UNIVERSITÀ DEGLI STUDI DI TORINO
FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
DIPARTIMENTO DI INFORMATICA
TESI DI LAUREA

Curricula di Studi: Modelli e Verifica di Competenze

Relatore
Prof. Matteo Baldoni

Controrelatore
Prof. Alberto Martelli

Candidata
Elisa Marengo

A.A 2007-2008

Indice

Introduzione

Il *World Wide Web* ha indubbiamente cambiato il modo in cui le persone utilizzano il computer. Inizialmente il loro ruolo era quello di sostituire l'uomo in lunghe e noiose operazioni e possederne uno era privilegio di pochi. Oggi, quasi in ogni casa, se ne può trovare uno che, nella maggior parte dei casi, viene usato per comunicare con altre persone, per fare acquisiti on-line o per la ricerca di informazioni sul Web. Quest'ultimo aspetto ha causato la rapida crescita del numero e del tipo di risorse fruibili, ma mentre il Web "di ieri" era un universo di informazione accessibile via rete fatto dalle persone per le persone, già da alcuni anni ci si sta muovendo verso un nuovo Web in cui le macchine sono in grado di operare sempre di più in modo automatico.

A tal proposito, si sente spesso parlare di *Semantic Web*, il cui obiettivo è automatizzare in buona parte l'elaborazione delle informazioni. Non si tratta, quindi, di qualcosa di diverso, separato o aggiunto a posteriori al Web "di ieri". Gli esperti preferiscono definirlo come l'*efficace completamento e la naturale conseguenza* [23]. Più concretamente il Semantic Web si propone di strutturare il contenuto delle pagine Web, creando le condizioni per cui agenti e programmi software possano utilizzarle e portare a termine task per conto degli utenti. Tim Berners-Lee propone la seguente definizione:

> *The Semantic Web is not a separate Web but an extension of the current one, in which information is given well-defined meaning, better enabling computers and people to work in cooperation.*

In quest'ottica le applicazioni che vengono sviluppate si propongono di utilizzare meccanismi automatici di ragionamento orientati all'utente, che permettano, cioè, di specificare quello che si sta cercando in modo sufficien-

temente preciso da consentire il risparmio di tempo con la certezza di trovare tutte le possibili e migliori alternative.

Tutti questi aspetti si adattano benissimo al particolare dominio dell'*e-learning*, che rappresenta l'ambito di riferimento di questa tesi. Esso si occupa dell'"apprendimento centrato sull'utente, che coinvolge elaboratori e reti di comunicazione". Un notevole impulso in questa direzione si ha intorno al 1999 quando l'apprendimento a distanza via Web viene adottato dalle aziende, consentendo loro di abbattere i costi per la formazione del personale e facilitando le attività di aggiornamento dei contenuti che, prima di allora, venivano distribuiti in modo cartaceo o su CD-ROM.

Lo scopo di questa tesi è quello di fornire uno strumento per la personalizzazione e la verifica di *curricula di studi*. Più precisamente si discuterà un approccio per la definizione e per il controllo dei vincoli che un certo piano di studi deve rispettare. Questi verranno espressi in termini di conoscenze acquisite. In particolare, si tratta di controllare che un piano non presenti *competency gaps* (conoscenze richieste in ingresso ad un corso che lo studente non possiede), che permetta all'utente di raggiungere il suo *learning goal* e che le relazioni e i vincoli temporali imposti sull'acquisizione delle competenze, definiti nel *curricula model*, siano rispettati.

A tal proposito, verranno descritti un linguaggio grafico basato sulla logica LTL (Linear Temporal Logic), per la definizione di constraints tra le competenze ed una rappresentazione dei curricula per mezzo di *activity diagram* UML. La tecnica adottata per realizzare la verifica, prevede l'utilizzo del model cheker SPIN. Pertanto, il piano di studi verrà tradotto in un programma PROMELA e su di esso verranno verificate le varie formule LTL, rappresentanti le proprietà d'interesse.

Il presente lavoro si articola in otto capitoli organizzati in tre parti:

Parte I: descrive una panoramica sullo stato dell'arte.

Nel *primo capitolo* si sottolinea l'importanza di riutilizzare risorse, learning objects nello specifico, in ambienti aperti e dinamici come il Semantic Web. Tuttavia, a questo aspetto deve essere affiancato quello della personalizzazione: l'utente deve essere messo in condizione di

esprimere quello che desidera ottenere e, nel caso sia possibile, deve essere accontentato.

