IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received December 10, 2018, accepted April 9, 2019, date of publication April 18, 2019, date of current version April 29, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2911884

INDIeAuthor: A Metamodel-Based Textual Language for Authoring Educational Courses

DANIEL PÉREZ-BERENGUER[1] AND JESÚS GARCÍA-MOLINA[2]
[1]Centro de Producción de Contenidos Digitales, Universidad Politécnica de Cartagena, 30202 Murcia, Spain
[2]Departamento de Informática y Sistemas, Universidad de Murcia, 30003 Murcia, Spain

Corresponding author: Daniel Pérez-Berenguer (daniel.perez@upct.es)

This work was supported by the European Union (Erasmus+ Programme) under Grant 2018-1-ES01-KA201-050924.

ABSTRACT This paper presents the INDIeAuthor authoring tool, which has been developed at the Digital Content Production Center, Polytechnic University of Cartagena (UPCT). INDIeAuthor has been developed in order for the university to have its own tool that supports all the desired features, in addition to a platform with which to investigate innovative features. When building INDIeAuthor, we have overcome some limitations identified for existing authoring tools: 1) Lack of two essential content reuse mechanisms: defined-user templates and course-independent units; 2) No support to sequence the units on a course; 3) Gamification feature is either very limited or does not exist. Two new aspects of the proposal are: providing a family of four textual domain-specific languages rather than a graphical user interface and applying model-based software engineering techniques during the implementation of the languages. Four essential aspects in the course definition can be specified to the language family: content, assessment, gamification, and sequencing. We discuss the benefits of representing courses as models and present two utilities developed as a proof of concept. This paper also contributes with the definition of a feature model that establishes a conceptual framework in which to compare authoring tools. An evaluation of INDIeAuthor is also presented: a case study was carried to evaluate the language characteristics, and the tool is contrasted with eight widely-used authoring tools. This paper presented here is the baseline of INDIe Erasmus+ European project that is currently ongoing.

INDEX TERMS Authoring tool, DSL, model-driven development, educational modeling.

I. INTRODUCTION

Higher education institutions are currently attempting to take advantage of elearning technologies to improve learning and teaching processes. Adopting elearning methods demands the creation of online educational content with the aim of providing more effective and attractive courses. Producing content is, however, a difficult and time-consuming task that requires knowledge and skills in each of the activities involved, i.e., designing learning processes, developing learning material, defining assessments or publishing content. Facilitating the creation of quality content is, therefore, a crucial factor if teachers are to be engaged in elearning practices. Institutions must, therefore, provide adequate support to teachers, such as training, guidelines and tooling. The Polytechnic University of Cartagena (UPCT) consequently created the Digital Content Production Center (DCPC) in 2014.

The associate editor coordinating the review of this manuscript and approving it for publication was Bora Onat.

As result of the UPCT's long-term vision the work of the DCPC has been mainly directed towards developing the UPCTforma content creation environment. The university aims to achieve several benefits by having its own environment: (i) a content creation platform with all the functionality desired, which can be extended at any time, (ii) avoiding the problems of having to integrate different vendors' tools, and (iii) the ability to experiment in the scope of content creation tools. As suggested in [1], a huge economy of scale could be obtained if the platform results from a collective effort, but no such initiative is planned in Spain.

UPCTforma integrates a *course authoring tool* denominated as INDIeAuthor in an *infrastructure* that provides some basic services with which to create content, such as standard-based tracking and interoperability, video production, and gamification activities. A detailed description of the component-based architecture of the UPCTforma infrastructure is presented in [2]. In this paper, we focus on INDIeAuthor.

2169-3536 © 2019 IEEE. Translations and content mining are permitted for academic research only.
Personal use is also permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

There are many course authoring tools that vary widely in their capabilities. Rather than choosing one or more of these existing tools, we have developed INDIeAuthor as part of the UPCT strategic vision commented on above. This development effort involved experimentation regarding several concerns of authoring educational courses and their content. In this paper, we present the main research contributions of our work.

We defined a feature model for authoring tools, which was used to evaluate 8 widely-used tools. In this evaluation, we identified three significant limitations of authoring tools: (i) A lack of two essential content reuse mechanisms: defined-user templates and course-independent units; (ii) The ability to sequence the course units is not supported; (iii) The gamification feature is either very limited or does not exist.

Authoring tools provide user graphical interfaces so as to facilitate their use. However, we consider that a textual authoring language could be a better choice in the case of users who are teachers of STEM fields (Science, Technology, Engineering and Mathematics). Many of these teachers have programming skills and/or use LaTeX and might, therefore, prefer a textual language to a graphical interface when producing content. A high percentage of teachers at technical universities such as the UPCT have these abilities.

INDIeAuthor has been designed as a textual authoring language that overcomes the limitations indicated above. It is actually a family of four domain-specific languages (DSL) which correspond to four main concerns or viewpoints in the creation of educational courses: content presentation, learning assessment, gamification, and content sequencing.

Model-Driven Engineering (MDE) approaches are increasingly used to build DSLs [3], [4]. INDIeAuthor has been implemented using a model-based approach to take advantage of MDE techniques and tools. The notation of the four DSLs were defined using a textual language workbench based on metamodeling. Language workbenches automatically generate the tools needed to manage DSL scripts (or programs), such as the language editor and the *tool* that converts DSL scripts into models [5], [6]. A INDIeAuthor execution engine was developed by means of model-to-text transformations [3] that automate the implementation of courses. These transformations generate the software artifacts that implement a course, which are integrated on a framework that provides common functionality.

INDIeAuthor models provide an abstract representation of the courses, which is platform-independent. This kind of representation favors the integration of INDIeAuthor with other tools, the portability of content to different platforms, and the creation of utilities with which to manage contents. As a proof of concept, we have developed a UML-based course sequencing viewer and a generator of Beamer presentations from INDIeAuthor content models.

A very limited number of works has explored how to apply MDE techniques in elearning. To the best of our knowledge, the most relevant proposals are [7] and [8], both of which focus on the definition of an operational semantic for a DSL whose objective is to automate a particular aspect of an elearning application. Instead, here we provide a detailed description of all the aspects involved in the creation of a solution based on the current practice in MDE. We also discuss the benefits of using models. In particular, we show the gain in productivity with respect to a previous manual development.

Easing content reuse and maintenance is essential if teachers are to be engaged in the creation of content and not abandon this pursuit. They will be discouraged if they come up against hurdles caused by the lack of an appropriate reuse, such as having to maintain several copies of the same content, or not being able to effortlessly reuse the same content for several courses. We have taken advantage of having a textual language to experiment with the modularization of courses, specifically, the ability for users to define templates and the definition of units as modules that can be reused on different courses.

Educational Modeling Languages (EML) [9], and IMS Learning Design in particular [10], received a great deal of attention in the past decade. Several EMLs were proposed for the modeling of three key aspects of the course design: content structuring, evaluation and learning activities. With most EMLs, modeling courses requires a great amount of effort, principally owing to the level of sophistication they offer and the use of XML-based notations. The use of EMLs is not currently a common practice, and these languages are not supported by more widespread authoring tools or learning platforms. INDIeAuthor is the result of a more pragmatic point of view than the one that underlies EMLs. The creation of courses is based on a simpler vision: individual learners follow a course that includes content and evaluation units, content can be gamified and the order in which units are accessed may depend of the learner's level of achievements. Languages that are easy for teachers to use are also offered.

The main research contributions of our work are, therefore, the following: (i) A feature model that establishes a conceptual framework in which to compare authoring tools. (ii) A family of textual DSLs whose objective is to create educational courses; these languages correspond to the main concerns on authoring courses. (iii) To show how current MDE practices can be applied in the elearning domain. (iv) The languages proposed provide mechanisms with which to improve the features of current authoring tools as regards modularization, sequencing and gamification. (v) An exploration of the benefits of using models in the domain of the course creation.

Finally, we would like to remark that the work here presented is the baseline of INDIe,[1] a three-year Erasmus+ Programme of the European Union whose aim is to develop a platform for the creation, publication and sharing of digital content for secondary education students.

[1]*Interactive Digital Content Platform to Share, Reuse and Innovate in the Classroom*, 2018-1-ES01-KA201-050924

Structure. The paper is organized as follows. Section II introduces the required background. Section III first presents the domain analysis carried out, after which the requirements elicited are described. This section concludes with a brief explanation of how a course is organized, and describe a previously applied manual process. Section IV presents an overview of the INDIeAuthor architecture, while Section V describes the metamodel and notation for the 4 DSL defined. Section VI explains the main issues regarding code generation, and Section VII shows the results of the evaluation performed. Section VIII provides a discussion of the related work, while Section IX shows our conclusions and further work.

II. BACKGROUND

This section introduces the background required for a better understanding of the course authoring approach presented in this paper. We first briefly describe the architecture of the UPCTforma *Infrastructure* on which INDIeAuthor was built, after which some basic concepts regarding DSLs and MDE are presented, as our work involves the metamodeling-based creation of DSLs.

A. UPCTforma INFRASTRUCTURE

As shown in Figure 1, the *UPCTforma infrastructure* is composed of a set of highly decoupled software components that are organized in three layers: *Technical Services*, *Content Services* and *Learning Analysis*. As indicated above, a detailed description of the architecture of the UPCTforma infrastructure is presented in [2].

The *Technical Services* layer includes the *Interoperability* and *Management* components. Interoperability is achieved using the IMS LTI standard. Content created with a UPCTforma can be linked to any LTI-compliant learning platform and content created with different UPCTforma components can be integrated. The Management component is a front-end that is in charge of the user control access and the communication between UPCTforma and external platforms and tools.

The *Content Services* layer includes components that provide services related to the creation of content, namely: video production (*UPCTmedia* component), creation of gamification activities (*UPCTplay* component), learner motivation management (*UPCTmotiva* component), and content deployment (*UPCTdeploy* component).

UPCTmedia component allows the creation, storage, publication and visualization either in streaming or on demand of multimedia content. The videos generated can be integrated into a course, linked to an elearning platform using the LTI standard or published in open mode. *UPCTplay* component includes a repository of games that can be integrated into a course. Two group games are currently available as described in [2]. INDIeAuthor courses are published in the UPCTforma infrastructure and the *UPCTdeploy* component is in charge of its deployment.

Learning analytics can be applied to any content produced with UPCTforma. This capability is supported by the *Learning Analysis* layer that includes the *Tracking* and *Learning Event Analyzer* components. The Tracking component captures, labels and stores the events occurring when learners interact with content. It uses the IMS Caliper standard, which is based on LTI. The *Learning Event Analyzer* component performs time-real processing of the Caliper events to obtain the information that is required for the learning analysis and motivation. This processing must be implemented for each course. The information obtained is stored in a database in order to be accessed from data analysis and visualization tools.

Creators can specify what conditions must be met to send bot-based motivation messages. The *Event Analyzer* component checks these conditions as it processes Caliper events and sends a notification to *UPCTmotiva* component whenever a condition is met. The *UPCTmotiva* component then generates and sends the corresponding bot message, as explained in detail in [2].

UPCTforma is currently based on the LTI and Caliper standards, but other interoperability and tracking standards could be supported because its architecture includes extension points for this purpose.

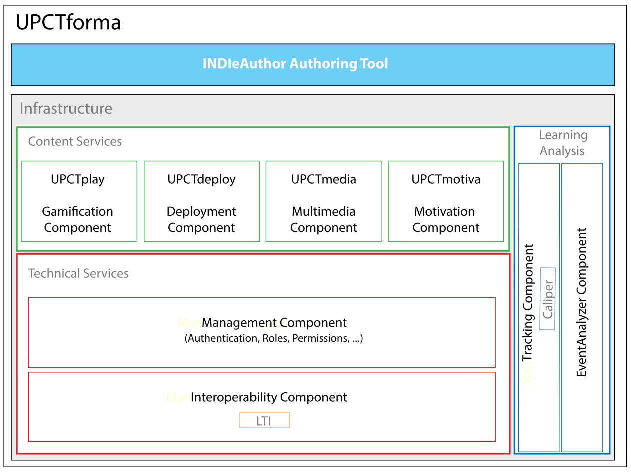


FIGURE 1. UPCTforma architecture.

