Declarative representation of curricula models: an LTL- and UML-based approach

Matteo Baldoni, Cristina Baroglio,
Giuseppe Berio, and Elisa Marengo
Dipartimento di Informatica — Università degli Studi di Torino
C.so Svizzera, 185 — I-10149 Torino (Italy)
{baldoni,baroglio,berio}@di.unito.it
elisa.mrng@gmail.com

***Abstract*—In this work, we present a constrained-based representation for specifying the goals of "course design", that we call curricula model, and introduce a graphical language, grounded into Linear Time Logic, to design curricula models which include knowledge of proficiency levels. Based on this representation, we show how model checking techniques can be used to verify that the user's learning goal is supplied by a curriculum, that a curriculum is compliant to a curricula model, and that competence gaps are avoided. This proposal represents the most recent advancement of a work, carried on in the last years, in which we are investigating the use of both agents and web services for building and validating curricula. We also outline future research directions.**

I. Introduction and motivations

As recently underlined by other authors, there is a strong relationship between the development of peer-to-peer, (web) service technologies and e-learning technologies [22]. The more learning resources are freely available through the Web, the more modern e-learning management systems (LMSs) should be able to take advantage from this richness: LMSs should offer the means for easily retrieving and assembling e-learning resources so to satisfy specific users' learning goals, similarly to how (web) services are retrieved and composed [17]. In [6], we have shown the possibility of automatically composing SCORM [1] courseware by exploiting semantic web technology and, in particular, LOM annotations. More rcently [3], we have developed a reasoning service that has been integrated in the Personal Reader framework, a service-oriented learning platform. The reasoning service is basically a planner, which can build curricula in a goal-driven way, where the goal is a set of desired competences. The reasoner is invoked in a service-oriented fashion to help a user and build a curriculum. To this aim, the reasoner is fed with a set of initial competences that the user has, the competences that the user would like to acquire, and the URL of a repository of descriptions of courses, given as RDF triples.

Besides building curricula, there are other interesting tasks that can be performed. Some of these concern curricula which are supplied directly by users. As in a composition of web services it is necessary to verify that, at every point, all the information necessary to the subsequent invocation will be available, in a learning domain, it is important to verify that all the *competencies*, i.e. the *knowledge*, necessary to fully understand a learning resource are introduced or available before that learning resource is accessed. The composition of learning resources, a curriculum, does not have to show any *competence gap*. Unfortunately, this verification, as stated in [15], is usually performed *manually* by the learning designer, with hardly any guidelines or support.

A recent proposal for automatizing the competence gap verification is done in [22] where an analysis of pre- and post-requisite annotations of the Learning Objects (LO), representing the learning resources, is proposed. A logic based validation engine can use these annotations in order to validate the curriculum/LO composition. Melia and Pahl's proposal is inspired by the CocoA system [12], that allows to perform the analysis and the consistency check of static web-based courses. Competence gaps are checked by a prerequisite checker for *linear courses*, simulating the process of teaching with an overlay student model. Pre- and post-requisites are represented as "concepts".

Together with the verification of consistence gaps, there are other kinds of verification. Brusilovsky and Vassileva [12] sketch some of them. In our opinion, two are particularly important: (a) verifying that the curriculum allows to achieve the users' *learning goals*, i.e. that the user will acquire the desired knowledge, and (b) verifying that the curriculum is compliant against the *course design goals*. Manually or automatically supplied curricula, developed to reach a learning goal, should match the "design document", a *curricula model*, specified by the institution that offers the possibility of personalizing curricula. Curricula models specify general rules for designing sequences of learning resources (courses). We interpret them as *constraints*, that are expressed in terms of concepts and, in general, are not directly associated to learning resources, as instead is done for pre-requisites. They constrain the process of acquisition of concepts, independently from the resources.

The availability of languages for designing curricula models, in a way that can automatically be processed by a reasoning system (be it an agent or a service) is a fundamental milestone in the development of checkers that perform the verifications described above, so to supply the the user and, when present, also the organization which supplies the courses, with a complete set of tools to develop personalized, sound and complete curricula.

In this paper we present a constraint-based representation of curricula models. Constraints are expressed as formulas in a temporal logic (LTL, linear temporal logic [16]) represented by means of a simple graphical language that we call DCML (*Declarative Curricula Model Language*). This logic allows the verification of properties of interest for all the possible executions of a model, which in our case corresponds to the specific curriculum. Curricula are represented as *activity diagrams* [2]. We translate an activity diagram, that represents a curriculum, in a *Promela* program [21] and we check, by means of the well-known SPIN Model Checker [21], that it respect the model by verifying that the set of LTL formulas are satisfied by the Promela program. Moreover, we check that learning goals are achieved, and that the curriculum does not contain competence gaps. This work also improves the proposal of [9], where we did not consider the duration of courses and the fact that they may (partially) overlap. This leads to a different representation based on the concept of *milestones*. As in [15], we distinguish between *competency* and *competence*, where by the first term we denote a concept (or skill) while by the second we denote a competency plus the level of proficiency at which it is learnt or known or supplied. So far, we do not yet tackle with "contexts", as defined in the competence model proposed in [15], which will be part of future work.