Il *secondo capitolo* tratta la composizione dei learning object per la definizione automatica di corsi. In queste proposte emergono alcuni vincoli che devono essere rispettati, quali il soddisfacimento dei prerequisiti di un certo corso, o la valutazione dell'effettivo apprendimento dello studente.

Nel *terzo capitolo* si presenta un possibile approccio per la verifica di vincoli di fruizione. Si tratta di condizioni legate alla struttura di un curricula di studi, più che al contenuto. Essi si riferiscono, ad esempio, al numero di crediti acquisito, al numero di corsi, al tipo e al numero di corsi per ogni periodo, etc.

Parte II: affronta il problema di come descrivere vincoli sulle competenze e di quali relazioni tra queste abbia senso definire.

Nel *quarto capitolo* si presenta il paradigma ad azioni per la rappresentazione dei corsi. In quest'ottica, essi vengono visti come azioni descritte in termini di precondizioni ed effetti. Su questa idea si basa il sistema *Wlog* [9]. Esso si occupa della pianificazione di curricula di studi, utilizzando il *procedural planning* e presentata una soluzione per la verifica dei piani, mediante tecniche di *temporal projection reasoning*. La soluzione dei problemi riscontrati in tale approccio, rappresentano l'ambizioso obiettivo di questa tesi.

Nel *quinto capitolo* si definisce DCML, un linguaggio grafico per la rappresentazione di vincoli e relazioni tra le competenze. L'insieme di tali condizioni definisce il *curricula model*, cioè l'insieme di proprietà che un curricula deve rispettare per definirsi corretto. Ad ogni vincolo esprimibile graficamente è associata una traduzione in formule logiche LTL.

Parte III: presenta una soluzione per la rappresentazione e la traduzione di curricula di studi.

Il *sesto capitolo* si occupa di definire come un curricula di studi pos-

sa essere “disegnato”, in modo semplice e intuitivo, per mezzo degli activity diagrams UML.

Il *settimo capitolo* affronta il problema della traduzione dei curricula, dalla rappresentazione grafica al linguaggio PROMELA. Segue la descrizione di come venga affrontata la verifica dei competency gaps, del raggiungimento del learning goal e dei vincoli descritti nel curricula model, ad opera del model checker SPIN.

Nel *capitolo conclusivo* vengono riportate alcune considerazioni su come la funzionalità della verifica di curricula di studi possa essere inserita all'interno di un framework accessibile via web, e su come sia possibile migliorare la definizione delle risorse mediante l'utilizzo di opportune ontologie.

Parte I

Stato dell'arte

Capitolo 1

Il ruolo dei Learning Objects nel Semantic Web

La rapida evoluzione delle tecnologie *ICT* [1] ed il sempre crescente utilizzo del Web ha fortemente influenzato il modo in cui le persone desiderano apprendere, tanto che oggi anche nel settore della scuola e dell'istruzione non è più possibile ignorare la loro esistenza [47]. In questo ambito si parla di *e-learning*, il cui successo, tuttavia, è da leggere principalmente in chiave economica, infatti permette l'abbattimento dei costi di formazione, facilita l'aggiornamento del materiale ed il personale apprende in meno tempo e senza doversi allontanare dal luogo di lavoro.

La diffusione ed il successo dell'e-learning trovano un alleato nel *Semantic Web* [1]. Esso rappresenta l'estensione del Web mediante l'aggiunta di un livello semantico alle risorse in modo da permettere a computer e persone di cooperare.

Il *World Wide Web Consortium*(W3C)[2] dà la seguente definizione:

> *The Semantic Web provides a common framework that allows data to be shared and reused across application, enterprise and community boundaries.*

Uno degli obiettivi del Semantic Web, che si adatta benissimo anche

[1] ICT: Information and Communication Technology

[2] http://www.w3c.org

in ambito e-learning, è quello di condividere e riutilizzare dati attraverso applicazioni, imprese e comunità.

Possiamo così distinguere tra due attori principali: il produttore ed il consumatore. Il primo si occupa di definire nuove risorse e materiale che possano essere usati, composti e organizzati dal consumatore al fine di raggiungere un obiettivo. Al secondo, quindi, deve essere data la possibilità di personalizzare il proprio percorso di studi sulla base degli obiettivi che intende raggiungere, dei metodi e dei tempi di apprendimento.

