Curricula Modeling and Checking

Matteo Baldoni, Cristina Baroglio, and Elisa Marengo

Dipartimento di Informatica — Università degli Studi di Torino
C.so Svizzera, 185 — I-10149 Torino (Italy)
{baldoni,baroglio}@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.

1 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 [17]. 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 [12]. 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 [10], 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 [17] 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 [8], 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".

R. Basili and M.T. Pazienza (Eds.): AI*IA 2007, LNAI 4733, pp. 471–482, 2007.

Together with the verification of consistence gaps, there are other kinds of verification. Brusilovsky and Vassileva [8] 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.

More specifically, 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 [11]) 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* [1]. We translate an activity diagram, that represents a curriculum, in a *Promela* program [16] and we check, by means of the well-known SPIN Model Checker [16], 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. As in [10], 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 [10], which will be part of future work.

This approach differs from previous work [5], 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 [18,19]. 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.

2 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 [21]. DCML, as well as DecSerFlow, is grounded in Linear Temporal Logic [11] 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 ($\alpha \mathsf{U} \beta$, the formula α remains true until β), see also [16, Chapter 6]. The set of LTL formulas obtained for a curricula model are, then, used to verify whether a curriculum will respect it [4]. As an example, 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.

2.1 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 [4,7] so

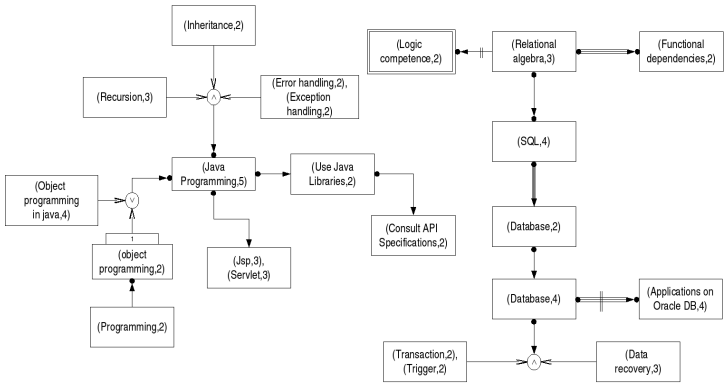

Fig. 1. An example of curricula model in DCML

as to include a representation of the *proficiency level* at which a competency 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*.

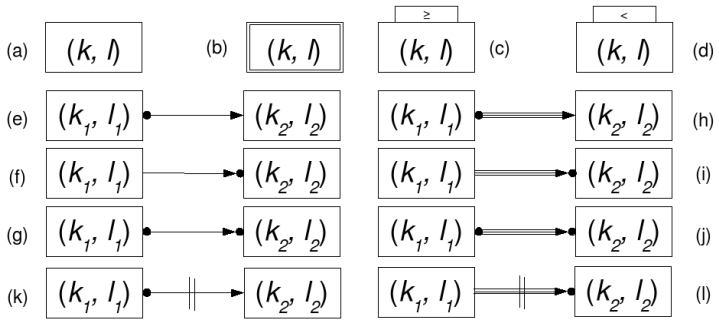

Fig. 2. Competences (a) and basic constraints (b), (c), and (d). Relations among competences: (a) implication, (b) before, (c) succession, (d) immediate implication, (e) immediate before, (f) immediate succession, (g) not implication, (h) not immediate before.

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 lack of space, we do not describe the notation here, however, an example can be seen in Fig. 1.

2.2 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 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)" $\vee\Diamond((k_1, l_1) \wedge \bigcirc\neg(k_2, l_2)))$.

3 Representing Curricula as Activity Diagrams

Let us now consider specific curricula. In the line of [5,3,4], 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. In our case, we represent courses as actions for acquiring some concepts (*effects*) if the user owns some competences (*preconditions*). So, a curriculum is seen as a sequence of actions that causes *transitions* from the initial set of competences (possibly empty) of a user up to a final state that will contain all the competences owned by the user in the end. 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 no new course erases from the students memory the concepts acquired in previous courses, thus knowledge grows incrementally. We represent

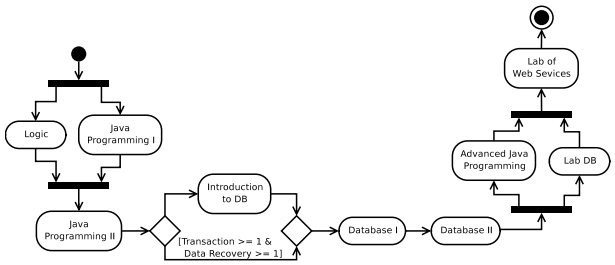

Fig. 3. Activity diagram representing a set of eight different curricula. Notice that *Logic* and *Java Programming I* can be attended in any order (even in parallel), as well as *Advanced Java Programming* and *Lab DB*, while *Introduction to DB* will be considered only if the guard *Transaction* and *Data Recovery* is false.

curricula as *activity diagrams* [1], normally used for representing *business processes*. We decided to do so, because they allow to capture in a natural way the simple sequencing of courses as well as the possibility of attending courses in *parallel* or in possibly conditioned *alternatives*. An example is reported in Fig. 3. Besides the initial and the final nodes, the graphical elements used in an activity diagram are: *activity nodes* (rounded rectangle) that represent activities (attending courses) that occur; *flow/edge* (arrows) that represent activity flows; *fork* (black bar with one incoming edge and several outgoing edges) and *join nodes* (black bar with several incoming edges and one outgoing edge) to denote parallel activities; and *decision* (diamonds with one incoming edge and several outgoing edges) and *merge nodes* (diamonds with several incoming edges and one outgoing edge) to choose between alternative flows.

