Designing a Learning Design Engine as a Collection of Finite State Machines

Implementing a learning design engine as a collection of finite state machines

Hubert Vogten, Colin Tattersall, Rob Koper, Peter van Rosmalen,

Francis Brouns, Jan van Bruggen, Peter Sloep, Harrie Martens

All authors are employed by the Educational Technology Expertise Centre of the Open University of the Netherlands.

Address for correspondence:

Hubert Vogten

Open University of the Netherlands

PO Box 2960

NL-6401DL, Heerlen

The Netherlands

hubert.vogten@ou.nl

tel: +31 45 5762126

fax: +31 45 5762800

Abstract

Specifications and standards for e-learning are becoming increasingly sophisticated and complex as they deal with the core of the learning process. With the arrival of the latest specifications and standards for e-learning, the sophistication, expressiveness and complexity has increased considerably. Simple transformations are not adequate to successfully implement these specifications and standards. IMS Learning Design (LD) (IMS, 2003b) is a representative of such a new specification. Its declarative nature and expressiveness increases the complexity for any implementation. This probably is the largest hurdle that stands in way of successful general deployment of this type of specifications.

This article describes the implementation of an engine for LD as collection of finite state machines (FSMs). Each state is constructed from a set of properties which can either declared explicitly in LD or implicitly by the engine. State transitions are implemented through a mechanism of events and event handlers, completing the finite state machine. By re-using certain properties across FSMs it is possible to create automatic propagation mechanism dealing with group dynamics without additional efforts. With the FSMs in place, personalization becomes a simple task. By combining the principles presented in the article, it becomes clear that an elegant implementation becomes feasible, and has been demonstrated in the first actual implementation.

Keywords

Learning Design, Web-based educational system, e-learning, Finite state machine, Personalization, Implementation

Introduction

As open specifications (and standards) in e-learning are becoming more mature, their richness and complexity increases (IEEE Learning Technology Standards Committee2003), (IMS, 2003a). Specifications that were solely dealing with meta-data have long since moved on, and as a result implementations for these specifications are not as straightforward anymore as they were a few years ago. Good examples of such emerging new specifications, dealing with the pedagogical frameworks, are IMS Simple Sequencing and LD. There is a need for additional guidelines in addition to the specification itself, to help developers incorporate these specifications in their e-learning systems. This article will provide guidelines for implementers wanting to incorporate the LD specification into their products.

LD specifies under which conditions, activities have to be performed by learners and teachers to enable learners to attain the specified learning objectives.
LD is used to specify the learning design of e-learning courses (so-called 'units of learning'). A unit of learning is a package that consists of meta-data about the course, the learning design of the course and references to physical resources and/or the physical resources themselves (learning objects and learning services) that are used in the course. The LD specification supports the use of a wide range of pedagogies by providing a generic and flexible language, and is based on a pedagogical meta-model(Koper & Manderveld, 2004), (Koper & Olivier, 2003). It expresses the pedagogical approach, supports personalization of learning routes and supports

reusability. The learning design specification is designed to allow for repetitive use in different situations with different persons and contexts.

LD starts from the principle that a person is assigned to one or more learner or staff roles. So all references to users, be it learners of staff, are made via these roles and never on an individual basis. In a role a person has to perform learning activities to attain the specified learning-objectives. Activities can be combined into two types of activity-structures. First an activity sequence by which the activities have to be performed in the order as specified in the structure. Second an activity selection, by which a given number of activities may be selected from the number present in the selection. These activities are performed in an environment consisting of learning objects and learning services (communication, search, collaboration, etc.). Which activities and which order these activities have to be carried out can be specified per role. LD uses the metaphor of a theatrical play for this purpose. For this purpose LD consists of one or more plays; a play consists of one or more sequential act or acts; an act consists of one or more concurrent role-parts. The role-part specifies the activity a role has to perform when the act is started. The act synchronizes activities of the different roles in time. A role-part of the next act can only be accessed when the current act is completed. There are several conditional constructs that control the completion of an act which allows the creation of cohorts of user working together. An example of this is the synchronization of tutors and learners via an act to ensure a sufficient number of tutors will be available when the learners start with their next activities. Finally, the play sequences the acts in such a manner that it meets the learning objectives, given certain prerequisites.

LD is implemented as an XML (W3C, 2003) binding. The actual learning design documents are expressed in XML are known as units of learning (UOL). The detailed specification itself can be obtained from the IMS website (http://imsglobal.org). We assume the reader has ample background knowledge about the major constructs of LD. When we refer to LD we always are referring to the level-c variant of the specification which is the most elaborate one.

LD is a declarative language meaning that it describes what an implementation supporting LD must do. LD does not state how this should be done. Furthermore LD is an expressive language which means that it has the ability to express a learning design in a clear, natural, intuitive, and concise way, closest to the original problem formulation. Both LD’s expressiveness and declarative nature make it ideal for its target audience of educational designers, but difficult for implementers because knowledge about the domain is required and implementation routes and strategies are not obvious.