B. DOMAIN-SPECIFIC LANGUAGES

Domain-specific languages (DSL) offer constructs and notation tailored to solve problems in a particular application domain. They make it possible to achieve a higher productivity than when using general-purpose languages (GPL), because they provide a higher level of abstraction. DSLs have been used from the early days of programming, but academic and industrial interest in DSL has increased significantly over the last 10 years. This has been mainly motivated by the emergence of Model-Driven Software Engineering (MDSE or simply MDE) [3]. MDE encompasses several software development paradigms, one of which is *Language-driven development* (a.k.a. *Domain-Specific Development*) that is based on using DSLs to automate the construction of new applications [11], [12]. In this kind of MDE solutions, DSLs are used to create models that are the input employed to code generators. The term "Modeling language" is also frequently used to refer to DSLs.

A formal definition of a DSL consists of three basic elements: *abstract syntax*, *concrete syntax*, and *semantics,voelter-dsl,cabot2012*. The abstract syntax specifies the language concepts and the relationships among them. The concrete syntax defines the notation used to create models. A textual or graphical notation is normally used. In the case of textual DSLs, the terms "script" or "program" are also used to refer to DSL models. Finally, the semantics establishes the meaning of the DSL models. A translational semantics is normally applied. Model-to-model (m2m) and model-to-text (m2t) transformations are used to translate models into the code of a language whose semantics is well defined (i.e., a GPL). We used only m2t transformations, which are discussed in Section VI. These transformations are normally written with template languages, which are easy to learn and use.

In MDE, *Metamodeling* is applied as a foundation on which to build DSLs. A metamodel formally describes the structure of models. A model is an instance of a metamodel, and the term *conforms-to* is normally used to express the *instance-of* relationship between a model and its metamodel. Object-oriented meta-modeling languages, such as Ecore [13], are normally used to define metamodels. These languages provide the object-oriented basic concepts traditionally used to create domain or conceptual models: classes, inheritance, aggregation, and references. Metamodels are, therefore, the core element around which DSLs are defined: The abstract syntax is expressed as a metamodel, the notation describes how the metamodel elements are rendered, and semantics is defined as a mapping between the DSL metamodel and other representations (i.e., GPL or HTML code). The separation between abstract and concrete syntax is a key characteristic of metamodeling-based DSLs.

DSLs have traditionally been built from scratch, but tools that automate their development have recently appeared. These tools are commonly called "DSL workbenches". Some widely used workbenches are Xtext and MPS for textual DSLs, and Sirius and Metaedit for graphical DSLs. Given the DSL metamodel and the concrete syntax definition, these tools are able to automatically generate the DSL editor. Most textual DSL workbenches also generate the *model injector*, which takes a DSL script as input and generates the corresponding model conforming to the DSL metamodel.

A DSL family is a set of related DSLs whose aim is to implement software solutions for a particular domain [6], [14]. Each member of the family addresses a different aspect or viewpoint of the solution. A solution is, therefore, formed of models (or scripts) for the different viewpoints, and these models are connected by means of relationships among their elements. Two common kinds of relationships among models of different DSLs are the following [6]:

- *Referencing*. A model expressed with one language L1 could reference elements of models expressed with another language L2. This requires: (i) the metamodel of the language L1 to import the metamodel of the language L2, and (ii) the concrete syntax of the language L2 to provide an importation mechanism.
- *Modularization*. An existing model could be reused in the creation of new models of a language. The concrete syntax should, therefore, support a namespace or package notion.

Two kinds of compositions are consequently required: language composition and model composition [6]. Languages should be composed so as to create a solution by combining models of different languages, which involves element referencing. Models should be composed do as to modularize models of a language. These two kinds of composition have been applied when building INDIeAuthor.

III. DOMAIN ANALYSIS AND REQUIREMENT ELICITATION

Once the UPCTforma infrastructure had been developed, we tackled the construction of INDIeAuthor. In order to achieve a better understanding of the authoring tool's domain and elicit the requirements, we carried out a domain analysis. The results of this analysis are explicitly presented in form of feature diagrams [15]. In this section, we first present the domain analysis performed, and we then describe the requirements of our tool. We also briefly describe how a course is organized, and the course development process applied while INDIeAuthor was being built.

A. DOMAIN ANALYSIS

An *elearning authoring tool* (or simply *authoring tool*) is a software application that is specifically created to facilitate the production of elearning courses. The feature of "asynchronous elearning (non live)" is also discussed in [16]. A *course* is organized in modules (e.g. lessons or units) that include "learning activities". Learners must complete learning activities to achieve the expected learning objectives. A learning activity ranges from simple drag and drop activities to gamification activities with which to engage students in the classroom. The use of mobile devices now allows the predominant use of authoring tools that can deliver content in HTML5 format [16], [17]. A list of 67 top tools can be found in [18]. These tools are mainly used to create content for companies, and their penetration into universities is very limited.

Elearning courses can also be created by means of general-purpose content creation tools (e.g. Flash and PowerPoint), Web authoring tools (e.g. Dreamweaver) or manual coding with software languages (e.g. Javascript and HTML5). However, authoring tools are prevailing over these alternatives because they provide two significant benefits: higher productivity and the fact that non-programmer users can create their own courses.

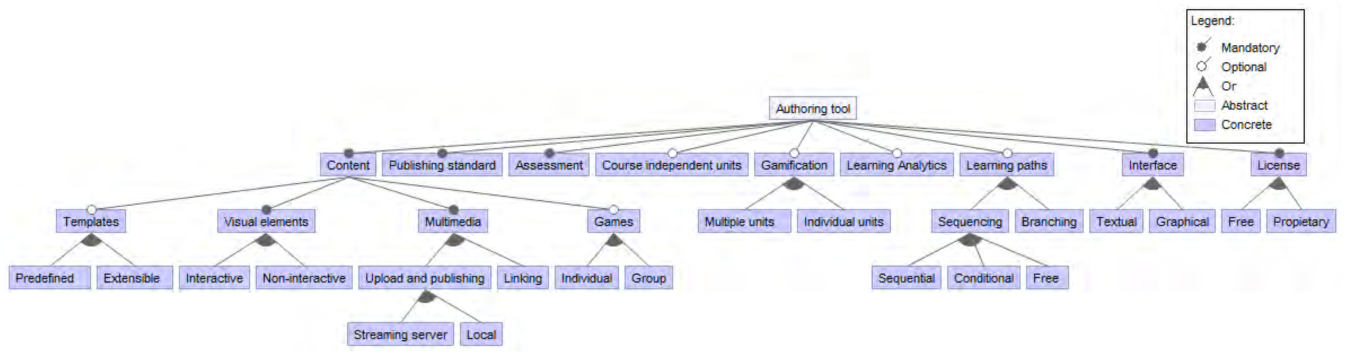


FIGURE 2. Feature model for authoring tools.

There are many authoring tools, which differ in the features provided. The number of features and the variability in the form of their support makes it difficult to classify them.

We have organized our domain analysis into three stages. Firstly, we have studied several sources of information on elearning authoring tools [16], [18], [19]. This information was then used to define a feature model in order to systematically organize the set of relevant features representing the variability of the domain. This framework has been used to evaluate 8 widely used authoring tools.

Figure 2 shows the feature diagram proposed. *Course-independent units* and *Learning analytics* features will be expanded in separated diagrams. The diagrams for the *Publish* and *Assessment* features will not be shown owing to space limitations. We comment on each of the features in the feature model as follows.

1) CONTENT FEATURE

Courses are commonly organized in units (or lessons) formed of sections. Sections may show different kinds of content. All the tools provide typical widgets or control elements in Web user interfaces. Video content is also supported by most of the tools. Games are another kind of content that increasingly provide more authoring tools.

Videos can be uploaded/published or accessed through a link. An external streaming server and/or local server can be used as a video-on-demand platform. *Games* are stored in a repository, and can be for individuals or groups.

Templates provide a content reuse mechanism. A template establishes an arrangement of content, including placeholders. When a template is used, the creator must replace placeholders with concrete content. Templates can be part of a library provided by the tool or can be defined by content creators.

Widgets, videos or games can be used to create learning activities (e.g., complete a drag and drop activity, play a video, or play a game of hangman). Each activity involves one or more actions to be performed by learners. Scores can be assigned to individual actions or activities to measure the student's learning progress.

2) PUBLISH STANDARDS FEATURE

Once a course is created, it must be published to be accessible from LMS or MOOC elearning platforms. There are two options by which to publish a course: (i) it can be packaged and uploaded onto an elearning platform, or (ii) it can be published on the platforms supporting authoring tools (e.g. the UPCTforma environment) and linked from an elearning platform. Several standards with which to package content are available (SCORM, xAPI, cmi5, IMS CC, IMS CP and AICC), and IMS LTI is the standard used for remote access.

3) COURSE-INDEPENDENT UNIT FEATURE

This feature refers to the possibility of creating units that are independent of a particular course. This would allow the reuse of the same unit on different courses. A course would, therefore, be created by combining (i.e., importing) several separated units. We have identified two points of variation in the combination of units.

- *Importation*: The format to express how a unit is imported can be proprietary or standard, and the units can be imported in the form of a link or content. If a unit is imported as a link, the changes made to it are automatically reflected on all the courses that import it. Otherwise, an importation of content involves the existence of several copies of the same content, and content changes will require the modification of each existing copy.
- *Delivery*: A course could be delivered to users in the form of a single package or link. This feature is distinguished from the ''Publish standards'' feature because the tools can provide a different support for the publishing of courses created by importing units

4) LEARNING PATHS FEATURE

Learning paths establish the order in which units or a unit's contents can be accessed in the learning process. By completing a learning path, learners achieve one or more learning objectives. The term *Sequencing* is used to denote a learning path through the units of a course, while the term *Branching* refers to sequence contents included in the same unit.

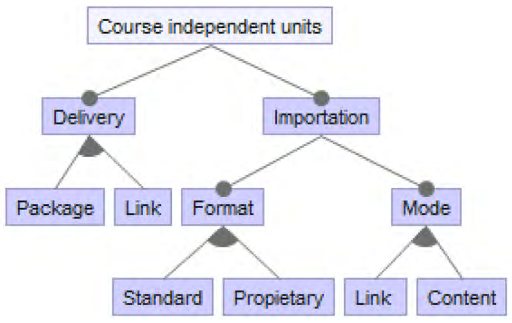


FIGURE 3. Multiple units feature for authoring tools.

Sequencing can be limited to express a sequential order for units or a rule-based conditional access could be provided by creators. In a conditional access, conditional expressions are associated with units to check a learner's level of achievement (e.g. if the score is greater than a particular value). Conditions can be expressed in different forms of rules. The absence of a sequencing means that learners can freely choose the order in which to complete the units.

5) ASSESSMENT FEATURE

Assessment or evaluation is a feature supported by all the authoring tools, but in different ways. All the tools offer the typical types of questions as a single answer, multiple answers or a filling answer. We have distinguished four main points of variation as: (i) whether the assessment can be performed in a training mode; (ii) the kind of feedback given to learners for correct and incorrect answers; (iii) what choices about questions can be randomized, e.g. selection and the ordering; (iv) whether the questions can be defined only by teachers or they can be selected from a question bank that can be updated by teachers.

6) LEARNING ANALYTICS FEATURE

Learning analytics can be defined as "the measurement, collection, analysis and reporting of data about learners and their contexts, for purposes of understanding and optimizing learning and the environments in which it occurs" [20]. Learning analytics, therefore, involves three aspects: tracking, event analysis and visualization. We have considered variations for each of them.