In the modeling of *learning processes*, we use activities to represent attending courses (or reading learning resources). For example, by fork and join nodes we represent the fact that two (or more) courses or sub-curricula are not related

and, it is possible for the student to attend them in parallel. This is the case of *Java Programming I* and *Logic*, as well as *Advanced Java Programming* and *Lab. of DB* showed in Fig. 3. Till all parallel branches have not been attended successfully, the student cannot attend other courses, even if some of the parallel branches have been completed. Parallel branches can also be used when we want to express that the order among courses of different branches does not matter.

Decision and merge nodes can be used to represent alternative paths. The student will choose only one of these. Alternative paths can also be conditioned, in this case a *guard*, a boolean condition, is added at the beginning of the branch. Guards should be mutually exclusive. In our domain, the conditions are expressed in terms of concepts that must hold, otherwise a branch is not accessible. If no guards are present, the student can choose one (and only one) of the possible paths. In the example in Fig. 3, the guard consists of two copetences: *Transaction* and *Data Recovery*. If one of these does not hold the student has to attend the course *Introduction to DB*, otherwise does not.

4 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* [9].

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 [16], 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 [13,14]. However, these proposals have a different purpose than ours and they cannot be used to perform the translation that we need to perform the verifications we list above. 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 consists of two main processes: one is called *CurriculumVerification* and the other *UpdateState*. While the former contains the actual translation of the activity diagram and simulates the acquisition of the competences for *all* curricula represented by the translated activity diagram, the latter contains the code for updating the state, i.e. the competences achieved so far, according to the definition in terms of preconditions and effects of each course. The processes *CurriculumVerification* and *UpdateState* communicate by means of the channel *attend*. The notation *attend!courseName* represents the fact that the course with name "courseName" is to be attended. On the other hand, the notation *attend?courseName* represents the possibility for a process of receiving a message. For example, the process *CurriculumVerification* for the activity diagram of Fig. 3 is defined as follows:

```
proctype CurriculumVerification()
{ activity_forkjoin_1();
  course_java_programming_II();
  activity_decisionmerge_1();
  course_database_I();
  course_database_II();
  activity_forkjoin_2();
  course_lab_of_web_services();
  attend!stop; }
```

If the simulation of all its possible executions end, then, there are no competence gaps; *attend!stop* communicates this fact and starts the verification of user's learning goal, that, if passed, ends the process. Each *course* is represented by its preconditions and its effects. For example, the course "Laboratory of Web Services" is as follows:

```
inline preconditions_course_lab_of_web_services()
{ assert(N_tier_architectures >= 4 && sql >= 2); }
inline effects_course_lab_of_web_services()
{ SetCompetenceState(jsp, 4); [...]
  SetCompetenceState(markup_language, 5); }
inline course_lab_of_web_services()
{ attend!lab_of_web_services; }
```

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.

The *fork/join nodes* are simulated by activating as many parallel processes as their branches. Each process translates recursively the corresponding sub-activity diagram. Thus, SPIN simulates and verifies *all possible interleavings* of the courses (we can say that the curriculum is only one but it has different executions). The join nodes are translated by means of the synchronization message *done* that each activated process must send to the father process when it finishes its activity:

```
proctype activity_joinfork_11()
{ course_java_programming_I(); joinfork_11!done; }
proctype activity_joinfork_12()
{ course_logic(); joinfork_12!done; }
inline activity_joinfork_1()
{ run activity_joinfork_11(); run activity_joinfork_12();
  joinfork_11?done; joinfork_12?done; }
```

Finally, *decision and merge nodes* are encoded by either conditioned or non-deterministic *if*. Each such *if* statement refers to a set of alternative sub-activity diagrams (sub-curricula). Only one will be effectively attended but all of them will be verified:

```
inline activity_decisionmerge_11()
{ course_introduction_to_database(); }
inline activity_decisionmerge_12() { skip; }
inline activity_decisionmerge_1()
{ if
  :: (transaction >= 1 && data_recovery >= 1) ->
       activity_decisionmerge_12();
  :: else -> activity_decisionmerge_11();
  fi }
```

On the other hand, the process *UpdateState*, after setting the initial competences, checks if the preconditions of the courses communicated by *CurriculumVerification* hold in the current state. If a course is applicable it also updates the state. The test of the preconditions and the update of the state are performed as an atomic operation. In the end if everything is right it sends a feedback to *CurriculumVerification* (*feedback!done*):

```
proctype UpdateState() { SetInitialSituation();
  do [ ... ]
  :: attend?lab_of_web_services -> atomic {
        preconditions_course_lab_of_web_services();
        effects_course_lab_of_web_services(); }
  :: attend?stop -> LearningGoal(); break;
  od }
```