This approach differs from previous work [7], where we presented an adaptive tutoring system, that exploits *reasoning about actions and changes* to plan and verify curricula. The approach was based on abstract representations, capturing the *structure* of a curriculum, and implemented by means of prolog-like logic clauses. Such representations were applied a procedure-driven form of planning, in order to build personalized curricula. In this context, we proposed also some forms of verification, of competence gaps, of learning goal achievement, and of whether a curriculum, given by a user, is compliant to the "course design" goals. The use of procedure clauses is, however, limiting because they, besides having a *prescriptive* nature, pose very strong constraints on the sequencing of learning resources. In particular, clauses represent what is "legal" and whatever sequence is not foreseen by the clauses is "illegal". However, in an open environment where resources are extremely various, they are added/removed dynamically, and their number is huge, this approach becomes unfeasible: the clauses would be too complex, it would be impossible to consider all the alternatives and the clauses should change along time.

For this reason we considered as appropriate to take another perspective and represent only those constraints which are strictly necessary, in a way that is inspired by the so called *social approach* proposed by Singh for multi-agent and service-oriented communication protocols [23], [24]. In this approach only the *obligations* are represented. In our application context, obligations capture relations among the times at which different competencies are to be acquired. The advantage of this representation is that we do not have to represent all that is legal but only those *necessary conditions* that characterize a legal solution. To make an example, by means of constraints we can request that a certain knowledge is acquired before some other knowledge, without expressing what else is to be done in between. If we used the clause-based approach, instead, we should have described also what can legally be contained between the two times at which the two pieces of knowledge are acquired. Generally, the constraints-based approach is more flexible and more suitable to an open environment.

II. DCML: A Declarative Curricula Model Language

In this section we describe the *Declarative Curricula Model Language* (DCML, for short), a graphical language to represent the specification of a curricula model (the course design goals). The advantage of a graphical language is that drawing, rather than writing, constraints facilitates the user, who needs to represent curricula models, allowing a general overview of the relations which exist between concepts. DCML is inspired by DecSerFlow, the Declarative Service Flow Language to specify, enact, and monitor web service flows by van der Aalst and Pesic [26]. DCML, as well as DecSerFlow, is grounded in Linear Temporal Logic [16] and allows a curricula model to be described in an easy way maintaining at the same time a rigorous and precise meaning given by the logic representation. LTL includes temporal operators such as next-time ($\bigcirc\varphi$, the formula φ holds in the immediately following state of the run), eventually ($\Diamond\varphi$, φ is guaranteed to eventually become true), always ($\Box\varphi$, the formula φ remains invariably true throughout a run), until ($\alpha \cup \beta$, the formula α remains true until β), see also [21, Chapter 6]. The set of LTL formulas obtained for a curricula model are, then, used to verify whether a curriculum will respect it [5]. The adoption of a graphical language with a logical grounding allows designers, who cannot be expected to feel comfortable with the logical notation, to take advantage of automatic tools for the verification of the various kinds of properties mentioned in the introduction. As an example of curricula model, Fig. 1 shows a curricula model expressed in DCML. Every box contains at least one competence. Boxes/competences are related by arrows, which represent (mainly) temporal constraints among the times at which they are to be acquired. Altogether the constraints describe a curricula model.

A. Competence, competency, and basic constraints

The terms *competence* and *competency* are used, in the literature concerning professional curricula and e-learning, to denote the "effective performance within a domain at some level of proficiency" and "any form of knowledge, skill, attitude, ability or learning objective that can be described in a context of learning, education or training". In the following, we extend a previous proposal [5], [10] so as to include a representation of the *proficiency level* at which a competency

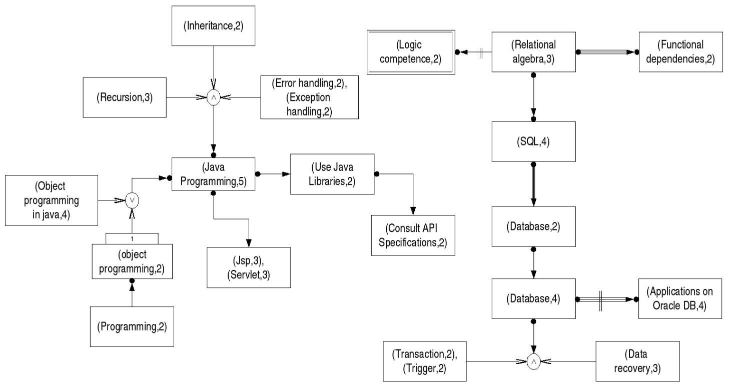

Fig. 1. An example of curricula model in DCML.

is owned or supplied. To this aim, we associate to each competency a variable k, having the same name as the competency, which can be assigned natural numbers as values. The value of k denotes the proficiency level; zero means absence of knowledge. Therefore, k encodes a *competence*, Fig. 2(a). On competences, we can define three basic *constraints*.

The "*level of competence*" constraint, Fig. 2(c), imposes that a certain competency k must be acquired at least at level l. It is represented by the LTL formula $\Diamond(k \geq l)$. Similarly, a course designer can impose that a competency must never appear in a curriculum with a proficiency level higher than l. This is possible by means of the "*always less than level*" constraint, shown in Fig. 2(d). The LTL formula $\Box(k < l)$ expresses this fact (it is the negation of the previous one). As a special case, when the level l is one ($\Box(k < 1)$), the competency k must never appear in a curriculum.

The third constraint, represented by a double box, see Fig. 2 (b), specifies that k must belong to the initial knowledge with, at least, level l. In other words, the simple logic formula $(k \geq l)$ must hold in the initial state.