- *Tracking*. The features of tracking are: the event source, the storage format and the storage platform. An event can be recorded for user interactions with any kind of content: widgets, multimedia elements, units and courses. Events to be recorded can be predefined or a tool can allow the elements whose interactions are registered to be chosen. The storage format can be private or based on standards, such as SCORM, xAPI and Caliper. The tracking generated can be stored on the elearning platform on which the course is deployed or in private or public learning record stores (LRS).
- *Event analysis*. The event analysis can be predefined or the creators can provide a mechanism to implement the desired analysis.
- *Visualization* A predefined analysis involves a set of predefined dashboards. Conversely, dashboards could be chosen if a course-specific analysis is implemented.

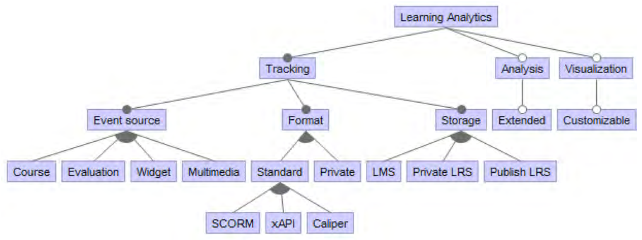


FIGURE 4. Learning analytics feature for authoring tools.

7) GAMIFICATION FEATURE

Units can be gamified by means of elements such as points, badges, missions or ranking. For example, a ranking could be based on the points obtained in the different units of a course, or missions (extra units) can be proposed to the student.

User Interface Feature: Three kinds of user interfaces can be considered: graphical, textual or hybrid.

License: We have distinguished free and proprietary licenses.

B. REQUIREMENTS FOR INDIeAuthor

In this sub-section we describe the functionality provided for INDIeAuthor. The configuration of the proposed feature model provides a list of features. We have also considered the following high-level quality attributes: interoperability, reusability, maintainability, portability, and adaptability. The ADL report states that some of these attributes are essential features for learning environments [19]. We show the choices for each point of variation of the top-level features below, while in section VII, we discuss how the quality attributes mentioned have been achieved.

1) CONTENT ELEMENTS

In addition to typical widgets, video and games will be supported by the UPCTmedia and UPCTPlay components of the UPCTforma infrastructure. A template mechanism is provided that allows creators to define their own templates and use predefined templates. Learning activities can be created, and a score can be assigned to each activity.

A content unit is formed of one or more sections. Each section begins with a background image and a title, and is composed of rows and columns. A column can contain visual elements (widgets, video and games) or recursively nested rows and columns. A section is completed when all its learning activities have been completed. Students must perform one or more actions to complete an activity. When an action is performed, a widget (e.g. a tick) is activated to indicate this. A content unit is completed when all its sections have been completed. The learning progress in a section is shown as the percentage of actions completed.

Some of the widgets provided are *ImageTextOver* (a hidden text appears in an image when the learner clicks onto it), *VerticalAccordion* and *HorizontalTabs* (a vertical or horizontal arrangement of tabs; the learner selects which content to view by clicking a tab), *AnimationInOut* (animation on a graphic provided by the creator, the learner has to click onto the widget), *RectangleDragAndDrop* (drag and drop interaction in which learners must correctly associate concepts and correct definitions)

A bank of templates has been created with different means of organizing content, such as: a row with a single column that contains a VerticalAccordion widget (*OneColumnVerticalAccordion*), or a RectangleDragAndDrop widget (*OneColumnRectangleDragAndDrop*).

2) PUBLISH STANDARDS

The option chosen was remote access through LTI links because UPCTforma is LTI-compliant. The content created is published in the UPCTforma deployment component, thus allowing content to be linked from LTI-compliant e-learning platforms and applications.

3) COURSE-INDEPENDENT UNITS

This feature is supported. Units are delivered as LTI interoperable links. They are imported through these links, and no copies are made. Units must be configured as accessible to or hidden from the student at the beginning of the course.

4) LEARNING PATHS

At this moment, sequencing but not branching is provided. The content of a unit can be followed in any order. When a course is produced, its creators will be able to choose the kind of sequencing: Sequential, conditional and free access. We have considered four types of conditional access: *strong*, *inhibitor*, *weak*, and *score*. With the exception of *score*, these types were taken from [21]. When conditional access is chosen, each unit should be labeled with one of these types. When a learner starts a course, all the units are closed except those that the creator indicates should initially be open. The types of access determine how the closed units will be opened. Units labeled as *strong* or *inhibitor* indicate which units will be opened or closed, respectively, when they are completed. The *Score* label is used to indicate which units will be opened depending on the score obtained. Finally, a unit can be labeled as *Weak* to indicate that one or more units will be opened only when it is opened.

5) ASSESSMENT

The tool should allow the creation of assessment units separated from content units. Training and evaluable units will be provided. Final units will have only one attempt. It will be possible to specify the number of attempts and the type of grade (the highest, the lowest or the average) in training units. We have considered four question types: SingleAnswer, MultipleAnswer, FillingAnswer, and TrueOrFalse. Feedback will be provided for each correct or incorrect answer. It is possible to define a bank of questions. The questions in an evaluation unit can be defined by teachers or randomly chosen from the bank.

6) GAMIFICATION

The tool makes it possible to gamify a complete course or individual units. Typical gamification elements are, therefore, provided for this purpose: points, badges, missions and ranking. Creators can assign badges or points for (i) the completion of a content unit, (ii) the level of achievement of learning activities, and (iii) the completion of an evaluation unit depending on the score obtained. For each course, a ranking is included to sort students according to the points obtained in all the units. According to the obtained points, missions or extra units can be proposed to the student.

7) LEARNING ANALYTICS

Tracking data are recorded in any kind of content: courses and units (e.g. objectives achieved or time spent), evaluation units (e.g. scores or responses), widgets (e.g. the answers in a drag&drop), video (e.g. time taken to watch a video), and games (e.g. points or time). Specifying events to be recorded for each kind of element is a very time consuming task and we have, therefore, preferred to record all the events. The tracked events can be stored on elearning platforms or in LRS (public or private). Predefined and customized analysis and visualization can be performed. This is not addressed in this paper, but a detailed explanation for a gamification activity is provided in [2].

8) LICENSE

INDIeAuthor will be available as freeware software.

C. COURSE PRODUCTION MANUAL PROCESS

Prior to developing the authoring tool presented here, courses were manually implemented by Web developers. The CPDC/UPCT defined a content production process that is explained in detail in [22]. This process consisted of four stages.

1) PowerPoint is used to write a script that describes the course content by using the templates and widgets available. The teacher and a scriptwriter interact to create the script. Each template and widget has an identifier that allows it to be referenced.
2) the script is provided to the web developers. They adapt the template code to each particular use, among other development tasks. The scriptwriter tests that the content developed conforms to the script. Once the scriptwriter states that no errors have been detected, the stage ends with the publication of the content.
3) the content published is reviewed by the teacher who informs the scriptwriter of the changes to be made. This phase ends when the content is validated by the teacher and the scriptwriter.

4) a programmer specialized in interoperability adds the code for the required standard: LTI or SCORM. The URL for LTI or the SCORM file is provided to the teacher. More stages are necessary if translation to another language is required.

In next section, we show how this manual process was automated in INDIeAuthor using a language engineering approach.

IV. OVERVIEW OF THE INDIeAuthor ARCHITECTURE

As Figure 5 shows, INDIeAuthor consists of three elements: (i) textual editors for the four DSL defined, (ii) a code generator that integrates the DSL execution engines, and (iii) a framework that is in charge of the interaction with the UPCTforma infrastructure: UPCTmedia, Tracking and Interoperability components. These three elements constitute a model-based generative architecture. These architectures are the result of applying a domain specific development (a.k.a. language-driven engineering) [11], [12].

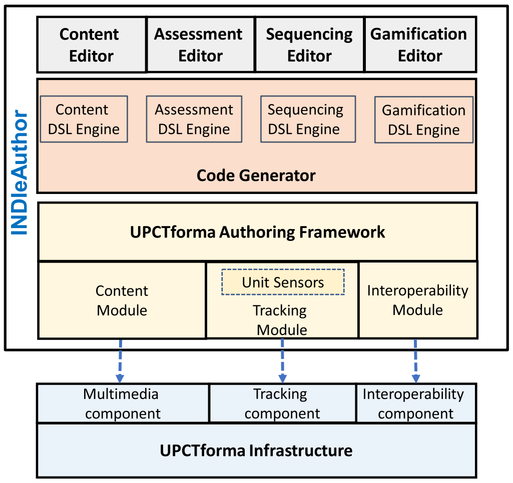


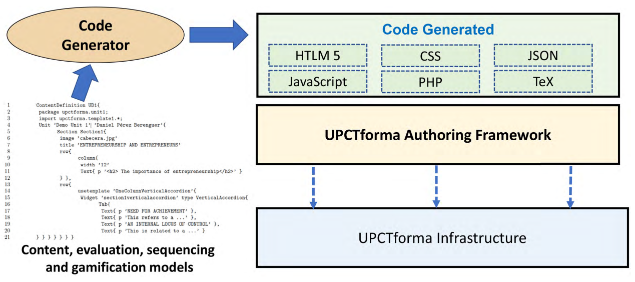
FIGURE 5. **UPCTforma authoring tool.**

The INDIeAuthor framework provides the functionality common to all the courses. With regard to learning data tracking, several Caliper sensors must be implemented for each course: a sensor for each unit and another for the course itself. We have used several Caliper profiles to represent the tracking data collected (entities, events and actions involved in learning interactions).

The generative architecture built is based on a family of four DSLs which make it possible to specify the main aspects or viewpoints of a course.

- *Content* DSL to define content units,
- *Assessment* DSL to create assessment units,
- *Gamification* DSL to gamify content, and
- *Sequencing* DSL to specify the order in which units will be presented to learners.

A textual editor is provided for each DSL. Scripts created with these editors are converted into models by means of a model injector. The DSL editor and model injectors have been


Code Generator
Code Generated
HTLM 5
CSS
JSON
JavaScript
PHP
TeX
UPCTforma Authoring Framework
UPCTforma Infrastructure
Content, evaluation, sequencing and gamification models

```
ContentDefinition UD1{
 package upctforma.unit1;
 import upctforma.template1.*;
 Unit 'Demo Unit 1'| 'Daniel Pérez Berenguer'{
       Section Section1{
        image 'cabecera.jpg'
        title 'ENTREPRENEURSHIP AND ENTREPRENEURS'
        row{
              column{
               width '12'
               Text{ p '<h2> The importance of entrepreneurship</h2>' }
        } },
        row{
              usetemplate 'OneColumnVerticalAccordion'{
               Widget 'section1verticalaccordion' type VerticalAccordion{
                     Tab{
                      Text{ p 'NEED FOR ACHIEVEMENT' },
                      Text{ p 'This refers to a ...' },
                      Text{ p 'AN INTERNAL LOCUS OF CONTROL' },
                      Text{ p 'This is related to a ...' }
 } } } } } } }
```

FIGURE 6. **UPCTforma authoring tool architecture.**

generated by the Xtext tool used to build the DSLs. Figure 6 illustrates how the generative architecture that implements INDIeAuthor works. Once a teacher has created the scripts (Content, Assessment, Gamification, and Sequencing) for a particular course, he/she should use the *Code Generator* tool to automatically generate the code (HTML, CSS, JSON, JavaScript and PHP files) that implements the course in the UPCTforma Authoring framework (*framework completion* technique).

The Code Generator implements a workflow that executes the DSL engines developed for each DSL, as shown in Figure 5. These DSL engines have been implemented as model-to-text transformations. These transformation will be explained later in Section VI.

V. DSLS IN INDIeAuthor: METAMODEL AND NOTATION

In this section, we present the metamodel and notation of the four DSLs defined for INDIeAuthor. The concepts and relationships of each metamodel will be described as the notation is introduced.

A. A DSL FOR CREATING CONTENT UNITS

Figures 7 and 8 show the metamodel defined for the *Content* DSL. A content definition can include zero or one content unit or zero or more template and type definitions. The definitions of templates and types should be declared separately from those of units in order to favor reusability.