When *attend?stop* (see above) is received, the check of the user's learning goal is performed. This just corresponds to a test on the knowledge in the ending state:

```
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. For example, the curriculum shown in Fig. 3 respects all the constraints imposed by the curricula model described in Fig. 1, taking into account the description of the courses supplied at the URL above. The assertion verification takes very few seconds on an old notebook; the automaton generated from the Promela program on

that example has more than four-hundred states, indeed, it is very tractable. Also the verification of the temporal constraints is not hard if we check the constraints one at the time. The above example is available for download at the URL `http://www.di.unito.it/~baldoni/DCML/AIIA07`.

5 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 [21], 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 [20], 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 [20], 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 [6,3,5], 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 [2], we discuss the integration into the Personal Reader Framework [15] of a web service that implements an earlier version of the techniques explained here, which does not include proficiency levels.

Acknowledgements. The authors would like to thank Viviana Patti for the helpful discussions. This research has partially been funded by the 6th Framework Programme project REWERSE number 506779 and by the Italian MIUR PRIN 2005.

References

1. Unified Modeling Language: Superstructure, version 2.1.1. OMG (February 2007)
2. Baldoni, M., Baroglio, C., Brunkhorst, I., Marengo, E., Patti, V.: Curriculum Sequencing and Validation: Integration in a Service-Oriented Architecture. In: Proc. of EC-TEL'07. LNCS, Springer, Heidelberg (2007)
3. Baldoni, M., Baroglio, C., Henze, N.: Personalization for the Semantic Web. In: Eisinger, N., Małuszyński, J. (eds.) Reasoning Web. LNCS, vol. 3564, pp. 173–212. Springer, Heidelberg (2005)
4. Baldoni, M., Baroglio, C., Martelli, A., Patti, V., Torasso, L.: 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)
5. Baldoni, M., Baroglio, C., Patti, V.: Web-based adaptive tutoring: an approach based on logic agents and reasoning about actions. Artificial Intelligence Review 22(1), 3–39 (2004)
6. Baldoni, M., Baroglio, C., Patti, V., Torasso, L.: Reasoning about learning object metadata for adapting SCORM courseware. In: Proc. of Int.l Workshop EAW'04, at AH 2004, Eindhoven, The Netherlands, August 2004, pp. 4–13 (2004)
7. Baldoni, M., Marengo, E.: Curricula model checking: declarative representation and verification of properties. In: Proc. of EC-TEL'07. LNCS, Springer, Heidelberg (2007)
8. Brusilovsky, P., Vassileva, J.: Course sequencing techniques for large-scale web-based education. Int. J. Cont. Engineering Education and Lifelong learning 13(1/2), 75–94 (2003)
9. Clarke, O.E.M., Peled, D.: Model checking. MIT Press, Cambridge (2001)
10. De Coi, J.L., Herder, E., Koesling, A., Lofi, C., Olmedilla, D., Papapetrou, O., Siberski, W.: A model for competence gap analysis. In: Proc. of WEBIST 2007 (2007)
11. Emerson, E.A.: Temporal and model logic. In: Handbook of Theoretical Computer Science, vol. B, pp. 997–1072. Elsevier, Amsterdam (1990)
12. Farrell, R., Liburd, S.D., Thomas, J.C.: Dynamic assebly of learning objects. In: Proc. of WWW 2004, New York, USA (May 2004)
13. del Mar Gallardo, M., Merino, P., Pimentel, E.: Debugging UML Designs with Model Checking. Journal of Object Technology 1(2), 101–117 (2002)
14. Guelfi, N., Mammar, A.: A Formal Semantics of Timed Activity Diagrams and its PROMELA Translation. In: Proc. of APSEC'05, pp. 283–290 (2005)
15. Henze, N., Krause, D.: Personalized access to web services in the semantic web. In: The 3rd Int.l Workshop SWUI, at ISWC 2006 (2006)
16. Holzmann, G.J.: The SPIN Model Checker. Addison-Wesley, Reading (2003)
17. Melia, M., Pahl, C.: Automatic Validation of Learning Object Compositions. In: Proc. of *IT&T'2005: Doctoral Symposium*, Carlow, Ireland (2006)
18. Singh, M.P.: Agent communication languages: Rethinking the principles. IEEE Computer 31(12), 40–47 (1998)
19. Singh, M.P.: A social semantics for agent communication languages. In: Dignum, F.P.M., Greaves, M. (eds.) Issues in Agent Communication. LNCS, vol. 1916, pp. 31–45. Springer, Heidelberg (2000)
20. Terenziani, P., Giordano, L., Bottrighi, A., Montani, S., Donzella, L.: 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)
21. van der Aalst, W.M.P., Pesic, M.: DecSerFlow: Towards a Truly Declarative Service Flow Language. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, Springer, Heidelberg (2006)