Language Design for a Personal Learning Environment Design Language

Felix Mödritscher, Fridolin Wild, Steinn Sigurdarson

Institute for Information Systems and New Media,
Vienna University of Economics and Business Administration,
Augasse 2-6, 1090 Vienna, Austria
{felix.moedritscher,fridolin.wild,steinn.sigurdarson}@wu-wien.ac.at

Abstract: Approaching technology-enhanced learning from the perspective of a learner, we foster the idea of learning environment design, learner interactions, and tool interoperability. In this paper, we shortly summarize the motivation for our personal learning environment approach and describe the development of a domain-specific language for this purpose as well as its realization in practice. Consequently, we examine our learning environment design language according to its lexis and syntax, the semantics behind it, and pragmatical aspects within a first prototypic implementation. Finally, we discuss strengths, problematic aspects, and open issues of our approach.

Keywords: Personal Learning Environments, Language Design, Learner Interactions, End-User Development

1 Introduction

New economic-technological trends, subsumed under the term Web 2.0, have started to influence the understanding for and approach to technology-enhanced learning. Additionally current research and development is focus on extending the classical didactic learning management systems by emphasizing functionality supporting the learner, e.g. by dealing with the design of personal learning environments, coping with social networking functions and recommendation services, or examining collaboration and community-boosting features in learning networks. In sum, networked communities are expected to be beneficial for lifelong learning [1].

Also heading into this direction, we summarized shortcomings of instructional design theories and adaptive educational technologies in [2] and came up with our approach; **Mash-UP Personal Learning Environments (MUPPLE)**. Hereby, we build upon three important concepts of learning environment design:

– First, we prefer the idea of '*learning to learn*' in connection with learning content, to 'transferring' domain-specific knowledge. Thus we a apply a simple and domain-independent model of learning activities which consists of a set of (learner) actions bound to objects (artifacts or outcomes) and tools. Furthermore, we emphasize the *acquisition of transcompetences* (i.e. social, self, and methodological competence) in addition to content competence.

- Second and consequently, we consider the *learning environment* an important part of the *learning outcome* as opposed to an instructional condition. Therefore, a learner *designs her learning environment* by establishing a network of people, artifacts, and tools (manually or with the support of personalization services) and interacting with that environment.
- Third and finally, we consider emergence of behavior as an unavoidable and natural phenomena of complex socio-technical systems. By emergent behavior we mean that the observable dynamics show unanticipated activity, surprising in so far as the participating systems have not been designed for it specifically (they may even not have intended it). *Designing for emergence* is, in our view, more powerful than rule-based personalization, as the models involved are simpler while achieving the same effect.

As a consequence of the application of these three concepts, we decided to follow a learner-centered approach to personal learning environments. In accordance with the paradigm of end-user development depicted in [3], we have been developing a domain-specific language for learning environment design which we named **learner interaction scripting language (LISL)**. In the following section we briefly summarize the principles of language design. In section 3 we describe LISL, a first prototypic implementation, and it's utility, before our approach is discussed.

2 Principles for Language Design

In order to achieve end-user developed emergence of learning behavior (MUPPLE), the scripting language (LISL) needs to take into account not only traditional software engineering issues, but also those of end-user development. We consider LISL to be the underlying model for the success of the MUPPLE approach.

Hoare meant a programming language to support three tasks within the software development process [4]: (1) program design, (2) programming documentation, and (3) program debugging reasons. Therefore, he made up five language design principles to achieve these task-specific requirements: First, *simplicity* of a language is critical for reducing the complexity of programming tasks. Second, *security* features should not be removed when going into production, so that programming errors are not hidden away from the end-user for reliability reasons. Third, *fast translation* of program code is of particular interest for prototyping or debugging, which already reminds of younger trends like scripting of command-line interpreters. Fourth, *efficient object code* is required, particularly if following the design for emergence paradigm for end-users developing and optimizing their learning environments. Fifth, compiling and processing programs should increase *readability*, e.g. by automated intention, understandable commands, filling in defaults, etc.