A template and several types are defined in the script shown in Figure 10, and a content unit definition script is shown in Figure 9. Each content definition must be part of a package. Typical conventions of package naming can be followed. A qualified name of a unit, template or type is formed of its name preceded by the package name. The package notion is used to achieve model composition (i.e., modularization), as described in Section II-B. We shall now use these two scripts to describe the concepts and notation of the Content DSL.

1) SIMPLE AND COMPOSITE TYPES

A type can be simple (or primitive), composite or widget. There are five simple types: *Text*, *Image*, *Video*, *Game*, *String* and *Any*. A text value is formed of one or more *Paragraphs*. Video and game values are specified by means of the ID of

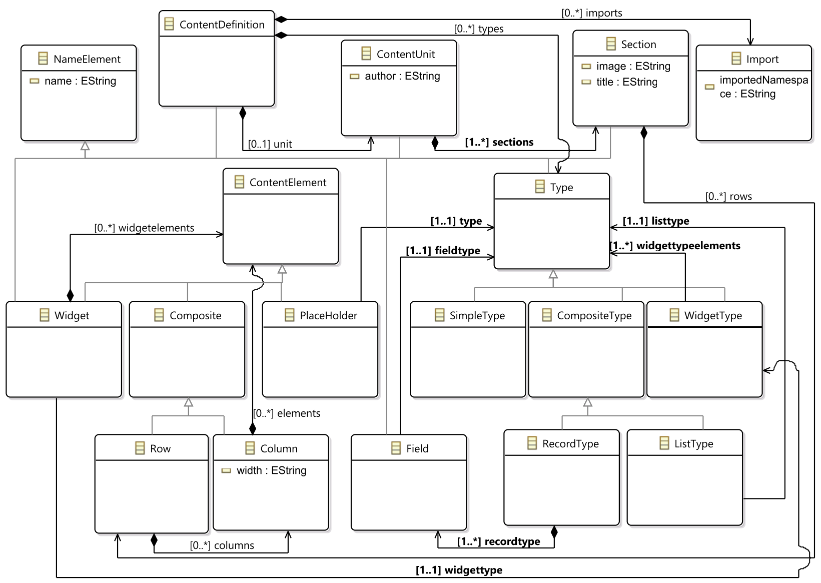


FIGURE 7. Content DSL metamodel.

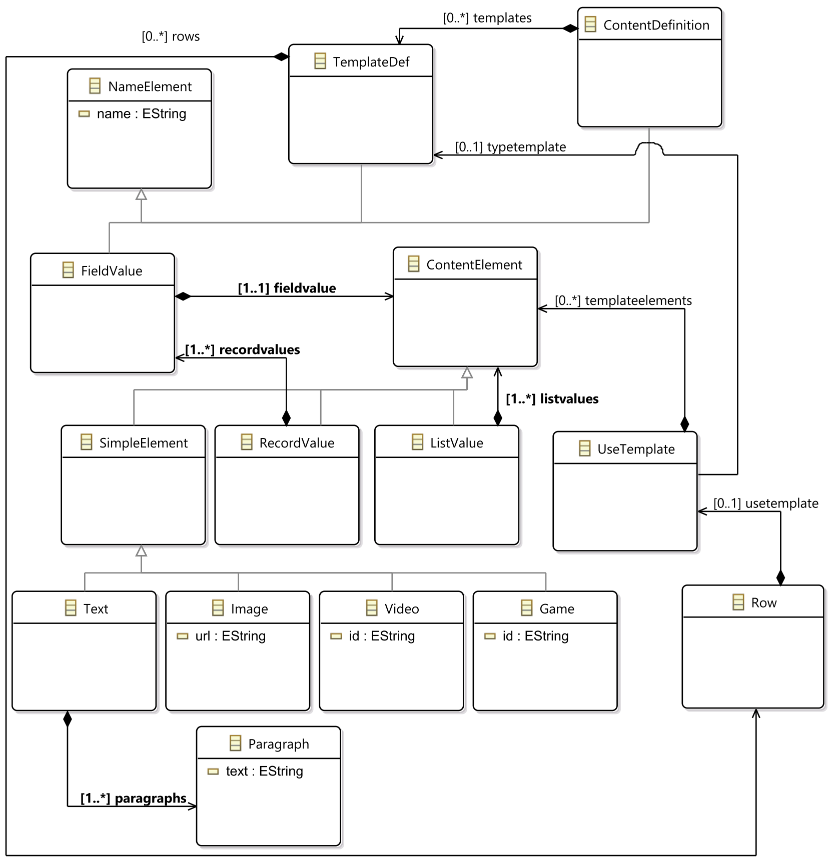


FIGURE 8. Content DSL metamodel (Templates and values).

a UPCTmedia video or a UPCTplay game, respectively. An image value is also specified using the URL that indicates its location. The *Any* type is used to indicate that a variable can have a value of any existing type.

A composite type can be a *record* or a *list*. A record is a set of pairs (or fields), and each pair is defined by its name and a type. A list is a sequence of values of the same type. Records and a list can be combined to form complex structures.

A record value is constructed by giving a value to each field. A list value is constructed by enclosing the list of values in curly brackets. The *DemoTemplate1* content definition (see Figure 10) includes examples of record and list type declarations. The *Tab* type is a record of two fields: *name* of text type and *content* of Any type; The *TabList* type is a list of *Tab*s; and the *Animation* type is a record of three fields: *size* of text type, *background* of image type, and *images* whose type is a list of images.

2) WIDGET TYPE

A widget type is declared by enumerating the types of the elements that form the widget element. For example, the *ImageTextOver* widget type is declared as an enumeration of two types: Image and Text, as shown in Figure 10 (line 14). Lists make it possible to specify that a widget contains an arbitrary number of elements of a particular type. For example, a *VerticalAccordion* can be formed of a variable number of Tabs (Figure 10, line 6). Types of widgets provided by a particular Web platform are declared in a pre-built content unit definition. In our example, *DemoTemplate1* declares the widget types used in the example of content unit definition, but these declarations would be part of the content definition including all the widget types of the Web platform used to give the course.

A widget value is constructed by indicating the widget name followed by the widget type's name, and the list of values enclosed in curly brackets. This list will include a value for each type in the widget declaration, as illustrated by the value `section1verticalaccordion` of VerticalAccordion type in the *DemoUnit1* script (Figure 9, line 16-23). Note that a name is given to each widget added to a course. This allows each widget to be associated with its tracking. Widget values can be used in column statements and template declarations, as explained below.

3) UNIT AND SECTION DEFINITION

A *content unit* definition aggregates one or more sections, and each *section* organizes its content in a set of rows formed of columns. Units and sections are identified by a *name*. A unit can also specify the name of its *author*, and a section has a *title* and a *background image*. Figure 9 shows the content unit definition, denominated as 'DemoUnit1', which has been authored by 'Daniel Perez' (line 4). This unit has only one section, entitled 'Section 1', whose background image is the file named "cabecera.jpg" (line 6) and whose title is "ENTREPRENEURSHIP" (line 7).

Rows and *Columns* are recursively composed, as a column can also contain other rows. A row declaration includes one or more column statements. A column statement specifies the associated element, which can be a simple value, a widget value, or a row. A row can be created from scratch or by re-using a template, as illustrated by the two rows declared in the 'Section 1' section (lines 8-23). The first row declaration specifies a row that is created from scratch (lines 8-13). It has only one column that contains a text value. The second row declaration uses the *OneColumnVerticalAccordion* template to create a vertical accordion that has only one tab (lines 14-23). We explain how the templates are declared and used below.

```
 1  ContentDefinition DemoUnit1{
 2   package upctforma.unit1;
 3   import upctforma.template1.*;
 4   Unit 'Demo Unit 1' 'Daniel Perez Berenguer'{
 5    Section Section1{
 6     image 'cabecera.jpg'
 7     title 'ENTREPRENEURSHIP AND ENTREPRENEURS'
 8     row{
 9      column{
10       width '12'
11       Text{ p '<h2> The importance of ...</h2>'}
12      }
13     },
14     row{
15      usetemplate 'OneColumnVerticalAccordion'{
16       Widget 'section1verticalaccordion': VerticalAccordion{
17        [
18         {
19          name: Text{ p 'NEED FOR ACHIEVEMENT'},
20          content: Video{ id 'Nzk='}
21         }
22        ]
23  } } } } } }
```

FIGURE 9. Example of unit definition.

4) TEMPLATES

Reusable content can be defined in the form of templates. A template can be reused to define new units or templates. A template specifies a content structure by indicating placeholders rather than concrete elements. In a template declaration, placeholders are expressed as the type of the expected element (Simple or Widget). When using a template, it is necessary to replace each of its placeholders with a content element of the expected type.

The *OneColumnVerticalAccordion* template is defined in the script shown in Figure 10 (lines 16 to 22). The template arranges content in a single row with a single column that expects a VerticalAccordion widget (i.e., a placeholder).

The use of a template is expressed by means of the *usetemplate* keyword followed by the template name, and a list of values enclosed in brackets. The type of each value in the list must be identical to the type associated with the corresponding placeholder. The 'Demo Unit 1' unit (Figure 9) shows an example of the use of a template in the declaration of the second row (lines 15-23). A VerticalAccordion type value replaces the single placeholder in the template declaration. When using a template, it is necessary to *import* the template definition. 'DemoUnit1' imports the *OneColumnVerticalAccordion* template (line 3). A content definition can also import types. These importations are an example of model composition.

```
ContentDefinition DemoTemplate1{
  package upctforma.template1;
  types{
    Tab{ name: TextType, content: Any },
    TabList{ list_of Tab },
    widget VerticalAccordion{ TabList }
    ImageList{ list_of ImageType},
    Animation{
      size: TextType,
      background: ImageType,
      images: ImageList
    },
    widget AnimationInOut{ Animation },
    widget ImageTextOver{ ImageType, TextType}
  }
  templates{
    TemplateDef OneColumnVerticalAccordion{
      row{
        column{
          width '12'
          Placeholder VerticalAccordion
} } } } }
```

FIGURE 10. **Example of template definition.**

B. A DSL WITH WHICH TO CREATING ASSESSMENTS

Figure 11 shows the metamodel defined for the *Assessment* DSL. An *Assessment* aggregates a set of *assessment unit* definitions that can be of two kinds: *Training* and *Final*. A training unit specifies the *number of attempts* and the *type of grade* that will be stored (the highest, the lowest or the average). Final units have only one attempt. Both kinds of units aggregate *Questions* of different types: *SingleAnswer*, *MultipleAnswer*, *FillingAnswer*, and *TrueOrFalse*. A single answer question is defined by a statement (i.e., a Paragraph), a set of *Single* answers, and the number of the correct answer. A multiple answer question is defined by a statement (Paragraph), a set of *Multiple* answers. A Multiple answer has a boolean attribute that indicates whether it is true or false. A filling answer question is defined by a set of *Hole*s. A hole has an attribute that indicates if the associated text is hidden or visible. A true or false question is defined by a set of *Assertions*. Each assertion has an attribute that indicates whether it is true or false. Each question has a different feedback depending on whether the answer is correct or incorrect.

Figure 12 shows an example of an assessment that aggregates an assessment unit definition denominated as 'Evaluation1'. This assessment unit is final and aggregates a set of questions. The first question declared is of the 'single answer' type (lines 5-16). An assessment unit also declares the number of questions that are shown in each attempt. This number is 10 in the 'Evaluation1' unit (line 3).

