Awareness of Execution in Designing Learning Activities with LAMS and IMS LD

Juan Manuel Dodero
Departamento de Lenguajes y Sistemas Informáticos
Universidad de Cádiz, Spain

Jorge Torres
Departamento de Electrónica y Sistemas Computacionales
Tecnológico de Monterrey, Mexico

Learning design languages such as IMS LD and LAMS can be used to describe execution issues of a learning process, but they depict different degrees of expressiveness. This paper classifies the execution issues, explains how these can be described with both languages, and compares them on the basis of LPCEL, a service-based framework that provides a language to describe the composition and execution of learning processes.

Introduction

Computer-aided design and execution of learning activities try to describe a learning experience from the point of view of the tasks that participants have to realize and the learning resources involved. That sort of design activities are generally known as *learning design* (LD), which aims at taking prescriptions from instructional design theory plus examples of best practices and applying them for defining learning courses ready to be run on the computer (Koper, 2005).

The IMS LD specification includes a language to describe learning experiences as *units of learning*, and provides a conceptual framework for the specification of learning process designs, which have to be afterwards deployed onto a computer-based execution engine. The LAMS approach has taken a step further by integrating design and execution of learning activities into the same LD language, design environment and execution engine. This paper focuses on discussing a number of design issues, which not always can be resolved at design time, but on the run time instead, as long as certain helper primitives are provided by the LD language. We describe a classification for such execution-aware design issues and provide a compared analysis of how IMS LD and LAMS tackle them, on the basis of LPCEL (Learning Process Execution and Composition Language), which provides a framework including the appropriate languages primitives to describe execution-aware learning designs.

Related work

Early efforts in standardizing the description of computer-assisted learning courses, such as SCORM, were basically content-oriented. Later versions stepped from describing only contents to describe simple learning activities, as the IMS *simple sequencing* (SS) specification was incorporated into the SCORM 2004. But sequencing primitives were limited to guide the navigation among individually-driven content-based resources. The IMS LD and other specifications in which it was inspired (e.g. OUNL EML) provided the conceptual basis for describing collaborative learning activities. Nevertheless, the coincidence of some issues approached by both OUNL EML and IMS SS motivated the IMS LD being deprived off relevant primitives to describe learning activity flow. LAMS grew as a de facto, not standards-aimed initiative that fulfilled the requirements to describe collaborative learning flows.

The description of collaborative learning flows involves several issues to solve, which are classified by Torres et al. (2005a) as a learning processes characterization framework that includes consideration for

pedagogical diversity, learning flow complexity, dynamic composition ability, separation of process vs. service, decentralization of services, learning service availability and containment, and synchronization ability. Since this work does not deal with a learning services model, we have focused only on the features aimed at designing collaborative learning flows aware of execution issues, on the basis of current LD languages. Both IMS LD and LAMS have provided LD languages which, though presenting remarkable differences, aim at the same goal of describing process-oriented LDs (Marjanovic, 2005). In the next section we discuss on how these LD languages can tackle execution issues at design time.

Execution-aware Learning Design Issues

The learning design of a course using an educational modeling language presents some issues that can be approached either at design-time or at run-time. If you define all activities and assign users at design time, you have a too rigid scheme that cannot be adapted to the concrete events coming out from executing the learning sequences. On the contrary, if you leave all activities and task assignments to the run time, you have a more flexible scheme, but you have no learning design at all. LD languages as LAMS or IMS LD adopt a mixed approach, define some elements at design time, and let other elements to the run time. For instance, LAMS *gates* and *grouping* activities, or IMS LD Level B elements, are used to set up design-time solutions prepared to manage run-time issues, such as binding user groups to running activities. In the following, we describe such issues and characterize them classified in three main categories, namely activity structuring, learning flow, and synchronization, as shown in Fig.1. Six major issues have been identified within these categories.