Additionally to these historical principles, younger trends focus on derivations of programming languages. For instance, Paige and his colleagues address modeling language design and divide them into four task-related classes [5]: (1) architectural description, (2) behavioral description, (3) system documentation, and (4) forward and backward generation. For these kinds of modeling languages the following design principles are recommended: First, a language should be *simple* to have it small and

memorable. Second, it should satisfy the principle of *uniqueness* (orthogonality) which means that every concept of interest can be expressed in exactly one way and, thus, the language is defined by a small number of powerful features. Third, the principle of *consistency* refers to a purpose to the design of the language, so that the features included or to be added to the language must further this purpose. Fourth, *seamlessness* contributes to producing maintainable software, i.e. through generating code from models by allowing the mapping of abstractions in the problem space to implementations in the solution space without changing the notation.

Fifth, the principle of *reversibility* requires that changes made during one stage of the development lifecycle can be automatically reflected back to earlier stages. Sixth, *scalability* ensures that a language should be useful to model systems with a few components and inter-relations, and systems with thousands of components and inter-relations, which is achievable by providing a concise mechanism for describing the fundamental abstractions for their problem domain. Seventh, *supportability* demands that a language can be easily produced by hand, so that it can be used, by humans, for writing or drawing models. Eight, a modeling language must support the production of *reliable* programs which meets its specification and reacts appropriately whenever unexpected or erroneous input is given. Ninth, the principle of *space economy* states that models should take up as little space on a printed page as possible but without scarifying the understandability of the language, which empowers the production of concise models and decreases the maintenance efforts.

Concerning language design methodologies, Van Deursen et al report about a design approach to domain-specific languages. Basically, a domain-specific language is defined as "*a programming language or executable specification that builds on appropriate notations and abstractions, offering expressive power focused on, and usually restricted to, a particular domain*" [6]. Thus, such a language is expected to be based in an application domain. They suggest building an algebraic model behind the language, consisting of normalization, variability, and satisfaction rules. This algebraic model should be derived from an analysis of the domain by applying feature descriptions, i.e. with feature diagrams or a textual notation.

All in all, this short inspection of literature already indicates that the design of a new language is a complex task. Considering the basic principles noted by Hoare nearly four decades ago is, however, a solid first step to designing a usable programming language. Moreover, design guidelines for modeling languages are of interest for our learning environment design approach. In addition, we consider our scripting approach to be highly related to a domain-specific language and, thus, a methodological approach like the one by Van Deursen et al is relevant. Finally, human-computer interaction research started to influence software engineering, which lead to the concept of end-user development and evolves software engineering from making systems that are "*easy to use*" to making systems that are "*easy to develop*" [3]. Again, this new way of user-driven development of the working environment which is used for interactions in shared spaces and networked communities requires special design guidelines, which amongst others, are elaborated by [7]. We consider end-user development as well as these guidelines to be of importance to our MUPPLE approach and explain them in the upcoming section when describing the development process of our learner interaction scripting language (LISL).

3 LISL and its Interpreter

With respect to the principles outlined in the last section, we decided to build up a human-like language for learning environment design and, additionally, focus on user-related issues like learnability, efficiency, simplicity, readability, uniqueness, seamlessness, reversibility, and supportability. More technical issues, e.g. security, fast translation or efficient object code, are considered to be secondary. In this section, our learner interaction scripting language (LISL) is described in detail.

The **lexical and syntactical structure** is kept very simple. The tokenizer splits each line of code into tokens separated by white spaces. Single or double-quoted constructs are considered to be one token. Moreover, tokens are case-sensitive, and identifiers are restricted to alphanumeric strings (beginning with a letter from the alphabet) and might be quoted. Syntactically we realized only a few constructs with this domain-specific language.

```
1>  define action Compose with url http://[...]?action=create
2>  define action Browse
3>  define action Bookmark
4>  define object ‘self-description’
5>  define object ‘self-descriptions of peers’ with url http://[…]/peers
6>  define object ‘selected self-descriptions’
7>  define tool VideoWiki with url http://videowiki.icamp.eu
8>  define tool Scuttle with url http://scuttle.icamp.eu
9>  connect tool VideoWiki with tool Scuttle
10> Compose ‘self-description’ using VideoWiki
11> Browse ‘self-descriptions of peers’ using VideoWiki
12> Bookmark ‘selected self-descriptions’ using VideoWiki
13> drag tool VideoWiki
```