C. A DSL FOR SEQUENCING UNITS

Figure 14 shows the metamodel created for the *Sequencing* DSL. A Sequencing definition determines the order in which the units can be followed in the learning process. It consists of a set of *unit flows* (*SequencingUnit* metaclass). A unit flow must be defined for each content and assessment unit. A unit flow expresses the initial status of the associated unit and which units are open when the unit is completed. The initial status can be *open* or *closed*. A *closed* status restricts access to the completion of other units. Free access is achieved with 'open' status for all units. Conditional learning paths can be expressed by means of conditions that are assigned to units. We have considered the four kinds of conditions introduced in Section III: *strong*, *weak*, *inhibitor* or *scores*. Please note that the sequencing a course requires the referencing of all its content and assessment units. The Sequencing metamodel, therefore, imports elements from Content and Assessment metamodels, as shown in Figure 14. Moreover, a Sequencing script must import all the units from the sequenced course, as shown in the script in Figure 13. This script shows a unit flow for the 'Evaluation1' evaluation unit, which includes conditional flows to the 'Demo Unit 3' and 'Demo Unit 4' units. The initial state of 'Evaluation1' is closed.

D. A DSL FOR GAMIFYING UNITS

Figure 16 shows the metamodel created for the *Gamification* DSL. Content and Assessment units, along with widgets, can be gamified with badges, points and missions. A gamification definition declares the badges used and specifies the gamified units and widgets. This metamodel, therefore, imports ContentUnit and Widget elements from the Content metamodel, and AssesmentUnit elements from the Assessment metamodel.

A badge is defined by its name, description, and url. A gamified unit is defined by its URL and an image, and has one or more achievements of points, badges and missions. Points and badges can be awarded in three different ways: (i) when the unit is completed (*Completed* type), (ii) depending on the access frequency to the unit (*LoginDaily* type), or (iii) depending on the score obtained in the unit (*Scores* type). In the case of points, the number of attempts can be checked if the type is *Completed*, and a score interval can be applied if the type is *Score*. Widgets can only be gamified with points. The gamification of a widget can be applied to all or some of the units in which the widget is included (*widgetref* relationship between WidgetGamify and WidgetType in the metamodel). Missions (extra units) can be proposed to the students on the basis of points accumulated on a course.

Figure 15 shows an example of a gamification script that includes the following statements: the definition of a badge (lines 5 to 8), the gamification of a *RectangleDragAndDrop* widget with points (lines 10 to 13), and the gamification of the *Evaluation1* assessment unit (lines 15 to 30), including a mission proposed to the student based on the points obtained on the course (lines 26 to 30). The *completed* point type has been used to assign points according to the number of attempts required by the student to finish the RectangleDragAndDrop activity. The gamified unit declaration specifies that: (i) A 'Badge1' is achieved when the 'FinalUnit1' unit is completed, and (ii) the points achieved depending on the grade obtained (e.g., 10 points if the grade is between 40.00 and 49.99).

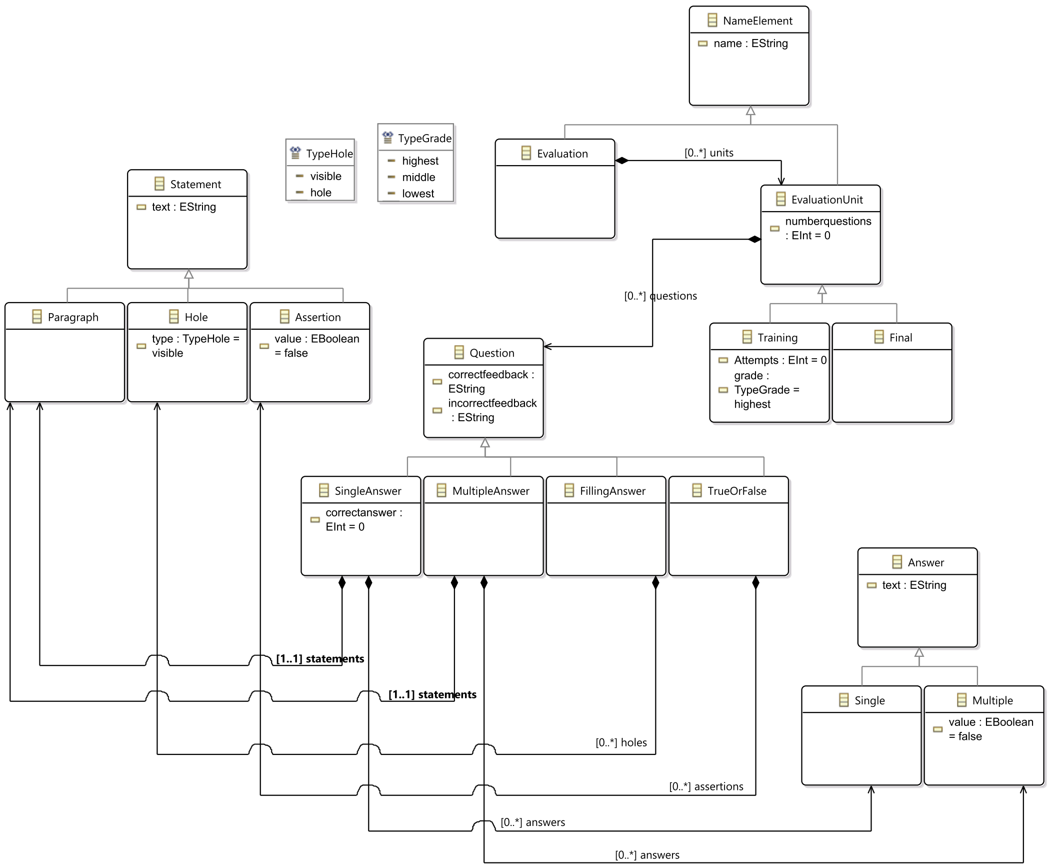


FIGURE 11. Evaluation abstract syntax metamodel.

```
Evaluation 'Evaluation'{
 Final 'Evaluation1'{
  numberquestions 10
  questions{
   SingleAnswer{
    statement 'An enterprise is:'
    answers{
     'An economic unit of production.',
     'A department of a company.',
     'A stakeholder.',
     %'The external functions develop by a company.'
    }
    correct 1
    correctfeedback 'Correct'
    incorrectfeedback 'Incorrect'
   },
   MultipleAnswer{
    ...
   }
   ... }}
 ... }
```

FIGURE 12. Example of assessment unit definition.

VI. CODE GENERATION FROM MODELS

A DSL engine was implemented for each INDIeAuthor DSL, as shown in Figure 17. As indicated in section IV, each of these engines was implemented as an m2t transformation.

```
Sequencing{
 import upctforma.template1.∗, upctforma.unit1.∗,
        upctformaevalua.evalua1.∗;
 unitFlow{
   state closed
   evaluationunit 'Evaluation1'
   flow{
     type scores
     InitialScore 00.00 FinalScore 65.00
     nextunit 'Demo Unit 3' },
   flow{
     type scores
     InitialScore 65.01 FinalScore 100.00
     nextunit 'Demo Unit 4'}}
  unitFlow {..}
}
```

FIGURE 13. An example of sequencing script.

These transformations take as input a DSL model and generate code that is part of the implementation of the course (e.g. HTML5 and JavaScript code). A workflow was implemented to integrate the execution of the DSL engines. The content and assessment scripts must be processed before the sequencing of the gamification scripts. This is because units must be available to express how they are ordered or gamified. In this section, we describe how each of the DSL engines (i.e., m2t transformations) works.

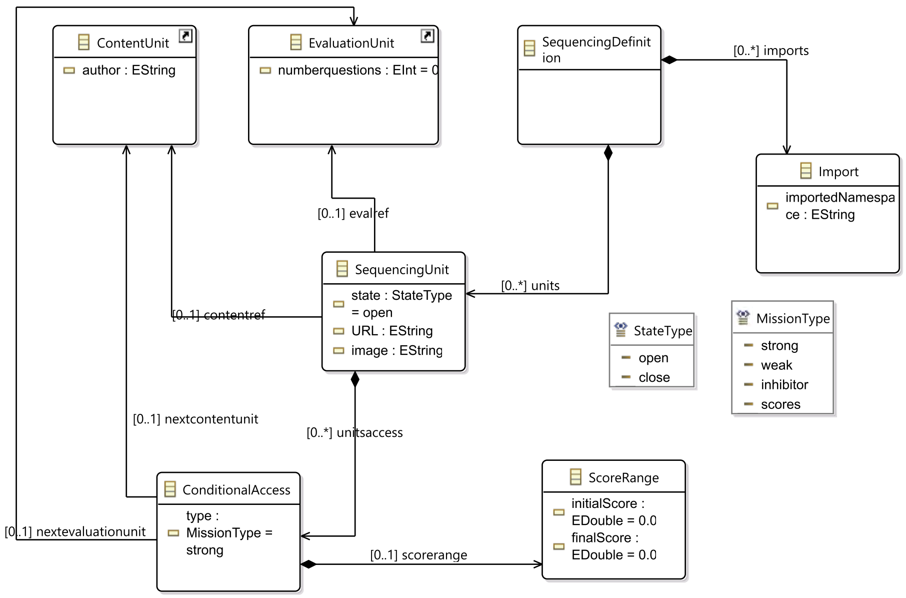


FIGURE 14. Sequencing abstract syntax metamodel.

```
Gamification{
 import upctforma.template1.*, upctforma.unit1.*,
        upctformaevalua.evalua1.*;
 badges{
  Badge Badge1 {
   description 'Badge obtained by final unit 1'
   url 'evaluation1.jpg'
 } }
 WidgetPoint{
  widget RectangleDragAndDrop
   Point{ type completed points 100 attempt 1 },
   Point{ type completed points 80 attempt 2 },
         Point{ type completed points 50}
 }
 UnitPoint{
   evaluationunit 'Evaluation1'
   URL 'https://server_url/evaluation1/index.php'
   image './images/evaluation1.png'
   BadgeUnit{ type completed Badge1 Evaluation1 }
   Point{ type scores points 10
     InitialScore 40.00 FinalScore 49.99 },
   Point{ type scores points 60
     InitialScore 50.00 FinalScore 79.99 },
   Point{ type scores points 100
     InitialScore 80.00 FinalScore 100.00 }
   missions{
    Mission{
     InitialPoint 80 FinalPoint 160
     extraunit 'Extra Unit 1'}
 } } }
```

FIGURE 15. Example of Gamification script.

A. CODE GENERATION FOR CONTENT DSL

Content models are independent of a particular software technology (e.g. a programming language). We have used HTML5, CSS, JavaScript and PHP to implement content units. The m2t transformation implemented is executed for each Content model, and a content unit is outputted in each execution. Sections aggregated in the *ContentUnit* element of a Content model are traversed. Firstly, a HTML5 section is created with a title and a background image. The rows in the section accessed are then traversed to generate the content to be included in the section. It is necessary to check whether or not a row is based on a template, and the placeholder values are retrieved if a template has been used. The columns in the row are then traversed to recover the associated content element. Code for each content element is generated, and placeholders are replaced with values if required. Note that LaTex mathematical expressions are supported as text elements. A JSON file is additionally created for each unit, which is intended to register the unit's sections and widgets. In this file, widgets are grouped by sections, and each of them has a unique identifier. The INDIeAuthor Framework uses this JSON file to show the percentage of widgets completed for each section. Whenever a widget is completed, the framework sends the unit progress to the elearning platform for it to be stored in the grade book. This is carried out by the Interoperability component. A PHP index file is also created with a header and a section menu.

B. GENERATION CODE FOR ASSESSMENT DSL

The INDIeAuthor Framework contains the code that manages assessment units. This code is parameterized by a JSON file that includes data on an assessment: number of questions, assessment type (training or final), question statements, responses, and the correct or incorrect feedback. The framework uses this information to deploy the assessment unit. The JSON file is generated by the m2t transformation implemented, which is executed for each Assessment model.