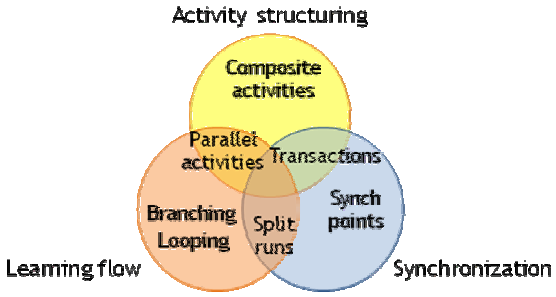

Figure 1: Learning process design issues to be considered during execution

Issue 1: Composite Activities

When designing a learning process, diverse ways of describing the executable structure of a group of activities are often needed. Composite activities are patterned groups of activities that can be sequenced, selected, concurrently executed, etc. For instance, during a typical computer programming course you can find the following requirement: “The students must complete both theoretical and practical activities; they have to do theoretical readings and completing a practical programming task assignment; both must realized concurrently on a (given) period of time.”

On the one hand, to realize a learning design for these requirements with IMS LD, *activity-structures* can be used, but they can be only sequenced or selected. If you need concurrent activities, these can be defined at the *play* level, but this imposes an obscure re-design of the *method*. An alternative is defining different *role-parts* within the same *act*, which does not make sense when a single learner is executing the activities. A third alternative is managing activities’ *completion status* properties, but this is too burdensome when a large number of activities are considered. On the other hand, LAMS provides sequenced, parallel, and optional activities that comprise the *complex activity* type, which makes easier to describe such concurrent composite activities.

Issue 2: Conditional Branching and Looping

The usual learning flow of activities must be branched sometimes, according to a special condition that often depends on the learner profile or the results of previously executed activities, either by the same or a different user. Other times, remedial learning activities may require executing an activity or activity group an indefinite number of times, until the remediation condition is achieved by the learners.

With IMS LD, to branch an activity you must define an ad hoc *property* (depending on if you need to branch the execution for every user, a single user, or all users in a role) and a set of *conditions* to drive the learning flow through the branched activities. When LAMS LD is used, conditional branching is expressed through *branching activities*, which drive the learning flow depending on the teacher selection, grouping objects, or the output of previous tool-based activities. Although conditional branching can be readily expressed in both LD languages, the verbosity required to express branching activities with LAMS is lesser than using the equivalent IMS LD constructions.

Another learning flow-related issue is looping through a sequence of activities. That may be useful, for instance, to implement remedial learning activities (Dolonen, 2006; chap. 8). According to the IMS LD specification, "Once the user indicates the activity to be completed, then this activity stays completed in the run". Therefore, it is not possible to describe indefinite, conditionally managed loops of activities in IMS LD. A workaround for this in LAMS should consist in using a branching activity that returns to a previously executed activity depending on the output condition of another.

Issue 3: Parallel Branching on Users' learning flows

Sometimes you may require forking the learning flow of one learner, a group of learners, or all learners, to define a branching of parallel but different activities, which are executed by different groups of learners. This is achieved in IMS LD by defining *roles* or properties plus conditions. Roles must be provided and associated to concrete learning activities through *role-acts* on design time, and bound to real users at deployment time. Once the execution of an IMS LD *run* (Tattersall et al., 2005) has started, the association of users to roles cannot change. An alternative in IMS LD is using properties and conditions to change the membership of users to roles during run time. On the contrary, LAMS provides a *grouping activity* type that permits expressing that association during run time. The combination of a LAMS branching activity with a grouping activity enables forking user groups on a set of activities during run time, without requiring defining additional properties and conditions, especially burdensome if the number of activities grows.

Issue 4: Synchronization points

Setting some synchronization points can be useful in a learning flow of activities, when you have parallel running activities that must join on a given instant before proceeding with the following ones. The IMS LD specification states that *plays* always run concurrently, whilst *acts* always run in sequence. Synchronization points can be only marked *on completion* of the activities, according to user choices or time limits, or when a property is set. You need again the help of properties and conditions and, what is worse, they do not hold any semantics related to concurrent access, which is relegated to the run-time engine —e.g. CopperCore Java-based implementation engine does not implement synchronized operations to gain access to properties; it means that using properties to implement activity synchronization does not actually prevent race conditions that may cause blocking or forbidden access to activities. Using LAMS, *gate* activities can be used to synchronize learning flows of activities; gate activities hold a concrete semantics of concurrent access that IMS LD property-based synchronization do not provide.