The following XML code is an example of small part of an LD instance.

```
<imsld:roles identifier="roles">
  <imsld:learner identifier="novice" min-persons="5"
                 max-persons="10">
    <imsld:title>Novice students</imsld:title>
  </imsld:learner>
  <imsld:learner identifier="advanced" min-persons="1"
                 max-persons="5" create-new=”allowed”>
    <imsld:title>Advanced students</imsld:title>
```

```
  </imsld:learner>
</imsld:roles>
```

The example code above demonstrates both the declarative nature of LD and its expressiveness. Notice that two roles are declared with attributes stating the minimum and maximum number of members for each defined role. For the second learner role it is possible to have N instances of this role during execution time due to the declaration of the create-new attribute. LD does not make any assumptions about how, when and who should be assigned to these roles nor does it state how and when the mentioned constraints should be checked. It merely declares valid states.

Another example below of a part of an UOL shows how LD can express dynamic behavior is a very declarative manner.

```
<imsld:complete-act>
  <imsld:when-condition-true>
    <imsld:role-ref ref=”tutor”/>
    <imsld:expression>
      <imsld:complete-support-activity-ref ref=”mark-assignment1”/>
    </imsld:expression>
  </imsld:when-condition-true>
</imsld:complete-act>
```

This example states that an act will be completed when all tutors have completed a certain support activity with id ‘marking-assignments1’. Apparently LD expects that the completion of activities will be tracked during run time (at least for the activity with id ‘mark-assignment1’) and that the activity is completed for all users in role tutor. Again how this is achieved is left up to the implementers of the specification. LD merely specifies valid state transitions.

To produce the learning experience expressed by an UOL, a software component capable of interpreting this UOL is needed. This component is referred to as an ‘engine’. The output of an engine will be a personalized version of the UOL in XML format according to all rules defined by LD. This article will demonstrate how an engine can be implemented relatively simply when approached from the perspective of a finite state machine (FSM) (Sipser, 1997). By extending the LD’s native property mechanism with new properties, each state is reflected by a set of properties. We will see that state transitions are realized via events and event handlers. With the FSM machine in place, execution of a UOL can be reduced to personalization of pre-parsed content. How the content is pre-parsed and persisted is part of what we call the publication process. Finally, we will see that personalization is a matter of a simple XML translation.

The engine as a collection of finite state machines

At the heart of LD are interactions, between users in particular roles or between users and the engine. The results of these interactions can be captured in properties. Properties can be explicitly declared in LD, but there are also properties in LD that are presupposed to exist. An example is a property that captures the completion status of an activity for every individual user. We will call these properties *implicit properties.*

The property mechanism defines an FSM for each *individual user*. An FSM consists of a set of states, a start state, an input alphabet and a transition function that maps an input symbol and current state to next state. An engine will always deal with multiple users, and so the engine is a collection of FSMs. FSMs offer a logical, methodical approach towards sequential input processing, that is relatively easy to design and implement which avoids error-prone conditional programming.

Within the context of LD, each state is represented by the set of values of all the properties that are either defined explicitly or implicitly by the LD. Properties are defined during a so called publication process. An UOL is parsed and analyzed by the engine during which all explicit and all needed implicit properties are defined and persisted in a database. All users will have their own values for these properties representing their state at any time. Execution of this UOL consists of personalizing the UOL for the user which is in fact adapting the UOL according the property values of this user. A state represents the position of a user with respect to his or her progress in the UOL. The start state is defined by the initial values of the properties. These initial values are either given in LD or are set as results from executing other UOLs at earlier stages. The input alphabet is made up of all LD constructs and the transition functions are defined by LD constructs dealing with interactions. When, for example, the engine provides feedback on completion of an activity, the engine reacts to a user action, namely completing an activity. In terms of an FSM, this can be formulated as follows: the engine responds to a change of state that is caused by the user completing an activity.

There are a number of cases defined in LD where the change of state should cause another change of state. A fairly obvious example is the *change-property-value* LD construct that can be triggered by the completion of an activity. In order to cope with these LD constructs when using an FSM, the definition of a FSM must be extended, to allow each state to have an output that itself can be an input for the FSM. This type of final state machine is also known as a Moore machine (Sipser, 1997). By introducing this feedback loop, we should be able to deal with chains of state changes that can occur through several LD constructs.

The subsequent sections explain in depth how the concept of FSM is implemented in the engine. First the concepts of runs and roles are introduced; these concepts together with the user are the primary key when accessing a single FSM from the collection of FSMs. The next section shows how each state is persisted by the use of properties. A number of property types can be distinguished each with their own characteristics and use. The following section deals with the transition function of the FSM. The concept of an event is introduced as the core of both alphabets. It will become clear how the engine is capable of dealing with these events. Then we will return to the start of the process, explaining the importance of the pre-processing the UOL. Finally, bringing all the previous concepts together, personalization will be shown to have become a straightforward XML transformation.

Populating the UOL

Before a UOL can be ‘executed’, users (learners, staff, etc.) have to be assigned to it. LD does not refer to users directly, but uses an indication via roles for this purpose. It is the engine’s responsibility to bind actual users to abstract role. A ‘run’ is introduced as a pedagogically neutral term for binding a group of users to an UOL via a publication.