Fig. 1. Example LISL script consisting of three actions, three objects, two tools, as well as one connect, three action and one learner interaction statements

As indicated with the example activity in Fig. 1, we differentiate between the following statement types:

- ‘Define’ statements (cf. lines 1 to 8) are used to initialize the mash-up personal learning environment, i.e. to declare actions, objects, and tools available in one mashup page. The actor is considered to be the current user. Each defined entity stands for a variable in the ‘program’, whereby an action and an object can have an optional URL, while the tool always has one. These kinds of constructs always start with the command ‘define’ followed by the type of entity (‘action’, ‘object’, ‘tool’) and a unique identifier. If a URL can be specified, the keyword ‘url’ followed by the URL itself is expected. Additional tokens are ignored, so that constructs like ‘with the url http://[...]’ are valid.
- ‘Connect’ statements (line 9) are necessary for interoperability reasons, so that tool combinations can be utilized for one action. Syntactically, such a statement allows combining two tools with each other, so that data is sent from one to the other. What this exactly means and how we intend to realize this kind of interoperability, will be clarified when explaining the semantics of this statement.

– 'Action' statements (lines 10 to 12) comprise the actions to be performed by a learner and consist of three parts: (a) the action, (b) the object, and (c) the tool. Generally, the first token, plain or quoted, stands for the (user-definable) action, while the second one represents the object. The tool has to be specified right after the keyword 'using', other tokens are ignored. If one of the elements of such a statement can not be resolved, an error with an explanation is displayed.
– 'Learner interaction' statements (line 13) materialize the user interactions with the learning environment, i.e. they describe if the learner navigates between two different learning tools or she rearranges the application on her screen.

Referring to the end-user development guidelines (DG) highlighted in [7], we consider the three syntactical ones as fulfilled for LISL: (DG 3.1) Syntactical errors are hard to be made due to adequate error messages and visualization techniques by the LISL interpreter (shown later). (DG 3.2) It is even impossible to make syntactical errors, as incorrect statements are not executed and, thus do not halt the execution of the activity. (DG 3.3) We tried to use objects as language elements in the way that one action statement comprises a tool within the learning activity. Such a tool can be drag and dropped as an 'object' (window) within a web application mashup [8].

Going on with the **semantics behind LISL**, we build up a simple, comprehensible model of learning activities, as shown in Fig. 2. Precisely, a learning activity consists of a set of (learner) actions which are bound to one object and involve at least one tool. These actions specify typical learner interactions within the activity, whereby they can be defined by learners. The ones with a specific URL are useful to perform a standard operation within a learning tool, e.g. creating a new Wiki page. An action without an URL delegates the initialization of the web application to the object or, if it does not have an URL, to the URL of the first tool after the 'using' keyword.

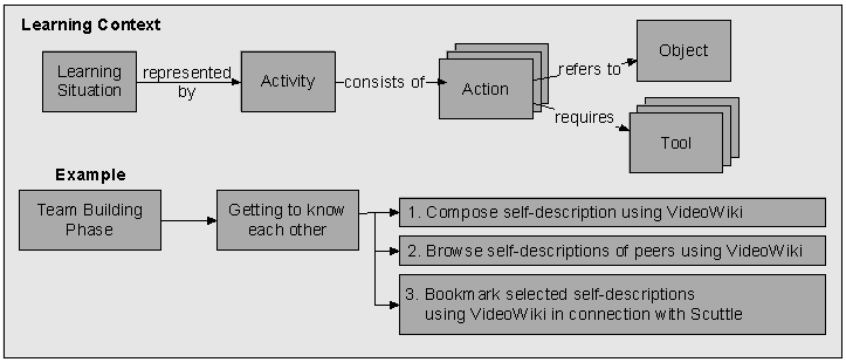

Fig. 2. LISL's semantic model of learning activities consisting of actions-object-tool triples

In analogy to variable declarations, an object can, but must not be defined with a specific URL. An initialized object means that the outcome has a pre-defined pointer to a specific artifact within a tool. An object without a URL, however, is indicating that the 'value' is assigned dynamically as a learner goes through the actions, e.g. by creating a digital artifact using an action with a URL. In our example described in the last two figures, for instance, the very first action creates a VideoWiki recording

which is assigned to the 'self-description' object after finalizing the action. Binding a tool combination to one action requires a certain degree of interoperability of the learning tools. As a precondition for our MUPPLE approach, we build upon distributed feed networks and an API we have been developing within the iCamp research project [9]. If this kind of interoperability is not supported, an error is given.

