Augmenting Authoring of Adaptation Languages via Visual Environments

Jan D. Bothma and Alexandra I. Cristea

Department of Computer Science, University of Warwick, Coventry, CV4 7AL
`{J.D.Bothma, A.I.Cristea}@warwick.ac.uk`

Abstract. Adaptive Educational Hypermedia (AEH) ideally allows for the delivery of the right information to the right student. Strategies for adaptation can be described via the LAG language. The PEAL tool offers many common programming environment features to ease development in LAG but authoring adaptation strategies is still considered time-consuming and difficult, especially for the inexperienced author. Visual Programming Environments have been shown to ease entry to programming for a new platform for beginner and experienced programmers. In this paper we ease entry to adaptation strategy authoring by providing a visual environment for creating strategies alongside the text-based programming environment of PEAL which propagates changes from either representation to the other. This allows the lay person author to create adaptation strategies with little experience of programming or the LAG language while becoming familiar with LAG at their own pace, thus lowering the threshold for authoring in the complex environment of Adaptive Hypermedia.

Keywords: Adaptive Hypermedia, Visual Programming, LAG

1 Introduction

Adaptive Hypermedia (AH) [7] adapts content for any given user according to what is called the *User Model*. This stores information about a particular user to adapt to their needs such as their knowledge level, interests, goals, preferred learning styles, situational parameters, etc., and thus delivers information that is appropriate to them. The LAOS Framework [12] separates the different aspects of Adaptive Hypermedia into the Domain, Goal and Constraints, User, Adaptation and Presentation Models.

There are currently few tools that deliver adaptive hypermedia, and even fewer that provide authoring support. My Online Teacher (MOT)[14] is one of the only tools to provide a generic platform for authoring and reusing adaptive educational hypermedia. It is supported by two export formats. The *Common Adaptation Format* (CAF) [14] stores the static content which makes up the relevant content of the Domain Model, and the lesson structure in the Goal and Constraints Model. The *LAG language* [13] implements the Adaptation Model. These formats offer a compact way of storing and exchanging adaptive hypermedia information between systems and together allow a compatible Adaptation

Engine to provide an online lesson which adapts to users according to their User Model.

In this paper we describe the research towards improving the authoring process for adaptive hypermedia. This includes improving the authoring tools that can deploy the following main methods:

- Help and support tools
- Function distance[1]
- (Semi-)automating the authoring process[2]
- Visualisation of authoring steps

Here we are describing the design and implementation of a tool aimed at weak (beginner) programmers and non-programmers which aims to help them be able to create relatively complex adaptation strategies for web presentations. Thus, we aim to lower the threshold for the lay person in creating personalization and adaptation scenarios. For this purpose, we aim specifically at one of the most challenging areas of improvement – that of visualization of the authoring steps. We will focus specifically on creating visualizations of the adaptation behaviour description, and not the more straightforward (and more commonly seen) content description.

As the MOT tool already presents a developed environment for generic authoring, as well as a language set for both content and adaptation specification, the current visualization improvements as proposed in this paper are based on the MOT tool. Thus the tool presented here is a visual representation of and equivalent to the LAG adaptation language; directly based on its grammar and semantics. It does not attempt to develop a new adaptation programming paradigm, as is being undertaken by the GRAPPLE project [15].

The remainder of this paper is organized as follows: Sect. 2 discusses the problem description; Sect. 3 explains the background to this project and related work; Sect. 4 describes the development and implementation of our solution; Sect. 5 documents our evaluation of the solution and Sect. 6 concludes with our findings.

2 Problem Description

A recently-completed dedicated LAG editor, the Programming Environment for Adaptation Language (PEAL) has been evaluated by Cristea et. al [13] where a sample of the target users of this editor, experienced programmers, unanimously agreed that it was better for authoring in the LAG language than the existing text editing tools. However, non-programmers and programmers new to LAG and the MOT system can still find programming in the LAG language intimidating. Cristea et al. noted in [13] that the existing strategy creation tool in MOT is

[1] Clicks needed to perform an action from a given state

[2] e.g. automatic content generation, automatic link generation, etc.

out of date, and that a visual tool for developing strategies is desired to lower the threshold for authoring in the LAG language.

It is important for the authoring environment to be online [13]. A web-based authoring environment offers wide availability and eases collaboration. PEAL is a web-based programming environment and has several valuable features which can be built upon to achieve our goals. This includes online private and shared storage of adaptation strategies which allows collaboration and reuse, meaning authors do not have to start with completely blank strategies but can work from an example. The use of PEAL's LAG parser for converting adaptation strategies to our visual representation is discussed in Sect. 4.