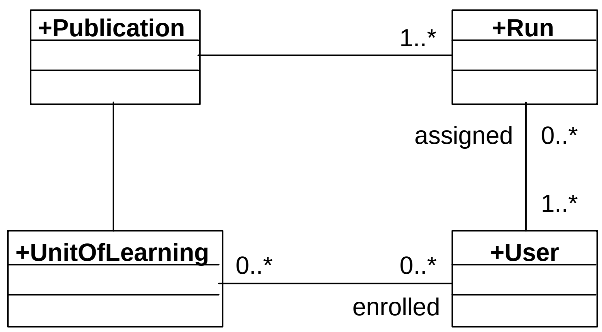

Figure 1: runs

Figure 1: runs, depicting an UML (OMG, 2003) class diagram of a run, shows the run as intermediate between users enrolled for a UOL and a publication for this UOL. Each run has one or more users assigned to it, forming the community of users taking part in the UOL together at the same time. Users can enroll for a particular UOL and are assigned to one or more runs for the UOL. A run has exactly one publication assigned to it, which in turn is associated with exactly one UOL. For now, it is sufficient to understand that a publication is the result of pre-processing an UOL so it can easily be processed by the engine during execution of the UOL. For each publication one or more runs may exist, allowing parallel execution of the same UOL.

Runs provide a mechanism for binding users to the UOL, allowing at the same time multiple reuse of the same UOL, both sequentially and in parallel. Furthermore, it allows users to be grouped together in cohorts. However, individual users still must be mapped to the roles defined in the UOL. In order to satisfy this requirement two new constructs are introduced: ‘role-participation’ and ‘run-participation’. Role-participation defines what roles a user may assume when participating in a run. Run-participation defines the active role for a user in a run at any moment in time.

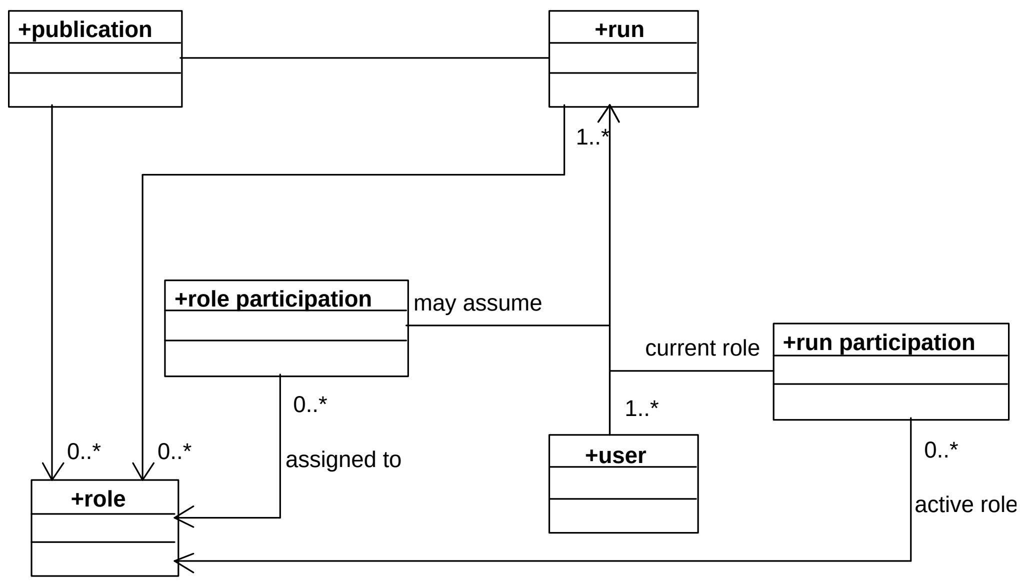

Figure 2: relation between run and role

Figure 2 depicts the relationships in a UML class diagram. LD specifies that it is possible to have multiple instances for some roles and the figure shows that the potential roles are associated with the publication as well as with the run. Role instances can be dynamically created during execution of the UOL as defined by LD. For a UOL to be re-usable, these newly created instances of the roles cannot be associated with the publication since they are different for each run. As a result, some of the roles are associated with the run and should be considered copies (or instances) of roles defined in the UOL. The difference between roles associated with the publication and those associated with the run is reflected in the way information about them is persisted. Information about roles associated with the publication is stored through global UOL properties whereas information about roles associated with the run is stored through local UOL properties. In the following section the difference is explained in more detail but for now it will suffice to understand that global UOL properties have the same value for all runs of the same UOL and that in contrast local UOL properties can have different values for each run of the same UOL.

With the addition of role-participation and run-participation, all members of a particular role can be determined, thereby satisfying the last remaining requirement with regard to user population i.e. assigning individual users to roles.

How, why, when and by whom users are assigned to roles is not part of the functionality of the engine. This is very much dependent on the business model of the party incorporating the engine and is considered out of scope for the engine. The engine, however, provides interfaces allowing the manipulation of the model presented in Figure 2. When doing so, the engine enforces the rules implied by both the model and the UOL preventing the system getting into a state not allowed by the UOL.