To specify relations among concepts, other elements are needed. In particular, in DCML it is possible to represent *Disjunctive Normal Form* (DNF) formulas as *conjunctions* and *disjunctions* of concepts. For the sake of semplicity, in the next section we present the various constraints that can be expressed by DCML without using DNF, the interested reader can find the extension in the appendix.

B. Positive and negative relations among competences

Besides the representation of competences and of constraints on competences, DCML allows to represent *relations* among competences. For simplicity, in the following presentations we will always relate simple competences, it is, however, of course possible to connect DNF formulas. We will denote by (k, l) the fact that competence k is required to have at least level l (i.e. $k \geq l$) and by $\neg(k, l)$ the fact that k is required to be less than l.

Arrows ending with a little-ball, Fig. 2(f), express the *before* temporal constraint between two competences, that amount to require that (k_1, l_1) holds *before* (k_2, l_2). This constraint can be used to express that to understand some topic, some proficiency of another is required as precondition. It is important to underline that if the antecedent never becomes true, also the consequent must be invariably false; this is expressed by the LTL formula $\neg(k_2, l_2)\ \mathsf{U}\ (k_1, l_1)$, i.e. $(k_2 < l_2)\ \mathsf{U}\ (k_1 \geq l_1)$. It is also possible to express that a competence must be acquired *immediate before* some other. This is represented by means of a triple line arrow that ends with a little-ball, see Fig. 2(i). The constraint (k_1, l_1) *immediate before* (k_2, l_2) imposes that (k_1, l_1) holds before (k_2, l_2) and the latter either is true in the next state w.r.t. the one in which (k_1, l_1) becomes true or k_2 *never* reaches the level l_2. The difference w.r.t the *before* constraint is that it imposes that the two competences are acquired *in sequence*. The corresponding LTL formula is "(k_1, l_1) *before* (k_2, l_2)" $\wedge\Box((k_1, l_1) \supset (\bigcirc(k_2, l_2) \vee \Box\neg(k_2, l_2)))$.

Both of the two previous relations represent temporal constraints between competences. The *implication* relation (Fig. 2(e)) specifies, instead, that if a competency k_1 holds at least at the level l_1, some other competency k_2 must be acquired sooner or later at least at the level l_2. The main characteristic of the implication, is that the acquisition of the consequent is imposed by the truth value of the antecedent, but, in case this one is true, it does not specify when the consequent must be achieved (it could be before, after or in the same state of the antecedent). This is expressed by the LTL formula $\Diamond(k_1, l_1) \supset \Diamond(k_2, l_2)$. The *immediate implication* (Fig. 2(h)), instead, specifies that the consequent must *hold* in the state right after the one in which the antecedent is acquired. Note that, this does not mean that it must be *acquired* in that state, but only that it cannot be acquired after. This is expressed by the LTL implication formula in conjunction with the constraint that whenever $k_1 \geq l_1$ holds, $k_2 \geq l_2$ holds in the next state: $\Diamond(k_1, l_1) \supset \Diamond(k_2, l_2) \wedge \Box((k_1, l_1) \supset \bigcirc(k_2, l_2))$.

The last two kinds of temporal constraint are *succession* (Fig. 2(g)) and *immediate succession* (Fig. 2(j)). The *succession* relation specifies that if (k_1, l_1) is acquired, afterwards (k_2, l_2) is also achieved; otherwise, the level of k_2 is not important. This is a difference w.r.t. the *before* constraint where, when the antecedent is never acquired, the consequent

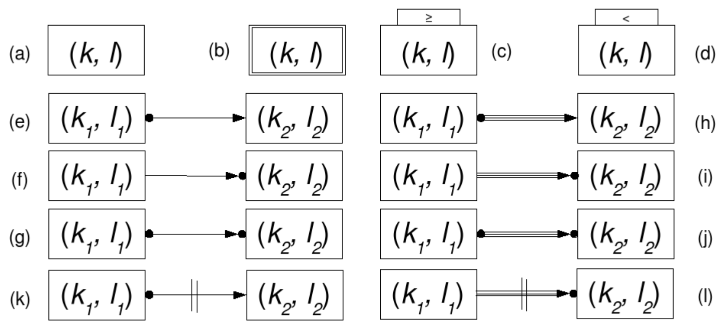

Fig. 2. Competences (a) and basic constraints (b), (c), and (d). Relations among competences: (e) implication, (f) before, (g) succession, (h) immediate implication, (i) immediate before, (j) immediate succession, (k) not implication, (l) not immediate before.

must be invariably false. Indeed, the *succession* specifies a condition of the kind *if* $k_1 \geq l_1$ *then* $k_2 \geq l_2$, while *before* represents a constraint without any conditional premise. Instead, the fact that the consequent must be acquired after the antecedent is what differentiates *implication* from *succession*. Succession constraint is expressed by the LTL formula $\Diamond(k_1, l_1) \supset (\Diamond(k_2, l_2) \wedge (\neg(k_2, l_2) \ \mathsf{U} \ (k_1, l_1)))$. In the same way, the *immediate succession* imposes that the consequent either is acquired in the same state as the antecedent or in the state immediately after (not before nor later). The immediate succession LTL formula is "(k_1, l_1) *succession* (k_2, l_2)" $\wedge \Box((k_1, l_1) \supset \bigcirc(k_2, l_2))$.