We view the semantic guidelines for end-user development [7], as an affirmation of our scripting language approach. First, LISL is a domain-specific language for end-user development, comprised of human-like, understandable constructs like the action statements (DG 4.1). Second, LISL also includes meta-domain orientation for general end-user development, i.e. by having neutral, domain-independent statements, like 'define' or 'drag', wherever abstraction from the TEL domain is possible (DG 4.2). Finally, the LISL runtime, MUPPLE supports semantic annotations (DG 4.3) in two different ways. On the one hand, a graphical user interface encapsulates the scripting activity itself, so that learners can use web-based widgets to 'program' their personal learning environment. On the other hand, semantic annotations can be realized by recommending action-object-tool bindings of peers.

From the point of view of **pragmatics**, we implemented an interpreter as well as a web-based interface for LISL. Technologically, this prototype is realized as part of the OpenACS framework, an open source toolkit for building scalable, community-oriented web applications (cf. http://openacs.org). The LISL interpreter is written in the programming language Tcl, precisely the object-oriented extension named XoTcl (http://media.wu-wien.ac.at), and is part of the Mupple package which itself is based on the XoWiki module (http://openacs.org/xowiki). The Mupple package, including the source files, can be retrieved from the iCamp code repository at Sourceforge (http://sourceforge.net/projects/icamp).

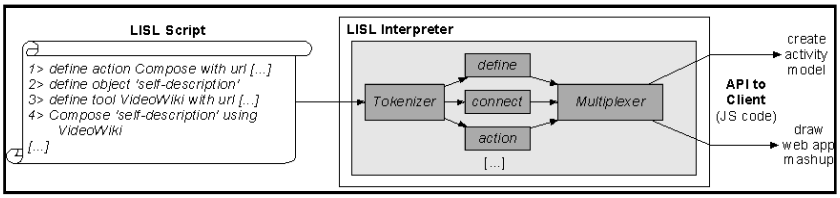

Fig. 3. Functional diagram of the LISL interpreter including a tokenizer, interpreter units for each statement type, and a multiplexer controlling different APIs

Fig. 3 visualizes how the LISL interpreter works beneath the surface. Going through the script line by line, the input is tokenized and, according to the first token, processed by the adequate interpreter unit. Each of these units is representing one specific statement type and produces output for different purposes. Our current implementation differentiates between two output channels, one for creating the learning activity model on the client-side, the other one for creating and updating the mashup of the web-based learning tool. Hereby, the interpreter calculates the latest version of the model and the mashup space, before this information is transferred via a JS-based API to the browser. Updates are made incrementally, so that learners (facilitators and peers!) can execute single lines of code at any time.