We will see that the engine is a collection of FSMs and that the user, run and role are the primary key when determining which FSM is being referred to at any point in time during execution. Before going into more detail, the property mechanism which is key in defining a state is discussed in the next section.

Properties

Properties represent data that needs to be persisted and each property consists of a property definition with one or more property values. The Properties can be either defined directly by LD, which makes it an *explicit property*, or can be presupposed which makes it an *implicit property*. The property definition determines the type, the default value, the scope and owner of each property. Initial values are used as the initial state for the FSM. The scope of a property is either local, which means that it is bound to the context of a run or global which means there is no direct relation with a run. The owner defines to whom or what a property belongs. The combination of scope and owner determines when and how properties are instantiated. The term ‘instantiated’ is informed by the world of object orientation. A property is instantiated when a new instance of a property, here a new persistent data store, is created according to its definition. The new property is assigned the initial property value of its corresponding property definition. The implicit value ‘null’ is assigned when no initial property value is defined. This is only needed for explicit properties as implicit properties always have an initial value which is set by the engine when creating this property.

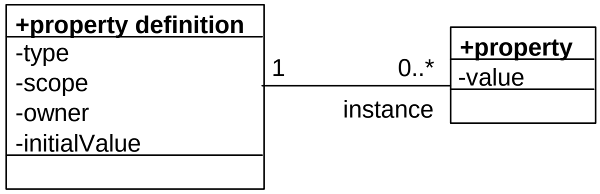

Figure 3: property definition and properties

Figure 3 shows an UML class diagram of a property definition and its instantiated properties. How and when properties should be instantiated is determined by the scope and owner. Table 1, shows valid combinations of scope and owner and describes the instantiation moment and the impact of this for the state.

Table 1 property types per scope and owner

		Scope	
		Local	**Global**
Owner	**User**	A property is instantiated for every user for every run. Parallel runs can result in different states per run as the values may vary per run.	A property is instantiated once for every user. This part of a user’s state is the same for every run.
Owner	**UOL**	A property is instantiated for each run. The property is a part of the state of all users of a run.	A property is instantiated for each UOL and is used for persisting results from the parser. This property isn’t part of anyone’s state.

Role	A property is instantiated for each role in each run. The property is part of the state for all the users in the group.	
None		A single property is instantiated once and typically contains information about the environment. This property isn't part of anyone's state.

There are some interesting things to note from this table. First of all it becomes apparent that the state of a user comprises a number of sets of properties. Some sets are unique per individual, others for each individual in a run and some sets are common between groups of persons in a particular role or to individuals in a run. Note that scope and owner applies both to implicit and explicit properties.

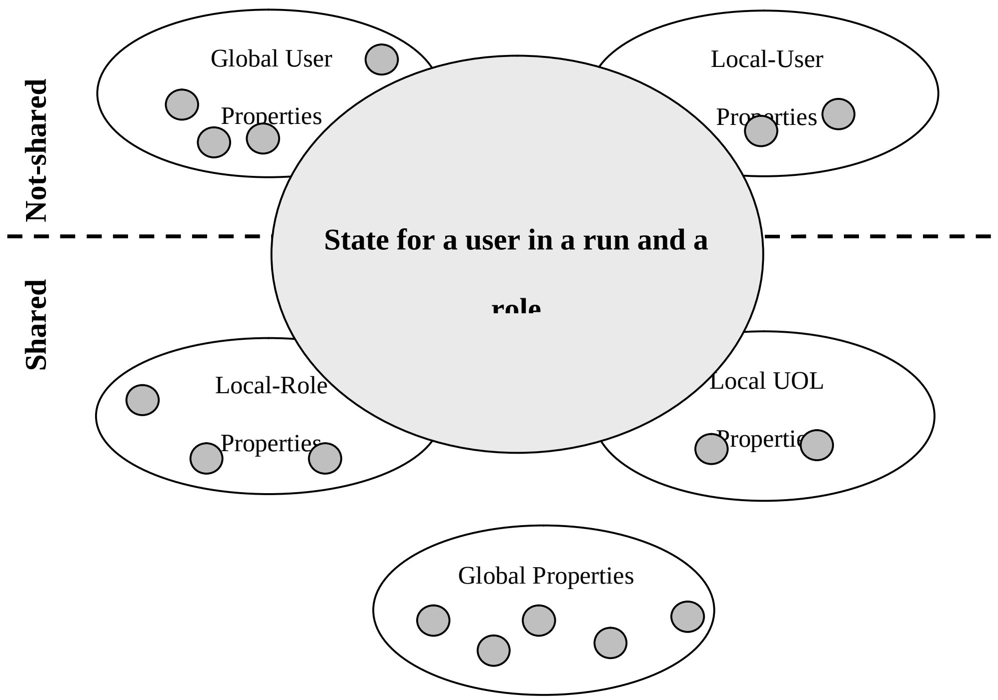

Figure 4: state as combination of sets of properties

Figure 4 shows how the different sets of properties make up the state for a particular user. Note that part of the state is shared amongst users and that a user can have more than one state at any moment in time if we take the perspective of the engine as a collection of FSMs. This can be explained from the fact that the state is not purely related to the user, but also to the run and the role in which the user is participating. So, from the perspective of the engine as a collection of FSMs, the user, run and role are the primary key when determining which FSM is being referred at any point in time. The collection of all states for a user is also known as the user's dossier. Since the FSMs are for a part making use of the same properties, manipulating these properties propagate to all the involved FSMs. This also explains why the initial state for one FSM could be influenced by the final state of another FSM. This interlocking of FSMs provides a mechanism for dealing with group behavior in the engine.