Issue 5: Split running activities

A special case of learning flow synchronization occurs when you have two flows of activities (possibly run by different users) that must synchronize in certain points along a number of iterations. For instance, if you consider a project-based course, one learner can act as a quality manager, while another learner acts as a regular developer; the development method iterates through several steps (i.e. requirements, analysis, design, implementation, testing, etc.); throughout all the steps, the quality manager has to follow a series of activities to control the outcome from activities executed by the developer; the learning flow of both kinds of learners must be synchronized; we call this *split-running* activity flow, portraying a mix of complex running flow and synchronization. For the same reasons explained above, IMS LD does not provide adequate primitives to describe this pattern. In LAMS you can use gates and flows to describe that complex structure, although it is not a straightforward learning design task.

Issue 6: Transactional activities

Last execution issue has to do with a special form of synchronization that usually occurs in distributed computing systems, known as *transactions*. In Computer Science, a transaction is any operation whose execution must comply with four conditions, namely atomicity (i.e. guarantee that all of the subtasks of the operations are performed or none of them), consistency (i.e. the status of the data accessed must be in a legal state when the transaction begins and when it ends), isolation (i.e. no operation outside the transaction must see the data in an intermediate state), and durability (i.e. transactions must survive system failures). In distributed transactions, these properties are hard to fulfill, so they are relaxed as loose transactions that provide compensating operations when, for example, atomicity cannot be ensured. When dealing with learning activities execution, these conditions can be relaxed as well. For instance, atomicity or isolation requirements are not usual for learning activity subtasks. But you can find learning design situations where a loose learning transaction approach can be useful (for instance, providing a remedial learning activity when the learner is not able to complete an activity that can be defined as a *learning transaction*. This is a higher-level abstraction of synchronization, proposed by Torres et al. (2005b), which none of the analyzed LD languages avail.

An Instance of Learning Process Execution with LPCEL

Recently, Torres et al. (2004) have described the LPCEL language, used to describe the composition and execution of complex learning processes that can manifest execution-related issues emerged in a process-oriented LD. Whereas we focus on execution issues, we have described an adaptation mechanism that allows implementing run-time modifications on an existing IMS LD unit of learning without provoking interferences with the existing LD specifications (Zarraonandia et al., 2006a). We have applied it to remedy some synchronization and timing lacks of the IMS LD language (Zarraonandia et al., 2006b).

The part of the LPCEL conceptual model that deals with learning process *execution* is shown in Fig. 3. It includes *basic* and *complex* activity primitives to describe the execution of complex learning processes. Such activities can be regular learning activities, assessment activities or context activities, depending on what is their aim on the overall learning process. LPCEL *complex-component* activities are primitives to describe usual flow control activity structures, as well as synchronization types. Such composite activities are applied to any *component* activity, which can be either a basic or complex activity, according to the composite design pattern (Gamma et al., 1994)

Next we show an instance of how a complex learning process can be described by wrapping IMS LD with LPCEL primitives. Fig. 2 is part of the learning design for a project-oriented software engineering course, which required that “the elaboration phase […] will be completed after project planning, problem domain analysis, software architecture and use case modeling activities are completed and use cases are 80% described.” The scarce IMS LD expressiveness (Caeiro, Anido & Llamas, 2005) to describe these requirements has been solved by wrapping simple IMS LD learning activities and activity-structures inside an LPCEL *split* synchronization composite activity, which in turn is nested in an *assessment*

activity. Parallel running activities of the IMS LD activity-structure are also nested in a *join* composite activity that uses a reference to the assessment activity as synchronization condition. The LPCEL design for these execution issues spares filling the LD with the countless IMS LD level B properties and conditions required in this case.

