a compact and understable form that can simplify the debugging process.

| Groups | Icons | Semantics | Index expression for references |
|---|---|---|---|
| Group1 | | This icon specifies current position. | A[i,j] |
| | | This icon specifies row number. | A[i] |
| | | This icon specifies column number. | A[j] |
| Group2 | | This icon specifies upper position from current position. | A[i,j-1] |
| | | This icon specifies right position from current position. | A[i+1,j] |
| | | This icon specifies upper right position from current position. | A[i+1,j-1] |
| Group3 | | This icon specifies vertical, horizontal positions, and their operations. (coefficient ±1) | Upper position: (+1)*A[i+1,j-1] Lower position: (+1)*A[i,j+1] Right position: (+1)*A[i+1,j] Left position: (+1)*A[i-1,j] |
| | | This icon specifies upper right, lower right positions, and their operations. (coefficient ±1) | Upper right position: (+1)*A[i+1,j-1] Lower right position: (-1)*A[i+1,j+1] |

Fig. 9 Visual references to matrix elements

During formula attachment, system provides visualizing and controlling all references to the structure elements. This allows debugging film structure and formulas activity during design-time.

The ***formula tracing technique*** is used visualizing nodes referred by a formula on a particular frame (Fig. 10).

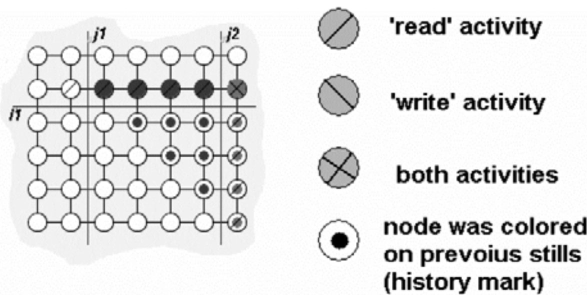


Fig. 10 Formula Tracing

Each C-formula is parsed in order to extract indices of nodes where data access is performed. Those nodes are marked as active with 'read', 'write' and 'read-write' access type. This allows visualizing any wrong access even before program execution (Fig. 11).

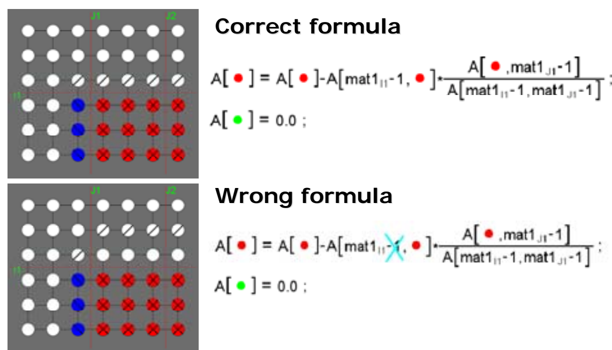


Fig. 11 Formula Tracung Example

# 4 Run-Time Debugging

The Run-Time Debugging Monitor is a software component to verify a movie-based program during its run-time implementation.

## 4.1 Debugging Operators

The data-flows and control-flows are conducted using special debugging operators. There are three type of debugging operators: ***Visualization Invoker*** allows showing an image reflecting the data. ***Pause*** is to set a breakpoint, and ***Kill*** stops program execution (Table 3). Visualization Invoker operator has a variable name and structure size as parameters. Pause and Kill operators have no parameters.

Table. 3 Debugging operators

| Operation | Image | Notation in formula |
|---|---|---|
| Visualization invoker | | (AA, $mat1_M$, $mat1_N$) ; |
| Pause | | ; |
| Kill | | ; |

Those operators can be inserted into movie-program by using formula attachment interface (Fig. 12). The user can specify an operation by clicking focusing a particular formula input area in the left panel an clicking appropriate button from a right panel. If Visualization Invoker is specified, the system also offers to choose the name of variable to be visualized.

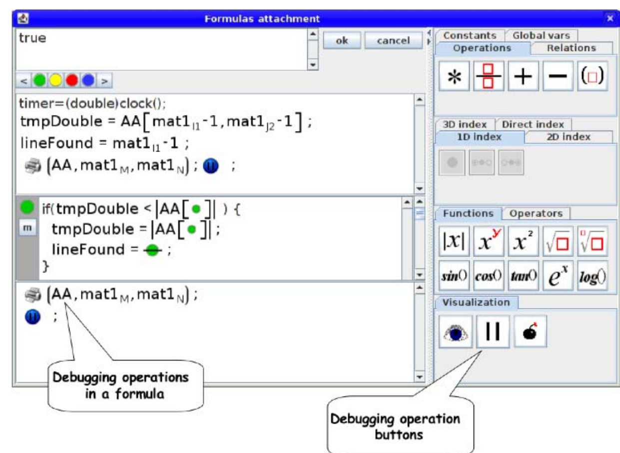


Fig. 12 Computational formulas and debugging operators in the formula attachment interface

## 4.2 Execution Environment

Run-time debugging monitor uses a special interface panel to demonstrate execution status of a program and the status of data in a program at the particular moment (Fig. 13). This panel contains the information about a metaframe number and frame position inside of metaframe, to point the user to a place of a movie-program where the stop is occurred. When a breakpoint have reached, the program is paused, and the data visualization panel is invoked reflecting the most recent state of data structure specified by the the most recent Visualization Invoker operator. It means that Visualization Invoker operator should be specified before any breakpoint in a formula. There are two

ISSN: 1792-4863

ISBN: 978-960-474-231-8

layers in the panel: control lines and matrix structure visualization layers.