It is important to understand that the engine is responsible for determining the scope and owner for each of the implicit properties it defines. In the example at the beginning of this section it was mentioned that the engine is responsible for adding completed properties for a number of constructs. The engine is also responsible for determining what the ownership and scope of each of the completed properties should be. In the example *learning-activity, support-activity, activity-structure, role-part, act, play,* and *unit-of-learning* were mentioned. The owner and scope for all these completed properties should be user and local. This is true for all except for the *unit-of-learning*. The completion of the *unit-of-learning* can be relevant beyond the run, for example in a curriculum, and its scope should therefore be global. These types of considerations should be made carefully for each implicit property that is introduced.

The second issue to notice from Table 1 is that a new type of property, the global UOL property, has been added in addition to the ones that are defined in LD. It is a special category of Global UOL properties, not known in LD, that are used by the engine to facilitate persistence of the parsing results during the pre-processing. Parsing converts the UOL into a format that can be easily interpreted during the personalization stage. The results of this parsing consist of XML documents, derived from the original UOL. The newly created XML documents are stored in global UOL properties. By doing so, the engine extends the use of properties as mechanism for persisting state for the FSM towards a more generic store. The extension allows an efficient implementation of the engine with minimal code and optimal re-use.

Event handling

We have seen that properties provide the means for persisting state of a user (even multiple states). In order to complete the idea of FSMs we need a transition function that is capable of changing the state on the basis of an input alphabet. As noted earlier, the engine will be a Moore machine, making it necessary to have a mechanism that can react to a change of a state in the manner required by LD for some of its constructs. These reactions will form the output alphabet.

LD provides some instructions to let the user manipulate properties, and thereby state, directly. Examples are the *set-property* or *user-choice* instructions. However, most constructs change property values in a more indirect fashion.

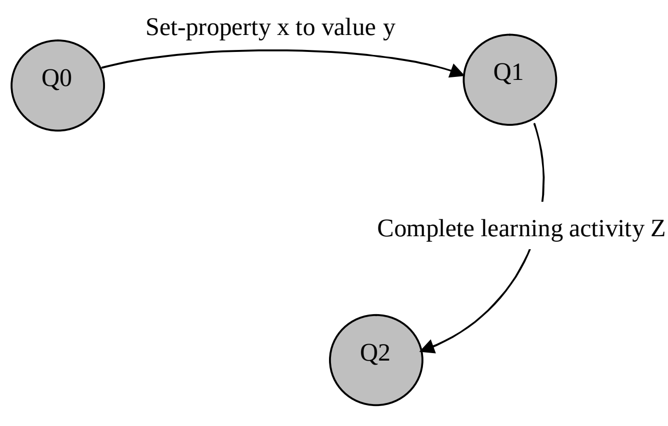

Figure 5: example state diagram

Figure 5 shows an example FSM responding to the input alphabet. Q0 represents the start state for the state machine for a particular user, run and role. The user interacts via the engine by manually setting a property and thereby changing state. The input is represented by the edge between Q0 and Q1. We assume that the UOL for which this state machine is drawn, contains a conditional construct stating that setting property x to value y should result in the completion of learning activity Z. The result of this output is state Q2 and the output itself is represented by the edge between Q1 and Q2.

The obvious questions arising are: what are the alphabets and how can they be ‘read’ and ‘written’? The answer to the first question can be found be thinking of both alphabets in terms of events. Everything that can change the state of an FSM is considered to be an event and the collection of events thus forms the input alphabet of the FSM. The output alphabet consists of the input alphabet extended with additional events as a result of the LD semantics. The input and output alphabet will of course vary from one UOL to another as the properties defined in the UOL will differ and therefore also the potential events. However, events can be classified and are limited to only two classes. These two classes are: property events that are triggered whenever a property value is changed and timer events that are triggered after a defined duration of time.

The output alphabet can consist of events triggered on the basis of changed property values and a number of events that will not cause any state changes. Among the latter are events triggering *notifications* and *e-mail messages.* The remainder of this section deals with the implementation of the event processing mechanism in the engine. Figure 6, shows the architecture of the event handling mechanism of the engine. The property store contains all states of all users. Whenever a property value is changed the property store raises a new event. This event is captured by the event dispatcher.

