5 An Architecture for Learning Design Engines

Hubert Vogten, Rob Koper, Harrie Martens, Colin Tattersall

Educational Technology Expertise Centre,
Open University of the Netherlands, Heerlen, The Netherlands

5.1 Introduction

Learning Design (LD 2003) 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. This expressiveness and declarative nature complicate the implementation of an engine that can interpret the specification. As a result, the main objective of this chapter will be to describe how such an engine can be implemented. We will provide guidelines which go beyond the published specification to help implementers incorporate LD into their products. The approach is generic in nature and has been tested in the CopperCore engine described in Chap. 6 of this book. We note, however, that the user interface aspects of the engine are considered to be out of scope for this chapter. These considerations are heavily influenced by the environment in which the engine is incorporated, and are not easily able to be generalized. LD specifies few requirements that have a direct impact on the user interface design.

To illustrate both the declarative and expressive nature of LD, consider the following XML code fragment.

```xml
<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>
```
Two roles, novice and advanced learner, 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 Unit of Learning (UOL) fragment shows how LD can express dynamic behavior in a very declarative manner:

```xml
<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 `mark-assignment1`. The LD specification makes the assumption that the completion of activities will be tracked during runtime (at least for the activity with id `mark-assignment1`) and that the activity will be completed for all users in role tutor. Again, how this is achieved is left up to those implementing the specification. LD merely specifies valid state transitions.

An engine is needed to present the learning activities to learners as expressed by a UOL. The output of the engine will be a personalized version of the UOL in XML format according to the rules defined by LD. The approach we take in this chapter is to demonstrate how an LD engine implementation can benefit from the perspective of finite state machines, FSMs (Sipser 1997). FSMs offer a logical, methodical approach towards sequential input processing, which is relatively easy to design and implement and which avoids error-prone conditional programming. They are a proven concept that allows for efficient and effective implementations.

### 5.2 Learning Design Engines as Collections of Finite State Machines

At the heart of LD are interactions between users in particular roles or between users and the LD system. The results of these interactions can be captured in properties which can be declared explicitly in LD. We further