After the "positive relations" among competences, let us now introduce the graphical notations for "negative relations". The graphical representation is very intuitive: two vertical lines break the arrow that represents the constraint, see Fig. 2(k)-(l). (k_1, l_1) *not before* (k_2, l_2) specifies that k_1 cannot be acquired up to level l_1 before or in the same state when (k_2, l_2) is acquired. The corresponding LTL formula is $\neg(k_1, l_1) \ \mathsf{U} \ ((k_2, l_2) \wedge \neg(k_1, l_1))$. Notice that this is not obtained by simply negating the before relation but it is weaker; the negation of *before* would *impose the acquisition* of the concepts specified as consequents (in fact, the formula would contain a strong until instead of a weak until), the *not before* does not. The *not immediate before* is translated exactly in the same way as the *not before*. Indeed, it is a special case because our domain is monotonic, that is a competency acquired at a certain level cannot be forgotten.

(k_1, l_1) *not implies* (k_2, l_2) expresses that if (k_1, l_1) is acquired k_2 cannot be acquired at level l_2; as an LTL formula: $\Diamond(k_1, l_1) \supset \Box\neg(k_2, l_2)$. Again, we choose to use a weaker formula than the natural negation of the implication relation because the simple negation of formulas would impose the presence of certain concepts. (k_1, l_1) *not immediate implies* (k_2, l_2) imposes that when (k_1, l_1) holds in a state, $k_2 \geq l_2$ must be false in the immediately subsequent state. Afterwards, the proficiency level of k_2 does not matter. The corresponding LTL formula is $\Diamond(k_1, l_1) \supset (\Box\neg(k_2, l_2) \vee \Diamond((k_1, l_1) \wedge \bigcirc\neg(k_2, l_2)))$, that is weaker than the "classical negation" of the *immediate implies*.

The last relations are *not succession*, and *not immediate succession*. The first imposes that a certain competence cannot be acquired after another, (either it was acquired before, or it will never be acquired). As LTL formula, it is $\Diamond(k_1, l_1) \supset (\Box\neg(k_2, l_2) \vee$ "(k_1, l_1) *not before* (k_2, l_2)"). The second imposes that if a competence is acquired in a certain state, in the state that follows, another competence must be false, that is $\Diamond(k_1, l_1) \supset (\Box\neg(k_2, l_2) \vee$ "(k_1, l_1) *not before* (k_2, l_2)" $\vee \Diamond((k_1, l_1) \wedge \bigcirc\neg(k_2, l_2)))$.

In Fig. 1, some examples of constraints are represented. Conjunctions and disjunctions are represented by connecting different competences (boxes) with and/or circles. For instance, *Object programming in Java* is required at least at level 4 **or** *Object programming* is required at least at level 2, before the competence *Java Programming* can be acquired (at least at level 5).

Another example is the implication that occurs, for instance, between *Database*, at least at level 2, and *Database*, at least at level 4. This relation means that Database at level 2 is not sufficient and, when it is acquired, sooner or later the student must increase its knowledge at least at level 4.

The competence (Database,4) is also connected with an *not immediate succession* constraint to the competence (Application on Oracle DB,4). This constraint can be interpreted as the intention to let the student assimilate the knowledges on Database before applying them on a real case.

Note that this example is divided into two different areas, one concerning programming competences and one about databases. There are no connection between competences of the two parts. Anyway all the constraints must be checked on the curriculum.

III. REPRESENTING CURRICULA AS ACTIVITY DIAGRAMS

Let us now consider specific curricula. In the line of [7], [4], [5], we represent curricula as sequences of courses/resources, taking the abstraction of courses as simple actions. Any action can be executed given that a set of preconditions holds; by executing it, a set of post-conditions, the effects, will become true. Specifically, courses are seen as actions for acquiring some concepts (*effects*) given that the student owns some competences (*preconditions*). So, a curriculum is seen as a sequence of actions that causes *transitions* from the initial set

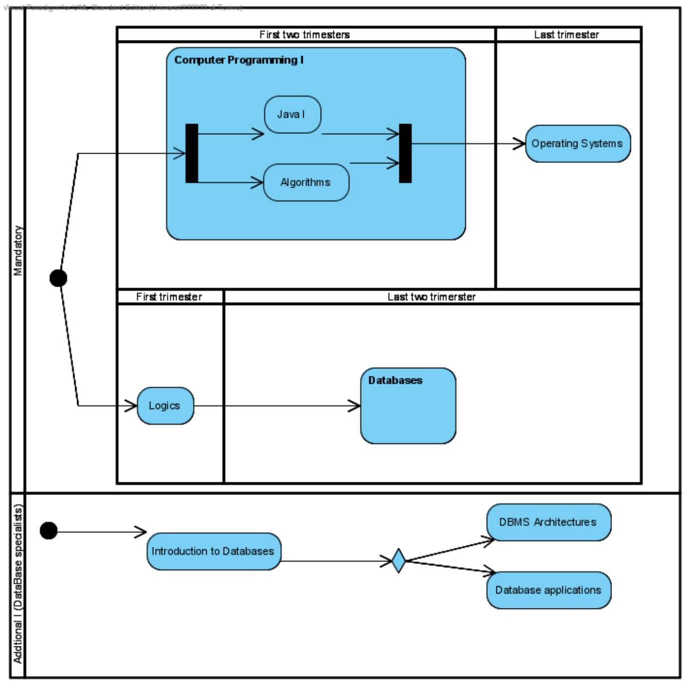

Fig. 3. Activity diagram representing a curriculum with mandatory and additional, student chosen, courses. Swimlanes represent the sequencings of courses. Vertical divisions capture the different milestones (trimesters).

of competences (possibly empty) of a user up to a final state that will contain also the acquired competences. We assume that concepts can only be added to states and competence level can only grow by executing the actions of attending courses (or more in general reading a learning material). The intuition behind this assumption is that new course do not erase the concepts acquired previously, thus knowledge grows incrementally.