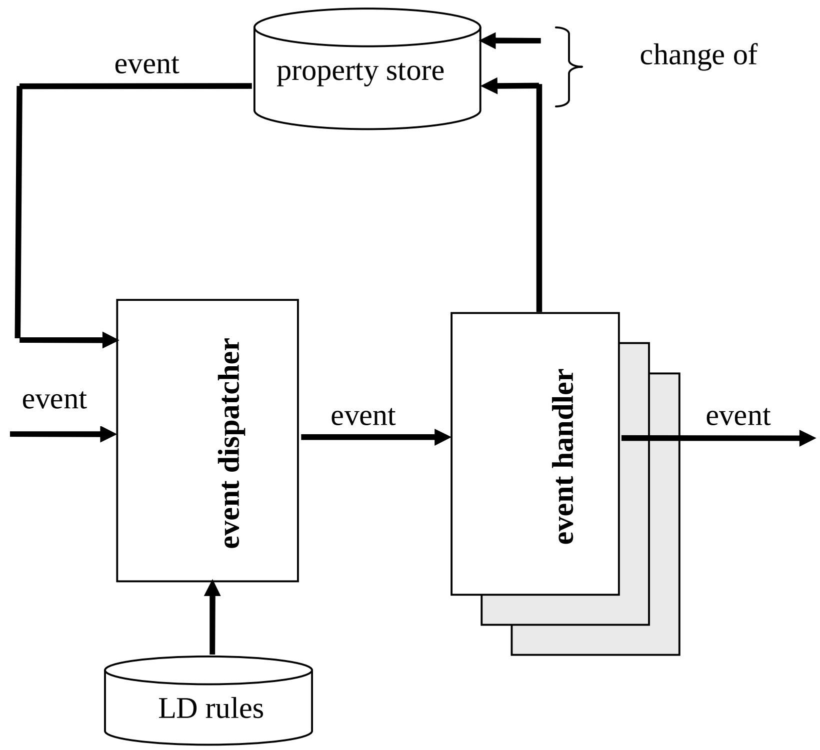

Figure 6: overview event handling mechanism

The event dispatcher consults a store containing the rules defined by LD. This store is filled with information during the pre-processing of the LD instances. With this information the event dispatcher knows what needs to happen next. In most cases, no information is found in this rule store, meaning no further action is needed. However, on some occasions, information is found, indicating what the next step that needs to happen. With the retrieved information, the event dispatcher can determine which event handler to call. Each of the event handlers represents a type of LD rule. For LD quite a number of event handlers can be defined amongst which are handlers that process the completion of *unit-of-learning, act, play* and *role-parts,* and handlers that deal with the conditional constructs in general. These event handlers react by changing one or more properties when certain conditions defined by the business rule in LD are fulfilled. This, in turn, causes one or more new events to be raised forming a chain of events. The event handlers do not necessarily react by changing property values. They may raise events triggering notifications or e-mail messages. Notice that an event can trigger zero, one or more event handlers and that an event handler can change zero, one or more properties. Furthermore, the change of properties can supersede the scope of a single FSM because the same properties can be shared amongst different FSMs. Therefore multiple FSMs can change state simultaneously as a result of a single event. This characteristic ensures propagation and as a result, the synchronization of different roles and groups working together. This propagation can occur within the perspective of a single user having multiple FSMs (one for every role the user may assume) or within the perspective of groups with a run or even at the level of the whole user community known to the engine. It is important to understand that in order for this mechanism to function properly, state changes propagating over several FSMs are considered as atomic actions.

Timer events do not start with a change of a property value, but are raised by some timer. The rest of the event handling mechanism is exactly the same as for events raised through change of a property value. It is clear that there is a risk of recursion causing endless loops. It is the responsibility of the validation process during the pre-processing stage to detect these recursions (see below).

Publication

A publication is the result of pre-processing a UOL. We have already seen that the properties and event handling mechanisms depend on the outcome of this process. The part of the engine responsible for this process is called the publication engine.