We have also analyzed how the course requirements described above can be implemented in LAMS, finding that LAMS provides branching activities based on tool outputs that in combination with gate activities requires a less verbose implementation of the LPCEL wrapping.

```
<imsld:learning-activity identifier="LA-Project-Planning">
  <imsld:title>Project Planning</imsld:title>
</imsld:learning-activity>
<imsld:learning-activity identifier="LA-Domain-Analysis">
  <imsld:title>Problem Domain Analysis</imsld:title>
</imsld:learning-activity>
<!-- ... declaration of all learning activities... -->
<lpcel:assessment-activity identifier="AA-Eval-Elaboration-Criteria">
  <imsld:title>Evaluate Elaboration Criteria</imsld:title>
  <lpcel:split>
    <lpcel:guard>
      <imsld:conditions>
         <imsld:if>
           <imsld:is>
              <imsld:greater-than>
                 <imsld:property-ref ref="LR-Use-Case-Completed"/>
                 <imsld:property-value>80</imsld:property-value>
              </imsld:greater-than>
           </imsld:is>
         </imsld:if>
      </imsld:conditions>
    </lpcel:guard>
    <lpcel:components>
       <lpcel:component-activity-ref ref="LA-Project-Planning"/>
       <lpcel:component-activity-ref ref="LA-Domain-Analysis"/>
       <lpcel:component-activity-ref ref="LA-Software-Architecture"/>
    </lpcel:components>
  </lpcel:split>
</lpcel:assessment-activity>
...
<imsld:activity-structure identifier="AS-Elaboration-phase"
                          structure-type="complex-activity">
  <imsld:title>Elaboration phase</imsld:title>
  <lpcel:join>
    <lpcel:parallel>
      <lpcel:component-activity-ref ref="A-Project-Planning"/>
      <lpcel:component-activity-ref ref="A-Problem-Domain-Analysis"/>
      <lpcel:component-activity-ref ref="A-Use-Case-Modelling"/>
      <lpcel:component-activity-ref ref="A-Software-Architecture"/>
    </lpcel:parallel>
    <lpcel:assessment-activity-ref ref="AA-Eval-Elaboration-Criteria"/>
  </lpcel:join>
</imsld:activity-structure>
```

Figure 2: Description of an assessment activity with LPCEL and IMS LD

The other side of LPCEL consists of a set of primitives that allows describing how learning services can be composed to participate in the execution of a learning process (i.e. the *composition* side). Although the learning service-based framework that LPCEL provides is out of the objective of this paper, we briefly describe it since they are part of the tools that can be considered when designing and running a learning process. The association in LPCEL between learning process execution and learning services composition is through the *resources* element that can be linked to any *component* activity, as described in the model of Fig. 3 and 4. When such resources are remotely located, the resource *RPC-based* associated elements provide access to the required service interface, via the *service-bus* aggregation described in Fig. 5.