Generally speaking, a curriculum may be represented with one or several sequences of courses to be attended, in alternative or as obligations. As a consequence, it seems very natural representing a curriculum by, for instance, a UML *activity diagram* [2]. The diagram represents essentially the "student personal process" to achieve the final degree. Apart its standard meaning and visualisation, a UML activity diagram may contain actions with pre- and post- conditions, combined in complex paths and possibly aggregated. Actions or activities (if further decomposed) correspond to courses or other elements, used to fundamentally build any curriculum in an organisation. Activity diagrams are rich enough to represent alternative, intermediate statuses and conditional paths.

However, we found very useful two principles when representing a curriculum: To carefully distinguish courses with distinct duration (in time); To carefully distinguish mandatory courses and additional optional courses. Modelling a curriculum with these two principles in mind introduces *(i)* a decomposition level and *(ii)* partitions among courses, being these courses from mandatory or from additional partitions.

Actvity diagrams are well suited for representing curricula under the two principle reported above. Fig. 3 reports an example with additional courses and distinguishes courses with distinct duration. The horizontal partition (*swimlanes* in UML) is corresponding to mandatory and additional courses (additional courses are for "database specialists" in this case). Vertical partitions provide information about actions and activities with distinct duration. In this case, we have used as time references the usual distinction implemented in Italian universities, in years and trimesters. The beginning/ending points of the trimesters correspond to a set of *milestones*; this temporal organization will be used to identify those states, at which the verification will be applied. In previous work, instead, courses were *atemporal* and each state was tied to the simulation of a single course. The introduction of durations allows a more realistic representation of the curricula and, especially, of the dependencies between competences. Therefore, we can easily see that the course "Logics" is delivered during the first trimester, while the course "Java I" is delivered in the first six months, being this java course part of an aggregated set of courses corresponding to the activity named "Computer Programming I". In the horizontal bottom swimlane, we are representing the fact that it is a student's choice to advance in the first year two courses of databases, once made the choice between "Database architecture" and "Database applications". The swimlanes representing additional courses can be used to represented once-time choice of the student. For instance, once the student has decided to become "database specialist", he has to complete the process represented in the swimlane. However, with additional swimlanes, we can also represent less stringent choices. In this case, however, there are typically no arrows between courses and there is no final node. It should also be noted that processes representing a curriculum are only *views* combining activities and actions (i.e. the real taught courses). Intuitively, a course like “Network I” in a curriculum for system specialists is to be followed by “Network II”; however, the same is not required in a curriculum for “database specialists”. This is very compatible with UML activity diagrams where it is possible to use reuse, in distinct contexts, activities and actions defined once.

More complex cases require special attention because hierarchical decomposition of the time-based partition does not apply directly. For instance, the case of where one two-trimester course overlaps in time with another two-trimester course. In this case, hierarchical time-based partition cannot be applied but it should be observed that the basic activity diagram is sufficient also in this case because it allows to represent the two courses in parallel. Indeed, due to overlapping, we cannot expect one to supply competences that are prerequisites for the other. Again, with reference to our example, in Fig. 3, the course “Databases” spans over the second and the third trimester, partially overlapping with the “Computer Programming 1” activity (spanning over the first and second trimester) and partly overlapping with “Operating Systems”, in the third trimester.

UML 2.1.1 is extremely powerful for making partitions. Indeed, partitions apply to activities, and contain several edges and actions. This means that each activity can be independently partitioned in the diagram. However, the size of the visualised partitions does not make sense in UML (as well). Therefore, time overlapping can be shown by regulating the size and the relative position of the several visualised partitions; however, the “timed semantics” remains underspecified and may be approached in the classical way by introducing time-dependent constraints on activity edges (or, on top of the interpretation of the UML superstructure specification that often does not provide a sufficient level of detail constraints attached to the partitions themselves).

IV. Verifying curricula by means of SPIN Model Checker

In this section we discuss how to validate a curriculum. As explained, three kinds of verifications have to be performed: (1) verifying that a curriculum does not have competence gaps, (2) verifying that a curriculum supplies the user’s learning goals, and (3) verifying that a curriculum satisfies the course design goals, i.e. the constraints imposed by the curricula model. To do this, we use *model checking techniques* [14].

By means of a *model checker*, it is possible to generate and analyze all the possible states of a program exhaustively to verify whether no execution path satisfies a certain property, usually expressed by a temporal logic, such as LTL. When a model checker refuses the negation of a property, it produces a *counterexample* that shows the violation. SPIN, by G. J. Holzmann [21], is the most representative tool of this kind. Our idea is to translate the activity diagram, that represents a set of curricula, in a Promela (the language used by SPIN) program, and, then to verify whether it satisfies the LTL formulas that represents the curricula model.