The highest level of the LAG Framework [11] is the 'Adaptation Strategy'. LAG strategies have a plain text description which allows authors to (re-)use existing strategies without needing any knowledge of LAG. Unfortunately these descriptions are rarely verbose enough, when written at all, to allow reuse by the lay author. It is therefore important to make writing the description an integral part of the authoring process and as easy and informative as possible.

3 Background

The field of Visual Programming Environments is heavily researched [16]. Like parser generators, visual programming environment generators are available [6, 10]. Unfortunately the formats of these outputs are difficult to port to the strict web browser-based format desired for this project. This means that even if a useful visual environment for LAG could be produced by such an automated tool, significant effort would be required to port the result to the web browser.

The LAG creation tool which exists within MOT is out of date in terms of LAG grammar [13]. It is also based on older website interaction methods which require reloading the page to update content. JavaScript engines in web browsers have improved significantly since the MOT tool was implemented, offering efficient and intuitive interactive application capabilities within web browsers.

The Adobe Flash [1] plug-in used by the GRAPPLE project and Java Applets used by AHA! are often used for rich interactive online applications. However, the advances in web technologies mean that similarly rich and efficient interfaces can now be created without plug-ins, offering capable applications with minimal software requirements.

Casella et. al [8] define a collection of visual languages implemented in the SEAMAN tool which allow authoring of adaptive e-learning courses. A recent assessment of their usability [9] shows encouraging results for the application of visual environments to adaptive e-learning course authoring. However, this system only updates the User Model based on a student's performance in tests following lessons. LAG, on the other hand, allows direct manipulation of the User Model and Presentation Model every time the user accesses a concept. This allows the adaptation of the next concept visited by the user based on the last concept they visited within the current lesson. While adaptive courses generated using the SEAMAN tool could potentially be converted to the common platform provided by MOT, we would like to maintain the expressivity of LAG in our visual environment. The SEAMAN tool is therefore not suitable for our requirements.

4 The PEAL 2 Visual Programming Environment

We followed a strategy of developing a basic working visual environment while integrating semantics and visual feedback of the programming domain of adaptation strategies in parallel.

4.1 The Visual Environment

Visual elements have been implemented to represent each LAG language construct. Where a construct can contain other constructs or code such as a list of statements or operands to an operator, a visual element container holds child elements which make up the complete element. Strategies can be created using the visual environment by inserting elements into appropriate containers. Visual elements can be dragged to other containers by clicking and dragging the element with a mouse.

To give an example of a visual element, Fig. 1 shows a screenshot of the *Each-Concept Condition-Action* visual element implemented in PEAL 2. This visual element represents the LAG *while* construct which can be seen with an additional comparison and statement in Listing 1. The effect of the *while* construct is to execute each contained statement with respect to each *Concept* which is part of the Goal Model of the current adaptive course for which the condition is satisfied. This visual element represents that behaviour as a flowchart-like loop from a *multiple-documents* symbol labeled “For each concept in the lesson”, to a *condition rhombus*, to a *process box* and back to the *multiple-documents* symbol.

The context menu[3] opened over the *process box* in Fig. 1 allows the insertion of new visual elements such as *Assignment* statements like `PM.GM.Concept.show = true` as shown in Listing 1. The insertion option is only available for visual element containers and depends on the context – the context menu for the *condition rhombus* condition container only allows insertion of condition visual elements which could, for example, represent the comparison in Listing 1.

```
while ( GM.Concept.weight > 5 ) (
    PM.GM.Concept.show = true
)
```

Listing 1. LAG *while* construct with a *greater-than* comparison and one assignment statement

The *Help* option, as shown in Fig. 1, displays the documentation section specifically relevant to the context. For example, in the above context, the section

[3] displayed upon right-click in Microsoft Windows operating systems

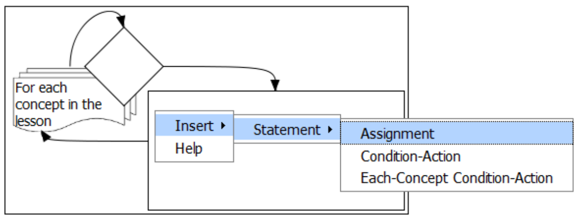
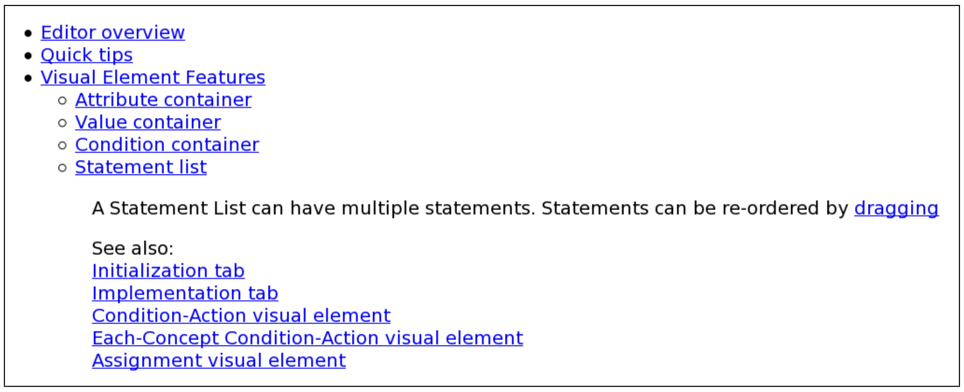