Questo significa innanzitutto assisterlo nella ricerca del materiale, rischierebbe altrimenti di essere disorientato dalla grandissima quantità di informazione accessibile attraverso il Web. Lo scopo principale è fare in modo che l'utente possa trovare facilmente ed in tempi brevi quello che cerca, così da poter impiegare il tempo risparmiato nella ricerca per l'apprendimento e per la consultazione del materiale.

A questo scopo un importante punto di partenza è sicuramente la definizione esauriente e rigorosa di tutto ciò che viene messo a disposizione sul Web, affinché possa essere utilizzato per comporre in modo automatico un corso, un piano di studi e quant'altro.

Per questo sono stati definiti i *Learning Objects*.

1.1 Learning Object e l'importante tema del "riuso"

Il termine "learning object" è stato introdotto a metà degli anni '90 con lo scopo di suddividere qualsiasi tipo di "learning material" in componenti più piccole che potessero essere messe a disposizione di chiunque e potessero essere ricombinate anche in modo automatico.

> *Un "Learning Object" è il più piccolo elemento stand-alone di informazione richiesto ad un individuo per raggiungere un certo obiettivo o risultato* [21].

Più precisamente, in ambito e-learning, è una qualsiasi risorsa digitale che possa essere riutilizzata per supportare l'apprendimento (un testo, un'ani-

mazione, un file audio o qualunque altra forma di apprendimento passivo o interattivo).

Per migliorare il riutilizzo delle risorse occorre che la definizione dei Leraning Objects sia regolata da standard, il cui compito è quello di facilitarne la ricerca, lo scambio e la gestione.

L'importanza di definire tali standard viene sottolineata da Mohan e Brooks [39]. Essi propongono una visione in cui è sufficiente specificare un insieme di *learning outcomes* per ottenere come risultato i learning objects (eventualmente con alternative tra cui l'utente può scegliere) che permettono di raggiungere l'obiettivo desiderato. In un certo senso è come se si desse al learning object maggiore "responsabilità", in modo che esso stesso sia in grado di stabilire se è adatto a fornire una certa competenza o meno. Al momento questa resta solo un'idea ma l'utilizzo di ontologie e metadati per descrivere gli oggetti rappresenta sicuramente un buon inizio, in quanto permette di fare ricerche più accurate e di selezionare solo elementi interessanti rispetto al fine che si vuole raggiungere. Quindi, il livello semantico che si aggiunge serve per far sì che ogni oggetto sia in grado di descrivere il suo contributo. In questo modo si riducono i tempi necessari al designer per comporre le risorse.

L'utilizzo di ontologie è importante in quanto permette di specificare come un learning object sia legato ai concetti di un particolare dominio, risolvendo i problemi che si vengono a creare quando sistemi distinti usano definizioni diverse per identificare la stessa risorsa. Esse vengono utilizzate per definire formalmente le relazioni tra i termini. Questo permette ad un agente di "capire" se un determinato learning object sia la soluzione migliore da adottare in un certo corso, confrontando gli outcomes forniti dall'oggetto e quelli richiesti nel corso. L'ontologia serve a fare in modo che il confronto avvenga utilizzando lo stesso linguaggio.

La ricchezza d'informazione associata ad un learning object se da un lato costituisce un vantaggio notevole per la ricerca e la composizione, dall'altro complica l'aspetto di definizione dell'oggetto stesso. In particolare occorre definire metadati, link ad ontologie che ne permettano il riutilizzo in contesti diversi e link ad altri learning objects con cui sono in relazione.

La proposta descritta in [39] prevede che l'autonomia dei learning objects permetta loro di rispondere ad interrogazioni sul contenuto e di cercarne altri, attraverso il Web, in relazione con essi. Questo è quello che viene definito *contributo attivo* dei learning objects alla ricerca.

L'implementazione proposta prevede l'utilizzo di *Object-Oriented Learning Objects*, quindi, in accordo con la filosofia Object Oriented, ogni LO[3] ha uno stato ed un comportamento, determinati rispettivamente da variabili e metodi. Le variabili permettono di definire i link necessari per metterli in relazione con altre risorse e con le ontologie che ne descrivono il contenuto. I metodi servono per verificare la correttezza dei link e per poter rispondere alle interrogazioni che vengono fatte all'oggetto. In questo modo, ogni risorsa è in grado di stabilire se il suo impiego in una certa posizione sia appropriato o meno.