In the literature, we can find some proposals to translate UML activity diagrams into Promela programs, such as [18], [19]. These proposals have a different purpose than ours and they cannot directly be used to perform the translation that we need to perform the verifications we list above, however, it is possible to follow them as guidelines to perform our translation. Generally, their aim is debugging UML designs, by helping UML designers to write sound diagrams. The translation proposed in the following, instead, aims to simulate, by a Promela program the acquisition of competencies by attending courses contained into the curricula represented by an activity diagram.

Given a curriculum as an activity diagram, we represent all the competences involved by its courses as *integer variables*. In the beginning, only those variables that represent the initial knowledge owned by the student are set to a value greater than zero. *Courses* are represented as actions that can modify the value of such variables. Since our application domain is monotonic, the value of a variable can only grow.

The Promela program corresponds to a process, that contains the translation of the UML activity diagram and simulates the way competences are acquired, for *all* the curricula represented by the activity diagram, updating the set of the achieved competences at every step. Steps corrispond to the various milestones into which the curriculum is organized. For instance, in Fig. 3 we identify the initial state, a second state corresponding to the end of the first trimester, another corresponding to the end of the second trimester, and a final state, corresponding to the end of the curriculum.

```
proctype CurriculumVerification() {
  milestone_1();
  milestone_2();
  milestone_3();
  LearningGoal();
}
```

If the simulation of all its possible executions ends, then, there is no competence gap.

Each *course* is represented by its preconditions and its effects. For example, the course “Databases” is as follows:

```
inline preconditions_course_databases() {
  assert(logical_reasoning >= 4);
}
inline effects_course_databases() {
  SetCompetenceState(database, 2);
  SetCompetenceState(relational_algebra, 4);
  SetCompetenceState(ER_language, 4);
}
```

assert verifies the truth value of its condition, which in our case is the precondition to the course. If violated, SPIN interrupts its execution and reports about it. *SetCompetenceState* increases the level of the passed competence if its current level is lower than the second parameter. If all the curricula represented by the translated activity diagram have *no competence gaps*, no assertion violation will be detected. Otherwise, a counterexample will be returned that corresponds to an effective sequence of courses leading to the violation, giving a precise feedback to the student/teacher/course designer of the submitted set of curricula.

Generally speaking, a milestone implements the act of adding to the state all the competencies that have been acquired within itself, plus the act of checking the applicability of the subsequent courses (those that will lead to the next milestone). Since each curriculum contains both mandatory and additional courses, the latter depending on a student's choice, every milestone verifies, by default, the mandatory courses and simulates the different alternatives concerning additional courses, which the student might has chosen. This is done by means of the introduction of a variable that is used to discriminate among the alternative paths. *Decision and merge nodes* can be used to represent such altenatives.

```
inline milestone() {
  atomic {
    preconditions_course_java_programming_II();
    if
    :: (path == 1) ->
       preconditions_course_logic();
    :: (path == 2) ->
       precondition_course_physics();
    :: else -> skip;
    fi;
    effects_course_java_programming_II();
    if
    :: (path == 1) ->
       effects_course_logic();
    :: (path == 2) ->
       effects_course_physics();
    :: else -> skip;
    fi;
  }
}
```

The test of the preconditions and the update of the state are performed as an atomic operation.

The last instruction of the process *CurriculumVerification*, which is applied only if all the curricula can be executed to their end, is *LearningGoal*. *LearningGoal* performs the check of the user's learning goal. This just corresponds to a test on the knowledge in the ending state. For example, a student interested in web and databases could have the goal:

```
inline LearningGoal()
{ assert(advanced_java_programming>=5
  && N_tier_architectures >= 4
  && relational_algebra>=2
  && ER_language>=2); }
```

To check if the curriculum complies to a curricula model, we check if every possibly sequence of execution of the Promela program satisfies the LTL formulas, now transformed into *never claims* directly by SPIN. The assertion verification is not computationally expensive. The automata generated from the Promela program encoding the first three years of courses at our University is still tractable. Also the verification of the temporal constraints is not hard if we check the constraints one at the time.

V. Conclusions

In this paper we have introduced a graphical language to describe curricula models as temporal constraints posed on the acquisition of competences (supplied by courses), therefore, taking into account both the concepts supplied/required and the proficiency level. We have also shown how model checking techniques can be used to verify that a curriculum complies to a curricula model, and also that a curriculum both allows the achievement of the user's learning goals and that it has no competence gaps. This use of model checking is inspired by [26], where LTL formulas are used to describe and verify the properties of a composition of Web Services. Another recent work, though in a different setting, that inspired this proposal is [25], where medical guidelines, represented by means of the GLARE graphical language, are translated in a Promela program, whose properties are verified by using SPIN. Similarly to [25], the use of SPIN, gives an *automa-based semantics* to a curriculum (the automaton generated by SPIN from the Promela program) and gives a declarative, formal, representation of curricula models (the set of temporal constraints) in terms of a LTL theory that enables other forms of reasoning. In fact, as for all logical theories, we can use an inference engine to derive other theorems or to discovery inconsistencies in the theory itself.