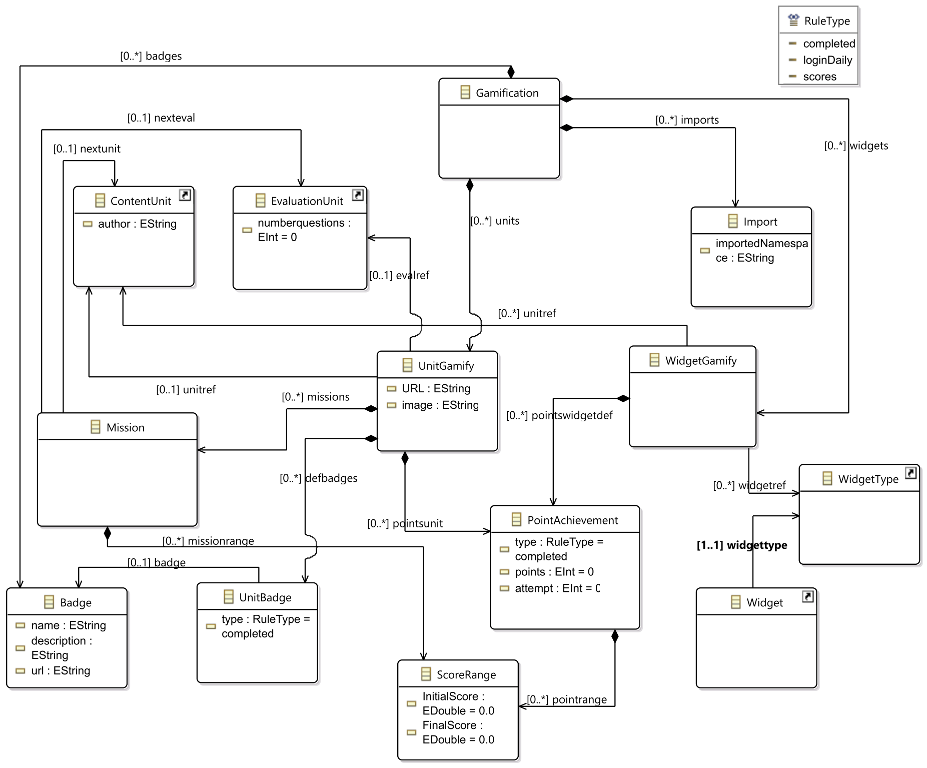


FIGURE 16. Gamification abstract syntax metamodel.

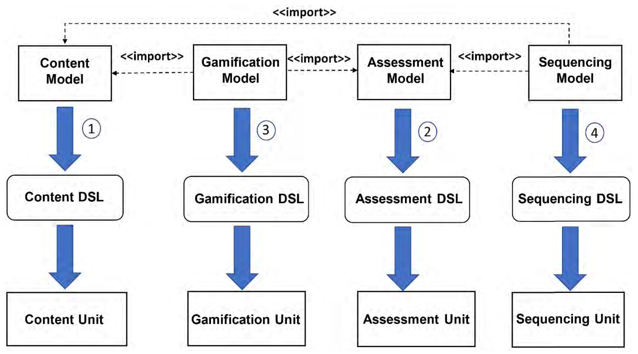


FIGURE 17. Code generation from UPCTforma models.

C. CODE GENERATION FOR GAMIFICATION DSL

The m2t transformation implemented is executed if a Gamification model exists. This transformation is organized in two steps. Firstly, gamified widgets are iterated and a Javascript object is generated for each of them. This object registers the points that are assigned depending on the number of attempts to complete the activity. As indicated previously, a widget gamification definition can be applied to specific widgets or all the units on a course. In the case of specific definitions, a map is used to associate each generated Javascript object with the units in which it is applicable.

Secondly, gamified units are iterated and another Javascript object is generated for each of them. These objects register points (by rank or attempts), badges and missions associated with each unit. This object will be used to assign badges and points during the user's interaction. In addition, missions will be proposed to students based on the points accumulated on the course. It should be noted that all units can be deployed in normal or gamified mode if there is a gamification definition.

D. CODE GENERATION FOR SEQUENCING DSL

The m2t transformation implemented is executed if a Sequencing model exists. Given an input sequencing model, the code that implements the course sequencing is generated. Unit flows are traversed and the access type specified for each flow is checked. The following code is also generated: (i) State checking calls to routines of the INDIeAuthor Framework; (ii) Code that shows an icon when a user signs into a unit whose state is open. (iii) Code that places a visual element (e.g. tick symbol) on the unit icon when a unit is completed. (iv) A PHP index file.

VII. EVALUATION

INDIeAuthor is a course authoring tool that provides its creators with a textual language that is formed of the four DSLs described previously. We have evaluated both the language and the tool. In this section, we present a case study carried out to evaluate the characteristics of the language. We also analyze the advantages of using MDE to implement the language. In Section VIII, INDIeAuthor will be contrasted with the tools evaluated in the domain analysis stage.

TABLE 1. User evaluation results.

Question	Strongly agree	Agree	NAND	Disagree	Strongly disagree
Is the language easy to learn?	45.45% (10/22)	27.27% (6/22)	22.73% (5/22)	4.55% (1/22)	0% (0/22)
Is the language easy to use?	72.72% (16/22)	22.73% (5/22)	4.55% (1/22)	0% (0/22)	0% (0/22)
Has the language an appropriate expressiveness?	81.82% (18/22)	9.09% (2/22)	9.09% (2/22)	0% (0/22)	0% (0/22)
Is the result obtained as I expected?	86.36% (19/22)	13.64% (3/22)	0% (0/22)	0% (0/22)	0% (0/22)
Would you use UPCTforma to produce your content?	77.27% (17/22)	18.18% (4/22)	4.55% (1/22)	0% (0/22)	0% (0/22)

A. METHODOLOGY

An experiment was conducted with a group of users. Twenty-two users participated: 14 teachers, 4 scriptwriters and 4 web developers. The teachers were experienced LaTeX users. They had previously participated in a training course on multimedia and gamification activities. These courses are regularly offered by the DCPC to UPCT teachers. The scriptwriters and web developers were DCPC workers. The scriptwriters had no programming skills, but they had two years of experience in creating scripts by means of the Powerpoint-slide notation used in the manual process commented on in Section III-C.

The experiment consisted of two sessions of 3 hours each. In the first session, the participants received training in the four DSLs. A user manual available on [23] was previously delivered to the participants. This manual uses a demo course to illustrate the syntax and semantics of each statement of the four DSLs. DSL statements were presented in the session for an hour and half. The participants were then asked to create the demo course and complete a questionnaire. A computer with DSL editors and engines, a document that described the course elements, and the URL of the questionnaire to be completed (Google Forms) were provided to each user. Once the course and questionnaire details had been clarified, the participants started to create the course. It was estimated that the time required to produce the course would be in the range of 90 to 160 minutes. The second session was held the next day, and the participants also completed the course development and the questionnaire.

The demo course is structured as follows. It is formed of three content units, an assessment unit, a gamification definition, and a sequencing definition. Unit 1 includes 4 sections, 7 gamified widgets, 3 LaTeX mathematical expressions and 6 rows of text. Units 2 and 3 are both composed of a section, a gamified widget, and a row that contains a text element. The assessment unit includes ten single answer questions. With regard to course sequencing, Unit 1 is the only unit open when starting the course. The assessment unit must be opened when Unit 1 is completed. Either Unit 2 or Unit 3 will be opened, depending the score obtained in the assessment unit. Points and badges must be specified for the four units and the widgets.

Once the participants finished writing all the DSL scripts, they were able to check the automatically generated course using a Web browser. They then had to complete the anonymous seven-question questionnaire intended to obtain feedback on their experience. The questions were graded using a 5-point Likert scale. A free text question was also included to enquire about the difficulties encountered when using the DSLs. These questions were asked in order to assess three characteristics of the DSLs that are of interest: Functional suitability, Usability, and Expressiveness. We also evaluated Productivity and Maintainability.

B. RESULTS AND DISCUSSION

The average times (measured in minutes) the participants spent producing the course were 80 (developers), 106 (teachers), and 135 (scriptwriters). The scriptwriters spent nearly 75% longer than the developers. The fact that scriptwriters are not experienced in coding could explain this increase. The time spent creating the course was, therefore, in the estimated range for the participants. Table 1 shows the questions and the percentage for each of their roles.

We shall now analyze the results obtained in order to evaluate the characteristics mentioned above.

1) FUNCTIONAL SUITABILITY

In [24], functional suitability refers to the degree to which a DSL meets the needs of the application domain. A DSL is functionally suitable if it provides constructs for the all concepts and relationships in the domain (completeness), and allows domain-specific solutions to be adequately expressed (appropriateness).

Most of the participants (95.45%) would use the authoring tool to produce their content. All the participants indicated that the results obtained were as expected.

2) USABILITY

This refers to the degree to which the DSL is easy to use. This quality concerns the ease of reading and understanding the code (understandability), and the ease of learning it (learnability). INDIeAuthor promotes usability by offering simple statements that make it possible to write clear and concise code.

The vast majority of the participants (95.45%) agreed or totally agreed that the language is easy to use. All the participants completed the course development within the estimated time, which could have contributed to their perception.

All the teachers and developers felt comfortable with the textual syntax, and they expressed their preference for the textual DSLs over the use of GUIs. However, the scriptwriters stated that they would prefer a typical authoring tool providing a GUI. They also considered that the DSLs were more difficult to use than the Powerpoint-slide notation.

72.72% of the participants agreed or totally agreed that the language is easy to learn. Although the scriptwriters had two years experience in writing scripts with the Powerpoint-slide notation, they encountered difficulties when writing DSL scripts. They got very confused with the use of separators (curly brackets and commas). In fact, they required help from the trainers. However, they recognized that they could be fluent in the four DSLs in a short time.

3) EXPRESSIVENESS

In the case of DSLs, an adequate expressiveness is achieved by having a single construct for each domain concept, and ignoring domain concepts that are not needed to express solutions. All the developers and teachers totally agreed that the DSL expressiveness is appropriate. Three teachers indicated that not supporting code content was an important limitation for courses in a Computer Science syllabus. But this evidences the lack of a kind of content rather than a concept in the domain. The language can be easily extended to support new kinds of content.

We obtained some interesting feedback from the free text question in the questionnaire. The scriptwriters showed a preference for WYSIWYG editors. The teachers suggested some extensions, such as supporting code content and importing content from Powerpoint presentations. They also indicated some improvements to the syntax, such as those related to ranges. The developers mentioned the convenience of editing styles.

4) PRODUCTIVITY

One of the main benefits of using DSLs is the improvement to productivity. DSLs allow solutions to be expressed at a higher level of abstraction, (more simple and concise programs), and this can significantly shorten the development time. We analyzed to what extent INDIeAuthor is productive by means of two experiments.

In the first, we contrasted the effort required to develop a 198-page course using INDIeAuthor and a traditional web development. The process explained in section III-C was applied in the traditional solution. Table 2 compares the development times for traditional and INDIeAuthor solutions. The time spent in the scriptwriting, web development and interoperability stages are shown. When using DSLs, the web development and interoperability stages are not necessary, as the code is automatically generated. This means a reduction of 57.1 hours (59%) in the development time. The scriptwriting time was increased by only 2 hours (5%), but the effort devoted to the scriptwriting task should be reduced as the scriptwriters gain experience in the language. In this experiment, a scriptwriter used INDIeAuthor to create the course, but please recall that the university's objective is for teachers to directly use the tool to create their own courses.

TABLE 2. Comparison of effort (hours) required to develop a course with INDIeAuthor and a traditional solution.

	Scriptwriting	Web developing	Interoperability implementing	Total time
Traditional solution	38	56,10	3	97,10
INDIeAuthor language	40	0	0	40

TABLE 3. Size of the metamodel, grammar and model transformation for INDIeAuthor.

	Metamodel elements	Grammar rules	Model to Text transformation
Content creation DSL	30 classes 49 relations 5 constraints	38	1451 code lines
Assessment creation DSL	17 classes 21 relations	22	84 code lines
Course Sequencing DSL	12 classes 16 relations	17	271 code lines
Total	59 classes 86 relations 5 constraints	77	1806 code lines

Some metrics commonly used to measure the size of a metamodel, (number of elements, relationships and OCL constraints), the grammar (number of rules), and model transformations (number of rules and lines of code) are shown in for INDIeAuthor. It took approximately 400 hours to develop the INDIeAuthor DSL. As gaining productivity was estimated to be 59% for the 198-page course, the development of only 5 courses of a similar size would be sufficient for a return on investment.