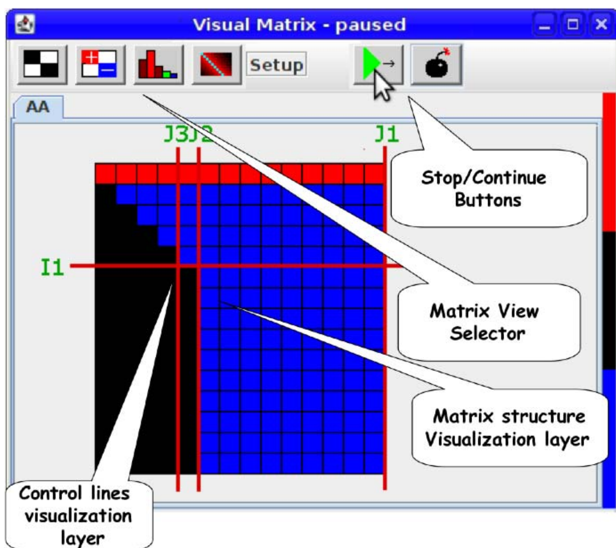


Fig. 12 Run-time Debugging Monitor Window

Matrix structure visualization layer shows a grid with colored elements in cells. Those cells are reflecting the data values of structure variables according to a scale (Fig. 13, right). Four scales are used to demonstrate the data: binary portray, sign portray, value spectrum portray and diagonal dominance portray. Control lines visualization layer demonstrates actual positions of control lines of a structure on current frame. The combination of two frames allows the user to understand the data-flow and control-flow consistency of a debugging program.

## 5 Conclusion

The Movie-based Programming allows manipulating with MP-metaframes that are objects allowing *automatically* and *synchronously* generating movie frames and *adequate* executable code. As we demonstrated, debugging operations can be implemented simultaneously with designing the MP-film. The important feature of our approach is in separating design/debugging procedures of the algorithm structure (prototype film) from the final application area (complete film). This allows designing/debugging the most common part of different applications and reducing the programming efforts. The C-formula debugging environment provides additional possibilities to collect and visualize a history of references to the structures and data. The Run-Time Debugging makes possible verifying a movie-based program data-flow using corresponding breakpoints. The visualization tool allows to debug applications working with the huge amount of data. Recently, the presnted environment is used in “Software engineering” educational course.

Our future investigations are oriented to extend a number of structure types by including trees, linked lists, 3D-structures.

*References:*

[1] Th. Grotker, Ul. Holtmann, H. Keding, M. Wloka, *The Developer's Guide to Debugging*, Springer Verlag, 2008.

[2] http://encyclopedia2.thefreedictionary.com

[3] P.T. Cox, Visual programming languages, *Encyclopedia of Computer Science and Engineering*, B.W. Wah (Ed.), John Wiley & Sons Inc., Hoboken, 2008.

[4] C. Geiger, W, Mueller, W. Rosenbach, SAM - An Animated 3D Programming Language, *Proc. of the 1998 IEEE Symposium on Visual Languages, Halifax Canada*, 1998, pp. 228 - 235.

[5] S. Tanimoto, Programming in a Data Factory, *Proc. of Human Centric Computing Languages and Environments,* Auckland, 2003, pp. 100-107.

[6] R. Oechsle, and T. Schmitt, JAVAVIS: Automatic Program Visualization with Object and Sequence Diagrams Using the Java Debug Interface (JDI), *LNCS*, Vol. 2269, Springer-Verlag, 2002, pp. 1-15.

[7] J. Czyz, Bh. Jayaraman, Declarative and visual debugging in Eclipse, *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange, Montreal, Quebec, Canada*, 2007, pp. 31-35.

[8] N. Mirenkov, A. Vazhenin, R. Yoshioka, Ts. Ebihara, at al., Self-Explanatory Components: A New Programming Paradigm, *Int. Jour. of Soft. Eng. and Knowledge Eng.*, vol. 11, No. 1, 2001, pp. 5-36.

[9] D. Vazhenin, A. Vazhenin, and N. Mirenkov, Movie-based Multimedia Environment for Programming and Algorithms Design, *LNCS*, Springer-Verlag, Vol. 3333, Part III, 2004, pp. 533-541.

[10] D. Vazhenin, A. Vazhenin, MP-templates Operating Toolkit in Movie-based Programming, *Proc. of Japan-China Workshop on Frontier of Computer Science and Technology*, Nagasaki, Japan, 2008, pp. 67-73.