The presented proposal is an evolution of earlier works [8], [4], [7], where we applied semantic annotations to learning objects, with the aim of building compositions of new learning objects, based on the user's learning goals and exploiting planning techniques. That proposal was based on a different approach that relied on the experience of the authors in the use of techniques for reasoning about actions and changes which, however, suffers of the limitations discussed in the introduction. We are currently working on the automatic translation from a textual representation of DCML curricula models into the corresponding set of LTL formulas and from a textual representation of an activity diagram, that describes a curriculum (comprehensive of the description of all courses involved with their preconditions and effects), into the corresponding Promela program. We are also going to realize a graphical tool to define curricula models by means of DCML. We think to use the Eclipse framework, by IBM, to do this. In [3], we discuss the integration into the Personal Reader Framework [20] of a web service that implements an earlier version of the techniques explained here, which does not include proficiency levels. Last but not least, if in a University framework the notion of competence that we have used is sufficient to represent and reason about curricula, in business organizations this notion usually requires more complex models. As future work, we mean to integrate the proposed approach with the CRAI competence model [13] and with competence management information systems [11].

Acknowledgements.

The authors would like to thank Viviana Patti for the helpful discussions. This research has partially been funded by the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Programme project REWERSE number 506779 (cf. http://rewerse.net), and it has also been supported by MIUR PRIN 2005 "Specification and verification of agent interaction protocols" national

project.

References

[1] ADL Technical Team. SCORM XML controlling document - SCORM CAM version 1.3 navigation XML XSD version 1.0, 2004. http://www.adlnet.org/.

[2] Unified Modeling Language: Superstructure, version 2.1.1. OMG, February 2007.

[3] M. Baldoni, C. Baroglio, I. Brunkhorst, E. Marengo, and V. Patti. Curriculum Sequencing and Validation: Integration in a Service-Oriented Architecture. In *Proc. of EC-TEL'07*, *LNCS*, 2007. Springer.

[4] M. Baldoni, C. Baroglio, and N. Henze. Personalization for the Semantic Web. In *Reasoning Web*, *LNCS 3564 Tutorials*, pp. 173–212. Springer-Verlag, 2005.

[5] M. Baldoni, C. Baroglio, A. Martelli, V. Patti, and L. Torasso. Verifying the compliance of personalized curricula to curricula models in the semantic web. In *Proc. of Int.l Workshop SWP'06, at ESWC'06*, pp. 53–62, 2006.

[6] M. Baldoni, C. Baroglio, V. Patti, and L. Torasso. Reasoning about learning object metadata for adapting SCORM courseware. In *Proc. EAW'04*, 2004.

[7] M. Baldoni, C. Baroglio, and V. Patti. Web-based adaptive tutoring: an approach based on logic agents and reasoning about actions. *Artificial Intelligence Review*, 22(1):3–39, 2004.

[8] M. Baldoni, C. Baroglio, V. Patti, and L. Torasso. Reasoning about learning object metadata for adapting SCORM courseware. In *Proc. of Int.l Workshop EAW'04, at AH 2004*, pp. 4–13, Eindhoven, The Netherlands, August 2004.

[9] M. Baldoni, C. Baroglio, and E. Marengo. Curricula model checking. In *Proc. of AIIA'07*. To appear.

[10] M. Baldoni and E. Marengo. Curricula model checking: declarative representation and verification of properties. In *Proc. of EC-TEL'07*, *LNCS*, 2007. Springer.

[11] G. Berio and M. Harzallah. Knowledge Management for Competence Management. *J. of Universal Knowledge Management*, 1:21–28, 2005.

[12] P. Brusilovsky and J. Vassileva. Course sequencing techniques for large-scale web-based education. *Int. J. Cont. Engineering Education and Lifelong learning*, 13(1/2):75–94, 2003.

[13] M. Harzallah and F. Vernadat. IT-based Competency Modeling and Management: from theory to practice in enterprise engineering and operations. *Computers in industry*, 48:157–179, 2002.

[14] O. E. M. Clarke and D. Peled. *Model checking*. MIT Press, 2001.

[15] J. L. De Coi, E. Herder, A. Koesling, C. Lofi, D. Olmedilla, O. Papapetrou, and W. Sibershi. A model for competence gap analysis. In *Proc. of WEBIST 2007*.

[16] E. A. Emerson. Temporal and model logic. In *Handbook of Theoretical Computer Science*, volume B, pages 997–1072. Elsevier, 1990.

[17] R. Farrell, S. D. Liburd, and J. C. Thomas. Dynamic assebly of learning objects. In *Proc. of WWW 2004*, New York, USA, May 2004.

[18] M. del Mar Gallardo, P. Merino, and E. Pimentel. Debugging UML Designs with Model Checking. *Journal of Object Technology*, 1(2):101–117, July-August 2002.

[19] N. Guelfi and A. Mammar. A Formal Semantics of Timed Activity Diagrams and its PROMELA Translation. In *Proc. of APSEC'05*, pp. 283–290. 2005.

[20] N. Henze and D. Krause. Personalized access to web services in the semantic web. In *The 3rd Int.l Workshop SWUI, at ISWC 2006*, 2006.

[21] G. J. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2003.

[22] M. Melia and C. Pahl. Automatic Validation of Learning Object Compositions. In Proc. of *IT&T'2005: Doctoral Symposium*, Carlow, Ireland, 2006.

[23] M. P. Singh. Agent communication languages: Rethinking the principles. *IEEE Computer*, 31(12):40–47, 1998.

[24] M. P. Singh. A social semantics for agent communication languages. In *In Issues in Agent Communication*, number 1916 in LNCS, pages 31–45. Springer, 2000.