In the case of the UPCT, the construction of INDIeAuthor has meant a significant saving of effort, as a Web development was being applied to produce courses. However, the existence of authoring tools, which also automate the process of creating a course, entails a comparison of INDIeAuthor with these tools to check whether the degree of automation provided is similar. A second experiment was, therefore, carried to contrast the productivity of INDIeAuthor with that of another authoring tool. Two teachers, who participated in the case study, received an hour and a half's training in the use of the EasyGenerator authoring tool. After this training, they used this tool to create Unit 1 of the demo course. They had required 40 and 44 minutes to produce this unit with INDIeAuthor, and the times taken with EasyGenerator were 42 and 41 minutes, respectively. Experienced Latex teachers, may, therefore, require a similar effort to create courses using a WYSIWYG editor and a textual notation. Please note that the participants ''copied&pasted'' the code examples of the INDIeAuthor user manual to create the content of the demo course, signifying that they only had to customize the copied code to the context of use.

5) MAINTAINABILITY

INDIeAuthor facilitates maintenance by being easy to use and understand. The effort required to discover code affected by changes is easier than using a general-purpose programming language, and is not more difficult than using languages such as HTML or XML.

When MDE techniques are used to implement a language, any change to the language requires the modification of three artifacts: metamodel, concrete syntax, and model transformations. In the case of INDIeAuthor, the use of viewpoints (i.e., the organization as a DSL family) facilitates the evolution of the language. For example, in order to add a new content type, it is necessary to add a new class to the Content metamodel, a new rule to the grammar of the notation defined for this metamodel, and rules to the corresponding model transformation so as to generate code content. However, artifacts related to the other DSLs will not be affected.

C. BENEFITS OF USING MDE TECHNIQUES

The main benefit of MDE is commonly recognized as its improvement to productivity. Section VI describes how models have been used to generate code in INDIeAuthor, and the gain in productivity in relation to the previous manual solution has been analyzed above. Here, we address other significant benefits obtained with MDE.

Models have advantages over other metadata formats such as XML and JSON. In particular, they allow the representation of data at a higher level of abstraction, and model transformations can be applied to automate tasks. Below, we present two proof of concept utilities that have been developed for INDIeAuthor.

On the one hand, we have implemented a utility that is able to convert INDIeAuthor content units into *Latex Beamer* presentations. Beamer is a LaTeX class consisting of a set of LaTex commands. We have written an m2t transformation that establishes a mapping between Content metamodel elements and Beamer commands. A Beamer presentation for Unit 1 of the demo course [25] can be found in [26]. A Beamer section is generated for each section in the model, a Beamer frame is generated for each column in a row, and visual elements associated with columns are transformed into Beamer commands. More details on the mapping can be found in [27].

The second tool developed is a utility to draw course sequencing flows. These flows are represented by means of UML activity diagrams. Courses are represented as activities and branching is used to represent the conditional access to courses. We used PlantUML[2] to develop this course sequencing viewer. PlantUML is a textual DSL that generates UML diagrams in several formats (e.g. PNG and Latex). We implemented an m2t transformation to map course elements onto UML activity diagram elements expressed in the PlantUML language. Figure 18 shows the sequencing flow for the Demo Unit 1. By representing the courses as models, we have taken advantage of an existing DSL to easily achieve a graphical representation of course sequencing.

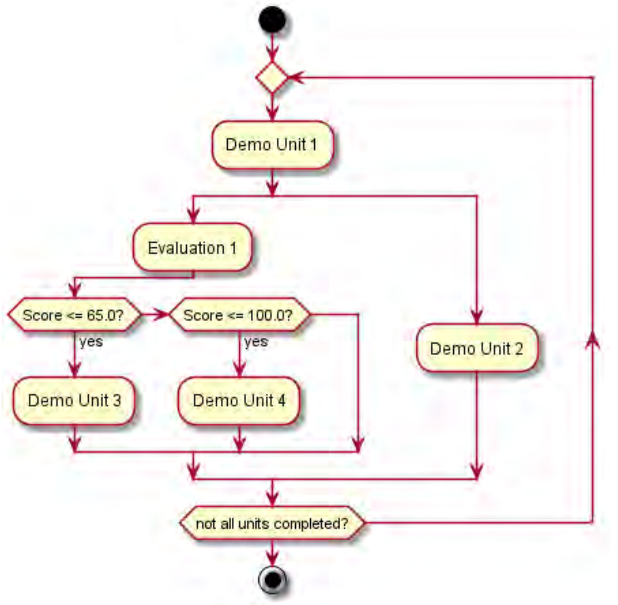


FIGURE 18. Example of course sequencing flow drawn as a UML activity diagram.

Models favor certain software qualities, such as interoperability, portability and evolvability. As noted in [3], a metamodeling language provides a *lingua franca* that facilitates the integration of tools: data from a tool can be represented in the form of models, and these models can be mapped onto models that represent data from another tool. Since they are technology-independent representations, models promote portability among platforms and migration to new technologies. Adobe's announcement in 2017 that Flash Player updates and distribution will stop by the end of 2020[3] is an example of the content migration scenario. Adobe encourages Flash content authors to migrate to new technologies such as HTML5. This would entail producing the content from scratch or the development of tools able to automate the migration. Both solutions would be very costly. Model transformations provide a more productive technology with which to implement software migrations [28]. Migrating INDIeAuthor content to a new platform implies only implementing the model transformations required to convert content models into the representation used on the target platform. Note that migrating INDIeAuthor content to new Web technologies does not involve changes to the existing scripts, but rather extending the DSL engine in order to address the new technology.

The separation between abstract and concrete syntax is a key characteristic of metamodeling-based DSLs. This separation makes it possible to define different notations for the same metamodel and the semantics is not affected. For example, a graphic notation could be defined for INDIeAuthor using some of the available graphical DSL definition tools, such as Sirius. The metamodels and the model

[2]http://plantuml.com

[3]https://theblog.adobe.com/adobe-flash-update/

transformations existing for INDIeAuthor would be completely reusable.

To the best of our knowledge, no authoring tool supports the exportation of courses in the form of models. An innovative feature of INDIeAuthor is, therefore, that it represents courses as Ecore models. Ecore was chosen as a metamodeling language because it is the core element of the EMF platform that is widely used in the academic and industrial MDE community.

VIII. RELATED WORK

This section presents related work organized in three categories: studies on authoring tools, educational modeling languages, and MDE-based approaches used to develop learning applications. Moreover, INDIeAuthor is contrasted with some widely used authoring tools.

1) STUDIES ON AUTHORING TOOLS

Elearning consortiums, networks and forums have published reports in order to provide useful information with which to choose an authoring tool, such as [16] and [19]. These studies identify a set of features to be considered when comparing tools. Some catalogs are also available on websites specialized in elearning topics [18]. These catalogs classify tools in several categories and list the features supported by each tool.

To the best of our knowledge, the feature model presented in this paper is the first effort made to establish a conceptual framework with which to compare authoring tools from the point of view of their technical features and functionality. This proposal allows a systematic and comprehensive comparison of authoring tools. Such a comparison is beyond the scope of this paper. We shall highlight below only the main differences between INDIeAuthor and some leading authoring tools. Please note that our feature model does not consider the evaluation of the *qualities* of the authoring tools, such as pedagogical quality and usability. Other frameworks are required for this. Diwakar *et al.* [29] present a comparative analysis focused on the pedagogical features of authoring tools. Three comparison dimensions are identified and a set of features to be supported by tools is established for each of them. It is noteworthy that all the features of each dimension are supported in INDIeAuthor except for collaborative learning.

2) CONTRASTING INDIeAuthor WITH OTHER AUTHORING TOOLS

We have selected seven leading authoring tools from analyses reported in [16] and [19]. These tools were compared to INDIeAuthor by using the feature model proposed in Section III-A. A course was, therefore, produced with each of the tools evaluated, which was published using the standards offered. The tracking data generated was also checked. The platforms used to test the courses created were Moodle and SCORM Cloud.

None of the tools evaluated provide the following features supported in INDIeAuthor: (i) the definition of new templates, (ii) the units are independent of a particular course, (iii) courses are formed of standards-based units that are imported in the form of a link, (iv) it is possible to gamify complete courses or individual units, (v) the definition of course sequencing, and (vi) predefined tracking data for all the widgets and video. SCORM-based tracking is supported by all the tools, and xAPI is imposed as a new standard format (it is supported by 6 of the tools evaluated). SoftChalk uses LTI but generates tracking data in a proprietary format. LTI/Caliper is supported only in INDIeAuthor. Lectora Inspire allows configurable tracking data. In INDIeAuthor this feature was discarded because it is time consuming. Five tools provide predefined data analysis and visualization. In addition to this feature, tracking data are available for teachers in INDIeAuthor, thus enabling them to implement a customized analysis and visualization, as described in [2]

Tables summarizing the results of the comparative study have not been included here owing to space limitations. They can be found in [30]. These tables show the choices of each tool for the proposed feature model.

3) EDUCATIONAL MODELING LANGUAGES

Learning design can be carried out as a modeling activity. Various languages with which to model aspects related to learning design have, therefore, been defined. These are referred to as ''educational modeling language'' (EML). A classification of these languages is presented in [9], where three categories are identified: evaluation, content-structuring and learning activity definition. The IMS consortium has produced specifications for each of these categories: IMS-QTI for evaluation [31], IMS SS for sequencing courses [32], and IMS LD for teaching-learning processes [10]. These specifications have not, however, been adopted by industry. To the best of our knowledge, no commercial authoring tools and LMS support them. The remaining EMLs have also received little attention from the elearning community to date. Three main factors have influenced this lack of interest: (i) they are difficult for content creators to use, (ii) they provide a XML-based notation to express models, rather than of DSLs, and (iii) the lack of tools to support them.

Some EMLs, e.g. IMS LD and EML-OU [33], define their semantic or conceptual model by means of a metamodel. However this metamodel is not formally expressed with a metamodeling language. In contrast to EMLs, INDIeAuthor provides a simpler vision of the learning design: individual learners follow a course that includes content, evaluation and gamification units, and the order in which units are accessed can depend of the learner's level of achievement. INDIeAuthor integrates four languages that correspond to three categories established in [9]. This avoids having to use different tools to model basic aspects of learning. For example, IMS LD is meant to model the learning-teaching process but has no evaluation and content-structuring aspects; and IMS QTI and IMS SS are focused on expressing evaluation and sequencing, respectively. We should also stress that INDIeAuthor languages were built by applying a metamodeling-based language development, rather than using a machine-readable notation, such as XML. The constructs and notation of these languages are tailored to the course creation domain. The inadequacy of XML notations for educational modeling languages is noted in [34], where the authors propose a graphical DSL for IMS LD.

4) MODEL-DRIVEN APPROACHES TO DEVELOP LEARNING APPLICATIONS

MDE paradigms have been extensively applied in many domains, but the number of relevant works is very limited in the elearning field. The idea of applying language-driven development to build elearning applications was described in [7] and [8]. These works outline how the use of DSLs differs from the conventional approach based on manual coding. Martínez-Ortiz *et al.* [7] illustrate the approach presented by introducing a sequencing language that includes a rule mechanism that allows the expression of an adaptive learning process. The abstract syntax of this language is expressed as an information model in the form of a UML class diagram, and three kinds of concrete syntax are briefly presented: textual, graphical and XML-based notation. However, unlike INDIeAuthor, metamodeling techniques are not really applied: the metamodel is not formally defined by means of a metamodeling language (e.g. Ecore), metamodel-based language workbenches are not used to define the concrete syntax, and model transformations are not implemented to build the language engine. Instead, the work is mainly focused on the definition of an operational semantics for the language example. Moreover, the benefits of using DSLs, and particularly how productivity can be improved in relation to conventional development, are not discussed. All these issues are addressed in depth in our work. We have defined semantics by means of the translational style (DSL sentences are translated into sentences of languages with a well-defined semantics). Translation and interpretation are commonly used to build DSL execution engines in MDE paradigms [6]. The approach described in [7] is also presented in [8], where it is applied to create Socratic tutors. The work is again focused on the operational semantics of a DSL example. A graphical DSL for IMS LD is presented in [34], but the graphical representations are directly transformed into XML, rather than generating models.