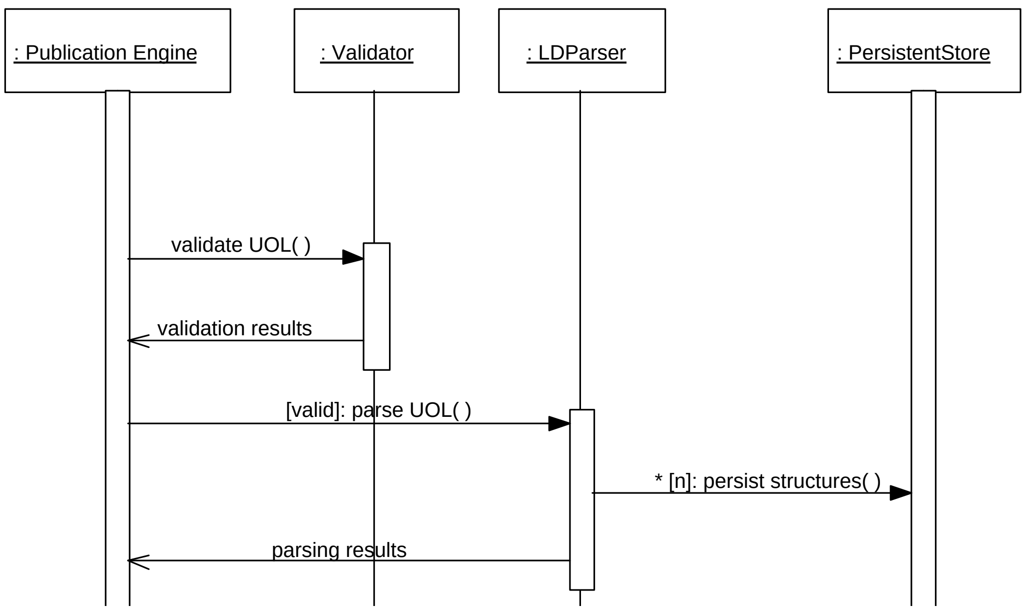

Figure 7: publication process

Figure 7: publication process, shows a UML sequence diagram representing the publication process. The first step of the publication process is to check the UOL validity. Validation covers a numbers of aspects. The UOL is checked for completeness, that is, whether all locally referenced resources are also included in the UOL. The UOL is validated against the LD schema using a validating parser (for example *Xerces*). These types of validation are straightforward and revolve around XML technology. More interesting types of validation cover the semantics of a UOL.

All references are checked to determine if no erroneous cross-references have been made. Examples of such errors would be a *role-ref* referring to a *property*. Another type of semantic validation includes the checks for invalid attribute values. For example, such as when the minimum number of persons in a role exceeds the maximum number of persons in a role. Recursions can occur whenever and wherever elements can include other elements by reference. The *environment* element is a good example of such a construct. Checking for recursion is especially important for preventing event handlers falling to endless loops.

If the validation is successful, the LD parser will is invoked. The LD Parser converts the LD into a format that can be easily interpreted during the execution phase. This intermediate XML format is used during the personalization stage. As noted earlier, global UOL properties are used to store these small XML documents. It is important to highlight that the actual resource is not part of such an XML document but is stored separately on a web server and are referenced from these XML documents.

Another important result of the parsing process is the store containing rules that should be applied for a UOL. These entries are retrieved by the event dispatcher in order to determine what actions need to be taken when an event occurs. Finally the publication process is responsible for creating all relevant property definitions for both all the explicit and all the implicit properties.

Personalization

A UOL is executed when a user accesses a run of a UOL in a *role* which should result in an adapted view of the UOL according to this role and the user’s property values. This adaptation process is known as personalization and is one of the core requirements of LD. Personalization involves adaptation of the LD according to rules defined by LD,. which describe how the engine should react to certain states. An example is feedback, which only should be provided when the corresponding activity has been completed, in other words when a certain state has been reached. Remember that states are constructed by sets of properties.

Once the FSM has been in place, personalization and therewith execution of LD becomes relative straightforward as most of the work has already been done by the event handling mechanism.

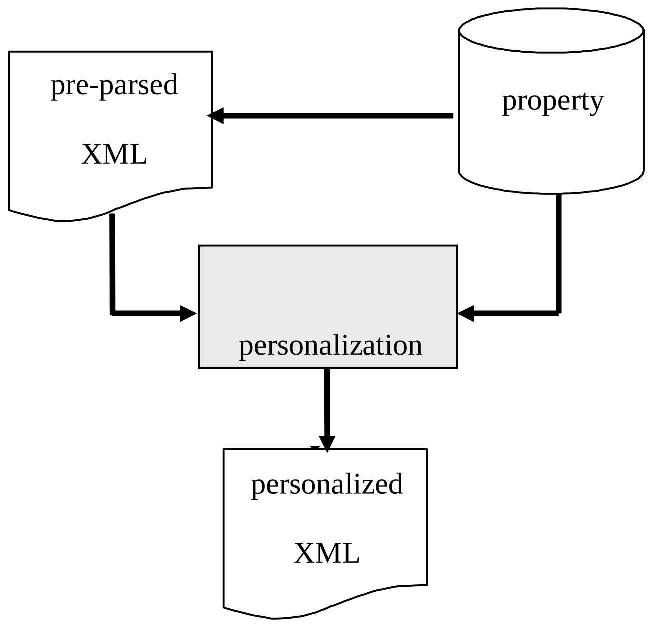

Figure 8: the personalization process

The result of the personalization process as shown in Figure 8, is a personalized XML document. This is achieved by merging the XML document that was stored as result of the publication, with the property values from the persistent property store. Note that the original XML document is stored as a global UOL property. How the pre-parsed XML document is merged with the property values varies slightly per type of element and corresponding rules. The process can result in the replacement, addition or removal of some XML elements. Although there are a number of personalization types defined in LD, we can classify them into the following three classes:

- Personalize the activity tree. An activity tree is the combination of all ***play***s and their sub-elements. The activity tree is personalized on the basis of the current FSM defined by the run and the current role of the user. Further personalization takes place on the basis of completed and visibility properties which were introduced earlier. The outcome will be an XML representation of the activity tree reflecting the current status of the user.