[25] P. Terenziani, L. Giordano, A. Bottrighi, S. Montani, and L. Donzella. SPIN Model Checking for the Verification of Clinical Guidelines. In *Proc. of ECAI 2006 Workshop on AI techniques in healthcare*, Riva del Garda, August 2006.

[26] W. M. P. van der Aalst and M. Pesic. DecSerFlow: Towards a Truly Declarative Service Flow Language. In *Proc. of WS-FM'06*, LNCS, 2006. Springer.

Appendix

Let k be a competence, we denote by (k, l) the constraint $k \geq l$ and by $\neg(k, l)$ the constraint $k < l$. A *conjuctive competence formula* cf is a conjuction of atomic competence constraints $cf = (k_1, l_1) \wedge \cdots \wedge (k_n, l_n)$. A conjunction can also be interpreted as the set of constraints $cf = \{(k_1, l_1), \ldots, (k_n, l_n)\}$. We can extend the definition of *negation*, *level of competence*, *always less than level*, and *next* to a conjunctive competence formula as follow:

- $\mathsf{negation}(cf) = \bigwedge_{(k_i,l_1)\in cf} \neg(k_i, l_i)$;
- $\mathsf{existence}(cf) = \bigwedge_{(k_i,l_i)\in cf} \Diamond(k_i, l_i)$;
- $\mathsf{absence}(cf) = \bigwedge_{(k_i,l_i)\in cf} \Box\neg(k_i, l_i)$;
- $\mathsf{possibility}(cf) = \bigwedge_{(k_i,l_i)\in cf} (\Diamond(k_i, l_i) \vee \Box\neg(k_i, l_i))$.
- $\mathsf{next}(cf) = \bigwedge_{(k_i,l_i)\in cf} \bigcirc(k_i, l_i)$.

A *disjunctive normal competence formulae* dnf is a disjunction of conjunctive competence formulas, $dnf = cf_1 \vee \cdots \vee cf_n$. Again, we also denote a disjunctive normal competence formula as a set of conjuctive competence formulas $dnf = \{cf_1, \ldots, cf_n\}$. Therefore, a disjunctive normal competence formula is a set of sets of atomic competences.

The positive relations presented in Section II-B can be generalised to a DNF formula as follows:

- dnf_1 *before* dnf_2: $\bigvee_{cf_i \in dnf_1, cf_j \in dnf_2} \mathsf{negation}(cf_j)\ \mathsf{U}\ cf_i$;
- dnf_1 *immediate before* dnf_2: $\bigvee_{cf_i \in dnf_1, cf_j \in dnf_2} cf_i$ *before* $cf_j \wedge \Box(cf_i \supset (\mathsf{next}(cf_j) \vee \mathsf{absence}(cf_j)))$;
- dnf_1 *implies* dnf_2: $\bigvee_{cf_i \in dnf_1, cf_j \in dnf_2} \mathsf{existence}(cf_i) \supset \mathsf{existence}(cf_j)$;
- dnf_1 *immediate implies* dnf_2: $\bigvee_{cf_i \in dnf_1, cf_j \in dnf_2} cf_i$ *implies* $cf_j \wedge \Box(cf_i \supset \mathsf{next}(cf_j))$;
- dnf_1 *succession* dnf_2: $\bigvee_{cf_i \in dnf_1, cf_j \in dnf_2} \mathsf{existence}(cf_i) \supset (\mathsf{existence}(cf_j) \wedge cf_i$ *before* $cf_j)$;
- dnf_1 *immediate succession* dnf_2: $\bigvee_{cf_i \in dnf_1, cf_j \in dnf_2} cf_i$ *succession* $cf_j \wedge \Box(cf_i \supset \mathsf{next}(cf_j))$.

The negative relations presented in Section II-B can be generalised to a DNF formula as follows:

- dnf_1 *not before* dnf_2: $\bigvee_{cf_i \in dnf_1, cf_j \in dnf_2} \mathsf{negation}(cf_i)\ \mathsf{U}\ (cf_j \wedge \mathsf{negation}(cf_i))$;
- dnf_1 *not immediate before* dnf_2: $\bigvee_{cf_i \in dnf_1, cf_j \in dnf_2} \mathsf{negation}(cf_i)\ \mathsf{U}\ (cf_j \wedge \mathsf{negation}(cf_i))$;
- dnf_1 *not implies* dnf_2: $\bigvee_{cf_i \in dnf_1, cf_j \in dnf_2} \mathsf{existence}(cf_i) \supset \mathsf{absence}(cf_j)$;
- dnf_1 *not immediate implies* dnf_2: $\bigvee_{cf_i \in dnf_1, cf_j \in dnf_2} \mathsf{existence}(cf_i) \supset (\mathsf{absence}(cf_j) \vee \Diamond(cf_i \wedge \mathsf{next}(\mathsf{negation}(cf_j))))$;
- dnf_1 *not succession* dnf_2: $\bigvee_{cf_i \in dnf_1, cf_j \in dnf_2} \mathsf{existence}(cf_i) \supset (\mathsf{absence}(cf_j) \vee cf_i$ *not before* $cf_j)$;
- dnf_1 *immediate succession* dnf_2: $\bigvee_{cf_i \in dnf_1, cf_j \in dnf_2} \mathsf{existence}(cf_i) \supset (\mathsf{absence}(cf_j) \vee cf_i$ *not before* $cf_j \vee \Diamond(cf_i \wedge \mathsf{next}(\mathsf{negation}(cf_j))))$.