Fig. 1. PEAL 2 *Each-Concept Condition-Action* visual element with the *Insert* part of its context menu open

on *Statement Lists* shown in Fig. 2 would be displayed, since the context is a container for a list of *Statement* visual elements. A single user documentation web page was created which can be read as a whole like traditional documentation. Each topic was, however, given a distinct name using the HTML *id* attribute like *statement-list-documentation-section* in the above example. JavaScript embedded in the documentation then collapses each section to show only its title, except for the chosen section which is displayed fully. Following cross-referencing links in the documentation expands those sections. This shows the user only the relevant information to avoid information overload but gives them the freedom to explore more topics easily.



- Editor overview
- Quick tips
- Visual Element Features
  - Attribute container
  - Value container
  - Condition container
  - Statement list

    A Statement List can have multiple statements. Statements can be re-ordered by dragging

    See also:
    Initialization tab
    Implementation tab
    Condition-Action visual element
    Each-Concept Condition-Action visual element
    Assignment visual element

Fig. 2. Screenshot of contextual documentation opened using the context menu in Fig. 1

We have also made use of *tooltips* which are very brief descriptions that can be used to identify visual elements and their parts when the mouse cursor is held over them.

Visual feedback is provided to indicate where a dragged element can validly be inserted. The containers where a visual element may be dropped are coloured green while the element is held, as can be seen in Fig. 3 b and 3 c. An assignment can be dropped in the *then* and *else* parts but not in the *condition rhombus*. The container resizes to accommodate the dragged element when it is entered by the mouse cursor (Fig. 3 c). As users drag elements around the programming environment, they can thus quickly see and learn where different kinds of elements can be placed.

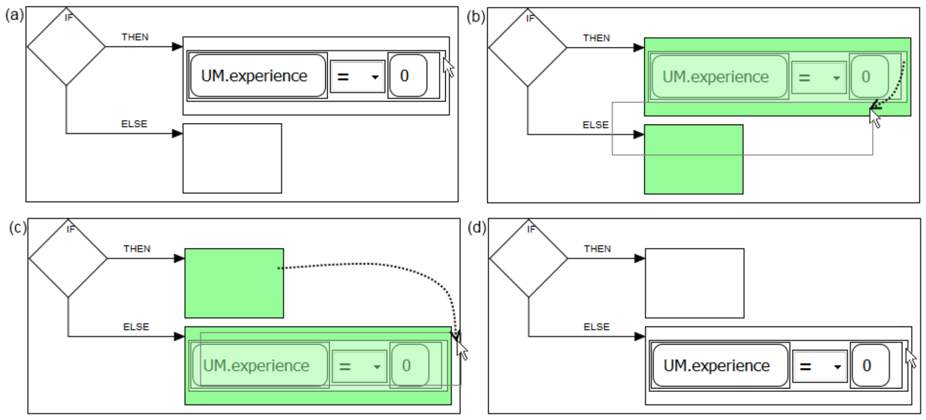


Fig. 3. Sequence of a PEAL 2 *Assignment* element in the *then* part of a *Condition-Action* visual element **(a)** which is dragged **(b-c)** to the *else* part where it is released **(d)**

The visual environment is placed on a *tab* on the PEAL 2 web application. This hides the PEAL text environment until the user changes to the text editor tab, at which point the visual representation is converted to LAG to update the text environment. Changing from the text tab to the visual tab converts the LAG language text to the visual representation to propagate changes made to the LAG code.

The strategy description is placed on a third tab. To encourage the author to write a thorough description, this tab is coloured red initially. The words are counted as the user types and the tab is changed to green once the strategy description consists of 20 words or more.

4.2 Conversion from the Visual Representation to LAG

The Object-Oriented implementation of visual elements proved to make converting the visual representation to LAG very easy. A *toLAG()* method exists for each visual element object. This returns the LAG representation of that element which wraps the LAG of its decendent elements by calling *toLAG()* on each child in its visual element containers.