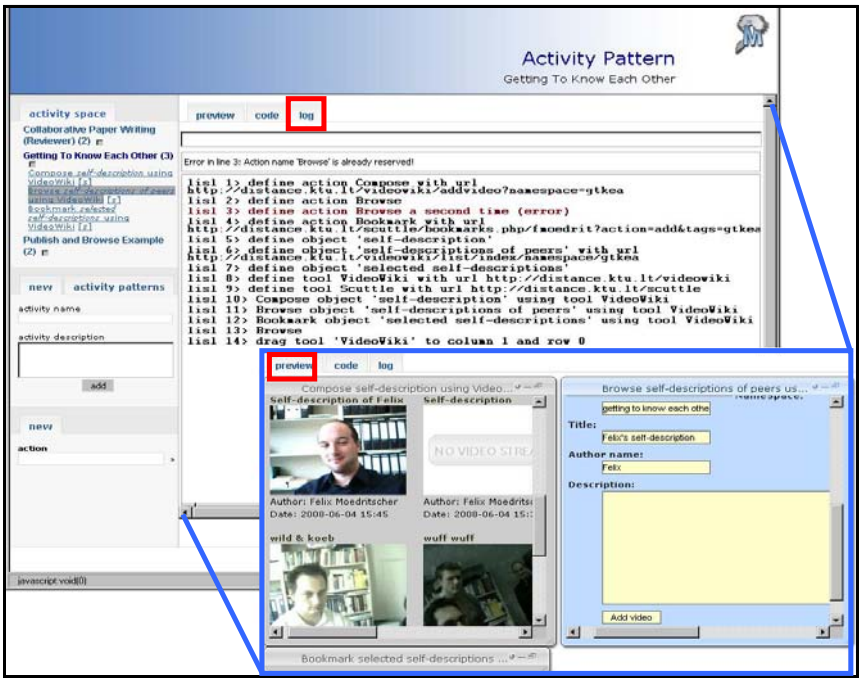

Fig. 4. MUPPLE page for the activity 'Getting To Know Each Other'; the tab 'log' shows the LISL script of our example executed by the interpreter (red indicates an error), while the tab 'preview' (blue border) visualizes the web application mashup and provides web-based widgets

As shown in Fig. 4, our MUPPLE prototype displays a learning activity in the form of a page which is the central user interface of the personal learning environment. At the top, the type (activity or activity pattern) and the title of the page are presented. On the left, all activities of a user are listed. Clicking on an activity folds out the list of its actions and loads it into the content area. Below there are various functions for creating new activities (blank or from a pre-defined pattern), deriving new patterns from the current activity, or starting a new action. For starting a new action MUPPLE supports the learner by recommending action-object-tool triples.

Within the tab 'log', the content area displays the materialization of the learner interactions; the LISL code of the current activity page. By switching the view to the tab 'preview' (the area with blue boarder), learners see the web application mashup space generated from the executing LISL code. The code itself is created either by manual scripting or by logging the interactions with the web-based widgets and stored within the MUPPLE page. On executing the script, the semantic model is built up, and the web applications are launched in windows on the mashup space. Now, learners can work with the web-based control widgets, whereby their interactions are materialized by appending new LISL statements to the MUPPLE page. On returning to a page, its last state will be restored, which we consider as important for scrutable behavior of MUPPLE. Furthermore, any part of the learning activity is subject to

adaptation by the user (facilitators and peers) for controllability reasons. Technically, LISL is similar to AppleScript (http://www.apple.com/applescript), but it does not limit the user to automate interactions with an application. Instead, it enables learners to reflect their learning process and to collaborate in learning networks.

Again, we analyze LISL according to the guidelines for the pragmatics of an end-user development language [7]. First, the scripting approach and the web-based command-line interpreter seem to be ideal for supporting incremental development of a personal learning environment (DG 5.1) by being able to see and modify the LISL script of a page (tabs 'log' and 'code'). While our approach does not directly facilitate decomposable test units (DG 5.2), the interactive design allows learners to instantly see the systemic reactions to new or modified code, thus simplifying testing. As the underlying storage layer also supports version management, learners can even visualize differences between versions of a MUPPLE page and choose to go back to a previous version. Third, our prototype provides multiple views with incremental disclosure (DG 5.3) for a MUPPLE page, e.g. by switching between the different tabs. Fourth, our approach integrates the 'development tool' with web services (DG 5.4), which is, according to our definition, the primary goal of a mash-up personal learning environment. Fifth, syntonicity is encouraged (DG 5.5) particularly through the web-based widgets (tab 'preview') for user-driven development of the learning environment. Sixth, the web-based LISL interpreter allows immersion (DG 5.6) with the LISL editor at the tab 'code' and the command-line interpreter at the tab 'log'. In both cases learners can play around with the LISL source code and see the effects of their modifications, this also owes to our preference for 'designing for emergence'. Seventh, LISL supports scaffolding typical design (DG 5.7) in the way that facilitators and peers can share activity patterns which other learners can use and adapt. For that purpose we realized the before-mentioned recommendation service as a support for learners. Finally, the whole MUPPLE approach aims at community building and tools to achieve community-based activities (DG 5.8). Overall, we think that LISL and the idea of mash-up personal learning environments perfectly ties in to end-user development and that we considered relevant guidelines of this field.