- Personalize the environment tree associated with an activity. The environment tree is adapted using visibility properties in a similar way as is the activity tree, resulting is an. XML representation of the activity tree reflecting the current status of the user.
- Personalize the content of various LD constructs. References to properties are replaced by their actual contents and parts of the content may be hidden on the basis of the value for the different class properties. Class properties are implicit properties created during publication which reflect the visibility status (hidden or visible) for classes of content.

In conclusion, it can be said that once the FSM mechanism is in place, personalization is reduced to a simple XML transformation obeying the rules of LD.

Implementations

The Open University of the Netherlands developed the predecessor of LD, called EML (Hermans, Manderveld, & Vogten, 2004) in 1998. EML has very similar objectives to those of LD, although it isn’t an open specification and the actual tagging of the XML language is quite different. The consecutive versions of EML have resulted in a number of players. A first prototype was developed in 1999 as a proof of concept, followed shortly after by the first system, called Edubox which went into regular exploitation at the OUNL in September 2000.

Recently we implemented an alpha version of an open source LD engine with the name ‘CopperCore’, which was partly funded by the European Commission via the Alfanet (IST-2001-33288) project. This engine was built using the approach outlined in this article and has been made available as an open source product through SourceForge. The analysis and ideas presented in this article were based on previous experience with the implementations of the Edubox player and put into practice in the CopperCore engine. The first release supports the view that the approach presented in this article results in an elegant, lightweight design capable of supporting the complete LD specification.

Conclusions

With the arrival of the latest specifications and standards for e-learning, the sophistication, expressiveness and complexity have increased considerably. Simple transformations are not adequate to implement these specifications and standards successfully. LD is a representative of such a new specification. Its declarative nature and expressiveness increases the complexity for any implementation. This is probably the highest hurdle that stands in the way of successful general deployment of this type of specification. Work needs to be done to help the community of implementers to lower this hurdle.

In this article we have shown that by taking the approach of an FSM, it is possible to break down a complex specification like LD into a fewer basic constructs allowing elegant and relative lightweight designs and implementations. This breakdown is accomplished by exploiting the property mechanism beyond its direct usage in LD itself. The use of implicit properties helps harmonize the different kind of rules defined in LD, and reduces them to simple property operations. Furthermore the property mechanism acts as a store for the result of the publication process especially for the pre-parsed XML content. The event mechanism helps break down the large number of rules down to their basics in the form of event handlers. These event handlers each have dedicated tasks, dealing with different aspects of the rules laid down by LD, but all have the same basic mechanism which again helps reduce the complexity enormously. Decomposition of the complexity is essential and is achieved by the fact that implementers only have to focus on the proper implementation of the event handlers themselves. Implementers of an event handler do not have to worry about the bigger picture as it is dealt with by the event handling mechanism. The same event handling mechanism ensures that reactions to certain events are adequately propagated throughout the whole system. By doing so, all group and role dynamics are automatically incorporated into the engine without additional efforts as the engine is considered to a collection of FSMs. By the introduction of the run and the roles, it has become clear what should be considered as primary key for each of the FSMs. We have shown that by selecting the right owner and scope of the properties we can interlock the FSMs which result in the correct propagation of state changes automatically. Again no additional efforts have to be made because the event handling mechanism propagates state changes throughout all interlocked FSMs.

With these constructs in mind, implementation of an engine has not become simple, but far less complex than may have been anticipated at first sight.

Acknowledgements

The authors wish to thank the management and staff of the Schloss Dagstuhl International Conference and Research Center for Computer Science for providing a pleasant, stimulating and well organized environment for the writing of this article.

References

IEEE Learning Technology Standards Committee (2003). Retrieved from Website of Learning Technology Standards Committee: http://ltsc.ieee.org

Hermans, H., Manderveld, J., & Vogten, H. (2004). Educational Modelling Language. In W. Jochems, J. van Merriënboer, & R. Koper (Eds.), *Open and Flexible Learning. integrated E-LEARNING implications for pedagogy, technology & organization* (pp. 80-99) (chap. 6). London, New York: RoutledgeFalmer.

IMS (2003a). *IMS Global Learning Consortium.* Retrieved November 11, 2003a, from Website of IMS Global Learning Consortium: http://www.imsglobal.org

IMS (2003b). *IMS Learning Design.*Information Model, Best Practice and Implementation Guide, Binding document, Schemas. Retrieved November 11, 2003b, from Website of IMS Global Learning Consortium: http://www.imsglobal.org/learningdesign/index.cfm

Koper, R., & Manderveld, J. (2004). Educational Modelling Language: Modelling reusable, interoperable, rich and personalised units of learning. *British Journal of Educational Technology (in press),* Koper, R., & Olivier, B. (2003). Representing the learning design of units of learning. *Educational Technology and Society*,

OMG (2003). *Unified Modeling Language (UML).* Retrieved November 06, 2003, from http://www.omg.org

Sipser, M. (1997). *Introduction to the Theory of Computation.* PWS Publishing Company.

W3C (2003). *XML Extensible Markup Language.* Retrieved November 05, 2003, from http://www.w3.org/XML/