Una caratteristica fondamentale, che motiva l'esistenza e l'importanza dei learning object, è che sono indipendenti tra loro e da altre risorse. Questo è il vero punto di forza che li rende effettivamente riutilizzabili e che consente di condividerli con altri[4].

Il tema del "riutilizzo" è spesso usato in diversi ambiti riferito a risorse e software. Si pensi, ad esempio, ai Web Services il cui scopo è quello di offrire servizi invocabili da altre applicazioni (non solo da utenti) in modo automatico e del tutto trasparente all'utilizzatore finale. In questo senso il Semantic Web è visto come *service provider* e non come *information provider*, come piattaforma per condividere risorse e servizi [6]. Spesso questo aspetto viene descritto con l'analogia "del supermercato", dove l'utente può comprare servizi complessi che vengono composti dinamicamente a partire da altri più semplici.

Ma perché questo tema è così importante? Non si possono definire nuove risorse ad ogni uso? Per rispondere si ricorrere ad una delle metafore più conosciute che è quella dei LEGO [21]: quello che stupisce è che qualsiasi mattoncino LEGO, di qualsiasi colore, dimensione o forma possa essere as-

[3]Learning Object

[4]La condivisione può essere semplicemente la memorizzazione in un database accessibile liberamente attraverso internet.

semblato con altri mattoncini. Questo è dovuto al fatto che la dimensione delle componenti che si incastrano è standard.

L'idea di riutilizzare LOs è proprio questa: "incastrare" elementi definiti in altre situazioni, da altre persone e per altri scopi, al fine di costruire strutture più complesse. È evidente che incastrare mattoncini esistenti richiede meno tempo ed energie rispetto a costruirne di propri. Un effetto secondario, ma molto interessante, che deriva dal processo di riutilizzo è che in questo modo le risorse vengono testate e "collaudate" in situazioni e per scopi diversi.

Per rendere la metafora applicabile al settore dell'e-learning dobbiamo precisare che il nostro obiettivo non è che i mattoncini siano combinabili in qualsiasi modo: per la composizione di LOs esistono dei vincoli che devono essere rispettati. Inoltre il loro utilizzo non deve essere casuale ma deve essere orientato al raggiungimento di obiettivi precisi.

La struttura di un learning object comprende:

Learning Object. Il learning object stesso, cioè il contenuto che si vuole rappresentare.

Metatagging. Consiste nell'aggiungere all'oggetto un tag che ne descrive il contenuto e permette di indicizzare il documento. Grazie all'indicizzazione ne beneficia sia la ricerca finalizzata alla composizione, sia quella condotta dall'utente con lo scopo di reperire informazioni riguardanti un particolare dominio.

Learning Content Management System. Un LCMS è necessario per la memorizzazione, l'indicizzazione dei contenuti e per fornire meccanismi di ricerca. Distinguiamo tra LMS (Learning Management System) e CMS (Content Management System)[5]. Il primo è un sistema usato nelle aziende o nelle Università per semplificare la gestione di un corso; in particolare per la distribuzione e l'organizzazione del materiale. Inoltre, fornisce un meccanismo di comunicazione tra utenti e permette loro di decidere il proprio percorso. In alcuni casi LMSs forniscono meccanismi per la definizione di nuovi learning object.

[5]Alcuni esempi di CMS sono TopClass, BlackBoard e webCT.

I CMS, invece, vengono usati per organizzare contenuti disponibili online, ad esempio per gestire articoli.

L'unione di LMS e CMS dà origine ai LCMS, i quali permettono sia la definizione di nuovi learning object sia la loro composizione[6].

Nella fase di definizione di un learning object occorre prima di tutto stabilire la granularità rispetto alla quale si vogliono esprimere i vari concetti. Questo è legato al principio di *modularizzazione*, che consiste nel dividere le varie componenti in gruppi, facilitando il mantenimento complessivo del sistema e la modifica di alcune sue parti. Ad esempio, se si cerca di definire le componenti di un corso sul linguaggio di programmazione Java occorre scegliere se i diversi tipi di loop debbano essere definiti in un unico learning object, oppure se debbano essere considerati come distinti. In quest'ultimo caso professori diversi possono cambiare l'ordine con cui vengono presentati e possono essere introdotti con livelli di dettaglio diversi a seconda dell'utente che ha richiesto quella conoscenza. Inoltre, learning object più piccoli rendono più facile il riutilizzo in contesti diversi. In generale più il learning object è grande[7] più le sue possibilità di utilizzo saranno limitate, in quanto è più difficile che concordi con altri oggetti e con le esigenze di altri insegnati o altri studenti.