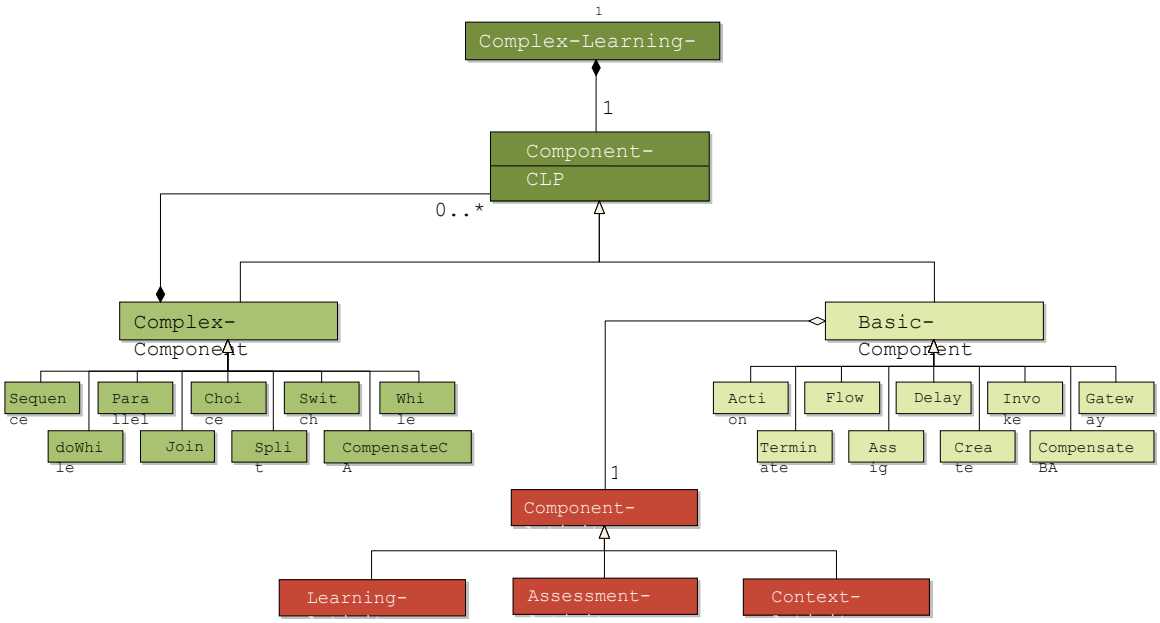

Figure 3: LPCEL conceptual model of complex learning process execution elements

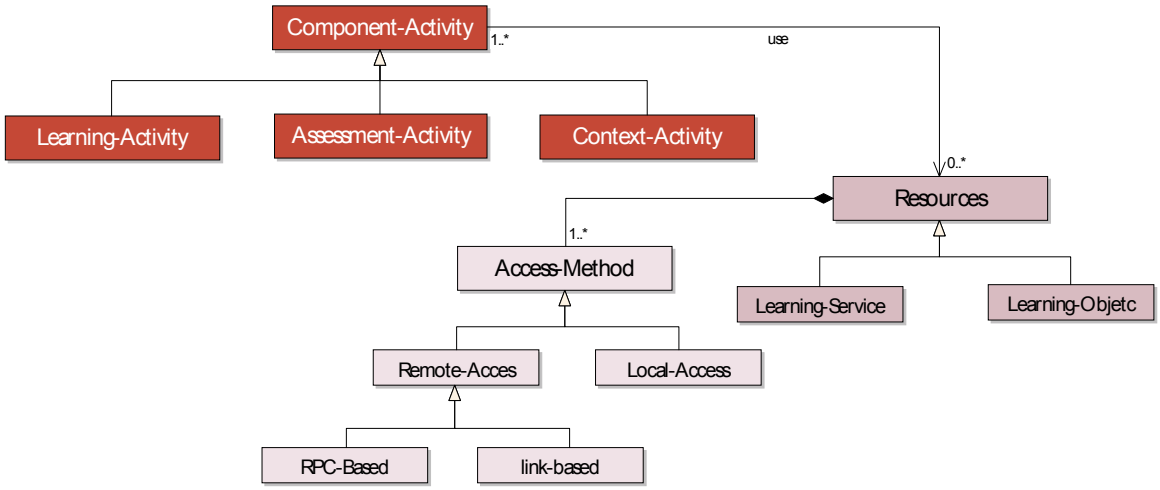

Figure 4: General resources model of a complex learning process in the LPCEL conceptual model