4.3 Conversion from LAG to the Visual Representation

PEAL already featured a LAG parser implemented in JavaScript based on the CodeMirror framework [2]. This is used to perform syntax colouring and syntax error checking. A parser is needed in PEAL 2 to convert LAG to the visual representation. Although parser generators exist that can output JavaScript implementations, we decided not to duplicate effort by implementing and maintaining two parsers but instead to use that of PEAL for all purposes in PEAL 2.

PEAL's parser is similar to recursive descent parsers [5] in that a function exists for each nonterminal. However, instead of these functions directly making recursive calls to expand nonterminals, the functions, referred to as *actions*, representing the expected non-terminals are placed onto a stack and the current token is consumed. These *action* functions check that expected tokens are found, set token styles for syntax colouring, report syntax errors and consume tokens as appropriate.

We added more actions which create visual elements as the relevant LAG tokens are consumed. The JavaScript code in Listing 2 shows part of the function which parses LAG statements – the part for the *if-then-else* construct in particular[4]. The *ToVisual.actions.startIf* action was added to create the *Condition-Action* visual element, seen in Fig. 3, which is the PEAL 2 visual representation of the common *if-then-else* construct. To insert nested visual elements in the correct block context, a stack of visual elements represents the current position in the visual element hierarchy.

```
// Dispatches various types of statements
//   based on the type of the current token.
function statement(type, tokenValue) {
    if (type == "if") {
        cont(
            ToVisual.actions.startIf,
            condition,
            then,
            ToVisual.actions.finishIf
        );
    } else if ( type == "while" ) { // etc.
```

Listing 2. Part of JavaScript function which parses the LAG *statement* non-terminal

The correct syntax is reliably converted to the visual representation, although incorrect syntax is not yet handled in conversion but is only marked red for the user to correct by hand with the help of syntax error messages as in the original PEAL tool.

[4] Actions maintaining lexical scope for indentation in the PEAL editor have been removed here for simplicity

4.4 JavaScript Implementation

Each HTML element exists in the browser as a Document Object Model (DOM) [3] node. The Yahoo User Interface library (YUI) [17] provides a JavaScript object which wraps a DOM node, allowing browser-independent interaction with the DOM.

Visual elements are created entirely using JavaScript to create HTML and SVG[4] elements and are styled using Cascading Style Sheets (CSS). This means additional elements can easily be specified by adding references to extension JavaScript and CSS files to the PEAL 2 environment in the future.

5 Evaluation

We assessed whether PEAL 2 makes it easier for non-programmers to author adaptation strategies using the following evaluation:

Based on the evaluation of the PEAL tool [13], we developed a set of tasks followed by a set of questions.

Tasks

1. Author a strategy from scratch using the PEAL 2 visual environment.
2. Author a strategy from scratch using the PEAL 2 text environment.
3. Extend an existing strategy using both the visual and text editors.

Questions

1. What are the main strengths of the visual and text LAG editors?
2. What are the main weaknesses of the visual and text LAG editors?
3. Would you prefer to use the PEAL 2 visual environment, the text environment or some other editor to author strategies in LAG?
4. Other comments?

This evaluation was performed with three subjects with no programming experience and one subject with only experience of writing HTML markup. All subjects were given a short presentation explaining the LAOS framework and how adaptation strategies can be implemented in LAG. The LAG grammar and semantic definitions were provided but only a verbal explanation of the visual elements was given as the user documentation had not been prepared at the time of the evaluation.

Subjects made the following comments after using the text editor and visual environment:

1. Two subjects found the visual representation easier to understand as an adaptation process while finding the text version very abstract.
2. Two subjects found the text editor alongside the grammar and semantics more intuitive and noted that this format is more familiar to them in their degree language studies.

3. One subject recommended using colour to help distinguish different kinds of visual elements like the syntax highlighting of the text editor.
4. The meaning of some of the components of the visual representation, e.g. the flowchart condition-rhombus, had to be explained to users not familiar with them.
5. To several subjects, the effect of a strategy was evident very quickly in the visual representation once the meaning of components was clear.

Two out of the four subjects preferred the visual environment implemented in PEAL 2 over the existing PEAL text editor. When a strategy was shown in the visual environment, those subjects immediately expressed how much easier the visual representation was to follow. It was clear that a tutorial-style explanation based on examples was desired by all subjects, as well as documentation of the visual elements. For this reason, we placed a higher priority on providing user documentation which was implemented after this evaluation. The two subjects who preferred the text editor were more comfortable referring to the LAG language grammar and semantics specifications than diagrammatic representations of a strategy. This shows the value of offering the option of using the text editor even to non-programmers as done in PEAL 2.