4 Conclusions and Discussions

To sum up this paper, we believe that LISL and the way we designed it is a key step in realizing the three concepts of learning environment design outlined in section 1: (1) The semantic model of learning activities is the key to the 'learning to learn' paradigm, as we primarily focus on the learner 'actions' – domain-specific aspects, however, might be subject to the 'objects' (e.g. artifacts) available in the activities. (2) The end-user development approach itself clearly manifests that we consider the learning environment to be the outcome of and not another pre-requisite to learning. Precisely, we prefer the idea of learners developing their learning environment by themselves to instructional design or adaptive strategies. (3) Designing for emergence is important to get learners to use this way of 'learning' and to experience the systemic behavior as a natural part of their daily processes. With respect to the

principles for language design highlighted in this paper, we think that LISL considers them quite well, although we have not conducted evaluation studies yet.

All in all, learning environment design and LISL might be a useful answer to what Koper calls the 'second road education' [1] by means of lifelong learning strategies beyond primary, secondary, and tertiary education. Particularly, the MUPPLE approach is promising for building and sustaining learning networks in which actors can collaborate within shared activities and on shared objects and in which they can share best practices with peers, e.g. through LISL-based activity patterns. However, we are aware of the fact that MUPPLE is at an early development stage and, for instance, aspects of networked collaboration are still an open issue. So far, LISL aims at the learner-centered perspective, namely the learning environment design and the interaction with the tools, but lacks typical workflow issues which occur if a learner group collaborates on shared objects (artifacts) and uses the same tools. Furthermore, MUPPLE requires regulation facilities and the considerations of social network issues, like privacy, to ensure a valid learning community approach. Another weakness comprises tool interoperability, where we built upon distributed feed networks. Although we realized the FeedBack API for a few systems (Moodle, Wordpress, Scuttle), a MUPPLE-compliant learning tool must implement this API or provide some kind of generic Web 2.0 API to support interoperability according to our needs. Finally, MUPPLE lacks of evaluation results. For the design of LISL it would be necessary to show that the principles for language design, especially the user-centered ones like simplicity, learnability, efficiency, etc., are taken into account.

References

1. Koper, R.: Supporting the Continuing and Lifelong Development of Individuals in Online Learning Networks. Invited Talk at the ED-Media Conference (2008)
2. Wild, F., Mödritscher, F., Sigurdarson, S.E.: Designing for Change: Mash-Up Personal Learning Environments. In: eLearning Papers, 9 (2008)
3. Lieberman, H., Paterno, F., Klann, M., Wulf, V.: End-User Development: An Emerging Paradigm. In Lieberman, H., Paterno, F., Wulf, V. (eds.): End-User Development, LNCS, vol. 4321, pp. 1-8. Springer, Dordrecht (2006)
4. Hoare, C.A.R.: Hints on programming language design. Report, Stanford University (1973)
5. Paige, R.F., Ostroff, J.S., Brooke, P.J.: Principles for modeling language design. In: Information and Software Technology, 42, pp. 665-675 (2000)
6. Van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: an annotated bibliography. In: ACM SIGPLAN Notices, 35(6), pp. 26-36 (2000)
7. Repenning, A., Ioannidou, A.: What Makes End-User Development Tick? 13 Design Guidelines. In Lieberman, H., Paterno, F., Wulf, V. (eds.): End-User Development, LNCS, vol. 4321, pp. 51-86. Springer, Dordrecht (2006)
8. Mödritscher, F., Neumann, G., García-Barrios, V.M., Wild, F.: A Web Application Mashup Approach for eLearning. In: Proc. of the OpenACS/.LRN Conference, pp. 105-110 (2008)
9. Wild, F., Sigurdarson, S.E.: Distributed Feed Networks for Learning. In: The European Journal for the Information Professional (UPGRADE), 9(3) (2008)