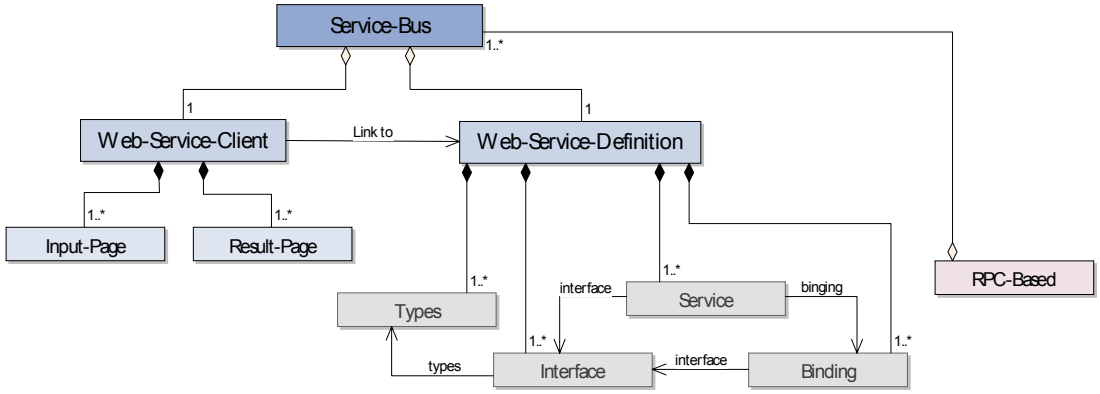

Figure 5: Service-based resources model of a complex learning process in the LPCEL conceptual model

Conclusions

In this work, we have discussed some execution issues to take into account in designing a learning process that deal with activity structuring, learning flow, and synchronization. We also analyzed how such issues can be solved with IMS LD and LAMS LD languages. Although both languages allow implementing ready-to-run learning designs, thanks to the LPCEL framework we describe how LAMS sequences are easier to adapt for execution than IMS LD units of learning. Nevertheless, in this work we have only centered on the *execution* side of the LPCEL model. On the other hand, the *composition* primitives of LPCEL also allow describing the run-time composition of learning processes by means of *services*, which we have not dealt with in this paper. In our opinion, our future work and that of LD community should pass by standardizing how learning services interfaces and intended behavior must be specified to allow for the interoperation between different LD systems and execution engines.

References

Caeiro, M., Anido, L., & Llamas, M. (2005). A Perspective and Pattern-Based Evaluation Framework of EMLs' Expressiveness for Collaborative Learning: Application to IMS LD. Proc. of ICALT'05, 51-53

Dolonen, J. A. (2006). Empirical Study of Learning Design. Tech. Report, Intermedia, University of Oslo, available at http://lemill.org/trac/wiki/ReportLearningDesign.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. M. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional.

Marjanovic, O. (2005). Towards A Web-Based Handbook of Generic, Process-Oriented Learning Designs. *Educational Technology & Society*, 8 (2), 66-82.

Koper R. (2005). An introduction to learning design. In Koper, R. and Tattersall, C. (Eds.) *Learning Design: A Handbook on Modelling and Delivering Networked Education and Training*, Berlin: Springer, 3-20.

Tattersall, C., Vogten, H., Brouns, F., Koper, R., van Rosmalen, P., Sloep, P., & van Bruggen, J. (2005). How to create flexible runtime delivery of distance learning courses. *Educational Technology & Society*, 8 (3), 226-236.

Torres, J., Dodero, J. M., Aedo, I., & Díaz, P. (2005a). A Characterization of Composition and Execution Languages for Complex Learning Processes. Proc. of Int. Conf. on Web-Based Education, Grindelwald, Switzerland, 255-260.

Torres, J., Dodero, J. M., Aedo, I., & Zarraonandia, T. (2005b). An architectural framework for composition and execution of complex learning processes. Proc. of ICALT'05, Kaohsiung, Taiwan, 143-147.

Torres, J., Dodero, J. M., Aedo, I., & Díaz, P. (2006). Designing the Execution of Learning Activities in Complex Learning Processes Using LPCEL. Proc. of ICALT'06, Kerkrade, The Netherlands, 415-419.

Zarraonandia, T., Dodero, J. M. & Fernández, C. (2006). Crosscutting Runtime Adaptations of LD Execution, *Educational Technology & Society*, 9 (1), 123-137.

Zarraonandia, T., Dodero, J.M., Fernández, C., Díaz, P., & Torres, J. (2006b). Late Modelling: a Timing of Learning Activities Approach, Proc. of ICALT'06, Kerkrade, The Netherlands, 656-658.