6 Conclusion

In this paper we have documented the development of a visual environment for authoring adaptation from the design, to implementation, to the evaluation. Section 4 described how our design was implemented using the latest web technologies stable and compatible enough for wide use without dependence on third-party plugins. The structured implementation prepares the tool for further extensions. The evaluation documented in Sect. 5 shows that the visual environment described here lowers the threshold for authoring adaptation strategies for visually oriented non-programmers. By offering the visual environment alongside the existing PEAL text environment, text-oriented authors are still catered for and authors can choose the most appropriate environment as they gain experience. Furthermore, we have shown how some key areas of improvement discovered through the evaluation have been implemented to further improve the state for the lay person author.

7 Further Work

In addition to improving the current state of authoring tools, PEAL 2 can be built on in the following ways:

- Extend PEAL's code fragment storage and reuse to the visual environment
- Improve compatibility with more common web browsers
- Improve handling of bad syntax during conversion to the visual representation

– Improve syntax checking to list multiple errors
– Framework for writing and importing documentation translations
– Compare attribute references in CAF content and LAG strategies to give author an indication of whether they are compatible
– Provide direct output to adaptation delivery engines to allow immediate preview and testing of new adaptive hypermedia systems
– Use colour to provide further visual feedback and identification cues

References

1. Adobe flash platform, http://www.adobe.com/flashplatform/
2. CodeMirror: In-browser code editing made almost bearable, http://marijn.haverbeke.nl/codemirror/
3. Document Object Model (DOM) Level 3 Core Specification, http://www.w3.org/TR/DOM-Level-3-Core/
4. Scalable Vector Graphics (SVG), http://www.w3.org/Graphics/SVG/
5. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: principles, techniques, and tools. Addison-Wesley, Boston, MA, USA, 2nd edn. (2007)
6. Backlund, B., Hagsand, O., Pehrson, B.: Generation of visual language-oriented design environments. Journal of Visual Languages & Computing 1(4), 333–354 (1990)
7. Brusilovsky, P.: Adaptive hypermedia. User Modeling and User-Adapted Interaction 11(1-2), 87–110 (2001)
8. Casella, G., Costagliola, G., Ferrucci, F., Polese, G., Scanniello, G.: Visual languages for defining adaptive and collaborative e-learning activities. In: Isaas, P., Kommers, P., McPherson, M. (eds.) Proceedings of IADIS International Conference e-Society 2004. vol. 1, pp. 243–250. IADIS Press, Ávila, Spain (2004)
9. Costagliola, G., De Lucia, A., Ferrucci, F., Gravino, C., Scanniello, G.: Assessing the usability of a visual tool for the definition of e-learning processes. Journal of Visual Languages & Computing 19(6), 721–737 (2008)
10. Costagliola, G., Deufemia, V., Polese, G., Risi, M.: Building syntax-aware editors for visual languages. Journal of Visual Languages & Computing 16(6), 508–540 (2005), selected papers from Visual Languages and Formal Methods 2004 (VLFM '04)
11. Cristea, A.I., Calvi, L.: The three layers of adaptation granularity. In: Brusilovsky, P., Corbett, A.T., de Rosis, F. (eds.) User Modeling. pp. 4–14. Springer, Johnstown, PA, USA (2003)
12. Cristea, A.I., de Mooij, A.: LAOS: Layered WWW AHS Authoring Model with Algebraic Operators. In: WWW (Alternate Paper Tracks) (2003)
13. Cristea, A.I., Smits, D., Bevan, J., Hendrix, M.: LAG 2.0: Refining a Reusable Adaptation Language and Improving on Its Authoring. In: EC-TEL. pp. 7–21 (2009)
14. Cristea, A.I., Smits, D., De Bra, P.: Towards a generic adaptive hypermedia platform: a conversion case study. Journal of Digital Information 8(3) (2007)
15. Hendrix, M., De Bra, P., Pechenizkiy, M., Smits, D., Cristea, A.I.: Defining Adaptation in a Generic Multi Layer Model: CAM: The GRAPPLE Conceptual Adaptation Model. In: Dillenbourg, P., Specht, M. (eds.) EC-TEL. LNCS, vol. 5192, pp. 132–143. Springer (2008)

16. Nickerson, J.V.: Visual programming. Ph.D. thesis, New York University, New York, NY, USA (1995)
17. YUI 3 – Yahoo! User Interface Library, http://developer.yahoo.com/yui/3/