IX. CONCLUSIONS AND FUTURE WORK

In this paper we present the INDIeAuthor course authoring tool developed at the DCPC/UPCT. When building INDIeAuthor, we experimented with several issues related to authoring tools: (i) several innovative features are supported, (ii) a textual notation is provided in order to create courses, (iii) MDE principles, techniques and tools were used to create a generative architecture based on a family of 4 textual DSLs, and (iv) a feature model was defined as a result of the domain analysis performed. Below, we show the main conclusions drawn in our work.

A. SOME CONSIDERATIONS REGARDING THE CURRENT STATUS OF AUTHORING TOOLS

As indicated in Section III-A, the authoring tools available vary greatly as regards their features, and several surveys are frequently published to help users choose the most appropriate tool to satisfy their requirements. Here, we illustrate how a conceptual framework in the form of a feature model could be useful to classify authoring tools. The feature model presented may be a first step in that direction. The authors of [29], present a pedagogical analysis of authoring tools carried in the context of engineering faculties. As noted in the previous section, all the features identified in that study as regards creating content with pedagogical quality are supported by INDIeAuthor, except for collaborative work. It is worth noting that our tool significantly improves the reuse currently provided by authoring tools by means of 2 new mechanisms: units can be reused on different courses and user-defined content templates. In INDIeAuthor, we have also explored more powerful gamification and sequencing mechanisms than those currently supported by a very reduced number of tools. The authoring tool market is continuously changing to provide better capabilities. With INDIeAuthor, the UPCT wishes to have its own authoring tool for its teachers, along with an environment that will make it possible to research new features and innovative ideas. INDIeAuthor is available as freeware for any interested educator.

B. CREATING COURSES WITH TEXTUAL LANGUAGES

To the best of our knowledge, all the authoring tools provide WYSIWYG editors, and some of them incorporate a scripting language to allow users to implement certain tasks. However, we have investigated the use of textual languages in INDIeAuthor. Starting with the fact that Latex and textual modeling provide some advantages over graphical editors, our research hypothesis was that ''textual languages could be productive as regards creating educational content, especially for teachers of STEM fields, who are usually experienced in programming and/or LATEX''.

The choice between Latex and WYSIWYG editors [35] to prepare documents is a scenario in which the use of a textual language is contrasted with a graphical editor. The authors of [35] show some of the benefits of using a textual language such as Latex: the ability to easily modularize documents, avoiding the use of proprietary formats that are likely to change, favoring the use of control version systems, the separation of concerns (e.g. content and presentation) can be done properly, and the ability to transform textual scripting into other formats. These advantages are applicable to our approach, as is illustrated throughout this paper. Our approach also provides a productivity similar to authoring tools with WYSIWYG editors. Moreover, the case study shows the appropriateness of our notation. The work presented, therefore, provides evidence that supports the research hypothesis.

C. REPRESENTING EDUCATIONAL CONTENT AS MODELS

Starting from our experience in MDE field, we stated the following research hypothesis: "Some significant benefits can be achieved by representing educational courses as models". The convenience of using models to represent artifacts involved in educational design was introduced by "Educational Modeling Languages". However, the use of XML notations was a serious inconvenience for the acceptance of these languages. An MDE-based approach was proposed in [7] and [8], but it was not based on the MDE common practices and also failed to clearly show the benefits of representing courses as models. In this paper, we show how current MDE techniques can be used to automate the development of learning applications, in particular an authoring tool. We also demonstrate how the representation in the form of models facilitates the building of utilities for course management.

As indicated in Section I, the work presented in this paper is the basis of a three-year KA2 European project. Teachers from three countries will be able create and share content using INDIeAuthor and INDIeOpen (the content repository). This project intends to develop a more mature freeware version of INDIeAuthor. The future directions of our work will, therefore, be focused on the aims of this project. The work planned includes: (i) developing a graphical notation using the Sirius workbench [36]; (ii) creating widgets for new content, such as code and an interactive video; (iii) designing a DSL whose objective will be to configure the tracking data analysis and visualization desired by teachers; (iv) defining a search functionality for the content repository; (v) establishing a mapping between INDIeAuthor and Powerpoint in order to import/export courses between both tools; (vi) applying a model checking to detect errors in the sequencing model.

REFERENCES

[1] L. S. Bacow and *et al.*, *Barriers to Adoption of Online Learning Systems in US Higher Education*. New York, NY, USA: Ithaka, 2012.

[2] D. Pérez-Berenguer and J. García-Molina, "A standard-based architecture to support learning interoperability: A practical experience in gamification," *Softw., Pract. Exper.*, vol. 48, no. 6, pp. 1238–1268, 2018.

[3] M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice*. San Rafael, CA, USA: Morgan & Claypool, 2012.

[4] J. Whittle, J. Hutchinson, and M. Rouncefield, "The state of practice in model-driven engineering," *IEEE Softw.*, vol. 31, no. 3, pp. 79–85, May 2014.

[5] M. Fowler, *Domain—Specific Languages*. Reading, MA, USA: Addison-Wesley, 2010.

[6] M. Voelter. (2013). *DSL Engineering*. [Online]. Available: http://dslbook.org/

[7] I. Martínez-Ortiz, J.-L. Sierra, B. Fernández-Manjón, and A. Fernández-Valmayor, "Language engineering techniques for the development of e-learning applications," *J. Netw. Comput. Appl.*, vol. 32, no. 5, pp. 1092–1105, 2009.

[8] J.-L. Sierra, B. Fernández-Manjón, and A. Fernández-Valmayor, "A language-driven approach for the design of interactive applications," *Interacting Comput.*, vol. 20, no. 1, pp. 112–127, 2008.

[9] B. Fernández-Manjón and J. M. Sánchez-Pérez, "A conceptual introduction and a high-level classification," in *Computers and Education. E-Learning, From Theory to Practice*. Springer, 2007, pp. 27–40.

[10] *IMS Learning Design*. Accessed: Aug. 31, 2018. [Online]. Available: https://www.imsglobal.org/learningdesign/index.html

[11] T. Clark, P. Sammut, and J. S. Willans. (2015). "Applied metamodelling: A foundation for language driven development (third edition)." [Online]. Available: https://arxiv.org/abs/1505.00149

[12] S. Kelly and J.-P. Tolvanen, *Domain-Specific Modeling: Enabling Full Code Generation*. Hoboken, NJ, USA: Wiley, 2008.

[13] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework 2.0*, 2nd ed. Reading, MA, USA: Addison-Wesley, 2009.

[14] J. S. Cuadrado and J. G. Molina, "A model-based approach to families of embedded domain-specific languages," *IEEE Trans. Softw. Eng.*, vol. 35, no. 6, pp. 825–840, Nov. 2009.

[15] K. Czarnecki and U. W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. Reading, MA, USA: Addison-Wesley, 2000.

[16] P. Shank and J. Ganci, "eLearning authoring tools 2013: What we're using, what we want," The eLearning Guild, Survey, 2013.

[17] *The Ultimate List Of HTML5 eLearning Authoring Tools*. Accessed: Nov. 14, 2018. [Online]. Available: https://elearningindustry.com/

[18] *eLearning Authoring Tools*. Accessed: Nov. 14, 2018. [Online]. Available: https://elearningindustry.com/directory/software-categories/elearning-authoring-tools

[19] *Choosing Authoring Tools*. Accessed: Aug. 31, 2018. [Online]. Available: https://www.adlnet.gov/public/uploads/ChoosingAuthoringTools.docx

[20] G. Siemens. (Aug. 5, 2011). *Learning and Academic Analytics*. [Online]. Available: http://www.learninganalytics.net/?p=131

[21] P. Herzig, "Gamification as a service," Ph.D. dissertation, Dresden Univ. Technol., Dresden, Germany, 2014.

[22] D. Pérez-Berenguer and J. García-Molina, "Un enfoque para la creación de contenido online interactivo," *Revista Educación a Distancia*, vol. 51, Nov. 2016, Art. no. 3. [Online]. Available: https://revistas.um.es/red/article/view/275151/199631

[23] *UPCTauthor User Manual*. Accessed: Dec. 9, 2018. [Online]. Available: http://cpcd.upct.es/upctforma/

[24] G. Kahraman and S. Bilgen, "A framework for qualitative assessment of domain-specific languages," *Softw. Syst. Model.*, vol. 14, no. 4, pp. 1505–1526, 2015.

[25] *Unit 1 of the UPCTauthor Demo Course*. [Online]. Available: http://cpcd.upct.es/autor/unidad1/

[26] *Beamer presentation for Unit 1*. [Online]. Available: http://cpcd.upct.es/upctforma/beamer.pdf

[27] *UPCTauthor Beamer Details*. [Online]. Available: http://cpcd.upct.es/upctforma/beamerdetails.html

[28] F. J. B. Ruiz, J. G. Molina, and O. D. García, "On the application of model-driven engineering in data reengineering," *Inf. Syst.*, vol. 72, pp. 136–160, Dec. 2017.

[29] A. Diwakar, M. Patwardhan, and S. Murthy, "Pedagogical analysis of content authoring tools for engineering curriculum," in *Proc. IEEE 4th Int. Conf. Technol. Educ. (T4E)*, Hyderabad, India, Jul. 2012, pp. 83–89.

[30] *Authoring Tools Comparative Study*. Accessed: Dec. 9, 2018. [Online]. Available: http://cpcd.upct.es/upctforma/study.html

[31] *IMS Question & Test Interoperability Specification*. Accessed: Nov. 14, 2018. [Online]. Available: https://www.imsglobal.org/question/index.html

[32] *IMS Simple Sequencing Specification*. Accessed: Nov. 14, 2018. [Online]. Available: http://www.imsglobal.org/simplesequencing/index.html

[33] R. Koper, "Modeling units of study from a pedagogical perspective: The pedagogical meta-model behind EML," Educ. Technol. Expertise Centre, Open Univ. Netherlands, First Draft, Tech. Rep. version 4, Jun. 2001.

[34] I. Martínez-Ortiz, J. L. Sierra, and B. Fernández-Manjón, "Authoring and reengineering of IMS learning design units of learning," *IEEE Trans. Learn. Technol.*, vol. 2, no. 3, pp. 189–202, Jul. 2009.

[35] *What are the Benefits of Using LaTeX Over MS Word, Especially for a Scientific Researcher Doing a Lot of Biology and Mathematics?* Quora, Mountain View, CA, USA, Accessed: Nov. 14, 2018.

[36] *Sirius Website*. Accessed: Nov. 14, 2018. [Online]. Available: https://eclipse.org/sirius/

DANIEL PÉREZ-BERENGUER received the Degree in computer science from the University of Murcia, Murcia, Spain, in 2002, where he is currently pursuing the Ph.D. degree.

He is also an Associate Professor with the Department of Information Technology and Communications, Universidad Politécnica de Cartagena, where he also leads the Digital Content Production Center. His research interests include educational technology, authoring tools, gamification, and learning analytics.

JESÚS GARCÍA-MOLINA received the Ph.D. degree in chemistry from the University of Murcia, Spain, in 1987, where he has been a Full Professor with the Faculty of Informatics, since 1991.

He leads the Modelum Group, an R&D group focused on model-driven engineering with a close partnership with industry. He has authored two textbooks, and its main research contributions are listed in DBLP. His research interests include model-driven development, domain-specific languages, and model-driven modernization.