L'IEEE definisce quattro livelli di granularità:

smallest level. È il livello più basso. È il caso in cui ogni competenza viene rappresentata in un LO distinto.

level 1. Alcuni LOs vengono aggregati arrivando a definire, ad esempio, il contenuto di una lezione.

level 2. Il livello di aggregazione aumenta ulteriormente arrivando a rappresentare addirittura un intero corso.

largest level. In questo livello, le diverse componenti assumono dimensioni

[6]Con il termine "composizione" si intende la costruzione di una sequenza composta da più learning object precedentemente selezionati.

[7]Per grande si intende la rappresentazione di un aggregato di componenti che potrebbero essere separate e diventare indipendenti.

notevoli, arrivando a comprendere il contenuto di un insieme di corsi che permettono il raggiungimento di una qualche certificazione.

In accordo con quanto detto fino ad ora e con questa suddivisione in livelli, anche un intero curriculum potrebbe essere visto come un learning object. Quello che viene meno, e che rappresenta un punto fondamentale in ottica di e-learning, è la possibilità di essere riutilizzato. Lo scopo, infatti, è quello di soddisfare le esigenze di ogni singolo utente. Questo risultato può essere raggiunto in modo tanto più vantaggioso quanto più si riescono a combinare elementi già definiti da altri in precedenza. Quindi anziché costruire learning object di grosse dimensioni (per questo destinati a soddisfare un singolo utente o poco più) si può ottenere lo stesso risultato definendo e componendo learning object più piccoli e, in un certo senso, più specifici. Questi a loro volta potranno essere riutilizzati in altri contesti.

1.2 "Personalizzazione" nel Semantic Web

Accanto al tema del riuso di risorse se ne deve considerare un altro altrettanto importante: quello della personalizzazione.

Si tratta di un processo mediante il quale è possibile fornire all'utente un supporto nell'accedere, nel ricercare e nel memorizzare informazioni. Le soluzioni devono essere costruite in modo da soddisfare le esigenze, i gusti, le preferenze e le caratteristiche di ogni singolo utente.

Perché questo processo sia possibile occorre che le informazioni e le risorse siano elaborabili in modo automatico. Questo è il motivo per cui si sente parlare di personalizzazione soprattutto in ambito di Semantic Web. Infatti, ciò non sarebbe realizzabile nel Web perché alle risorse non viene aggiunto alcun livello semantico, dunque queste risultano utilizzabili quasi esclusivamente dalle persone.

Come sottolineato dagli autori in [6], perché sia possibile una qualsiasi forma di personalizzazione occorre che il sistema possieda una qualche rappresentazione dell'utente. Questo prende il nome di *user model* (modello utente). L'obiettivo, infatti, non è quello di costruire una soluzione, ma è quello di costruire una soluzione per l'utente. In ambito e-learning non

avrebbe senso assemblare learning objects al fine di costruire corsi che non soddisfano le esigenze dell'utilizzatore finale. In questo caso si dice che la ricerca della soluzione è *goal-driven*. Restando in questo ambito, si possono individuare due punti di vista rispetto ai quali il concetto di personalizzazione cambia: quello dello studente e quello dell'insegnante. Nel primo caso il learning goal definisce quali conoscenze devono essere acquisite. Il piano deve quindi contenere la sequenza di risorse che portano lo studente a raggiungerlo. Nel caso dell'insegnante, invece, cambia lo scopo: la ricerca è orientata a trovare del materiale per poter integrare un corso o per definirne uno nuovo.

Questo concetto può essere riassunto con il termine "*ruolo*" che deve essere tenuto in considerazione dal sistema, ad esempio per presentare viste diverse (in questo caso si parla di personalizzazione a livello di presentazione). Inoltre, cambiano le azioni che gli utenti possono compiere ed il materiale consultabile.

Il compito del Semantic Web è quello di facilitare l'accesso alle risorse giuste. Si tratta, quindi, di risolvere due sottoproblemi, quello della *ricerca*, per cui si devono trovare o comporre le risorse in modo da permettere il raggiungimento dell'obiettivo finale, e quello della *presentazione*, per visualizzare i risultati ottenuti nel miglior modo possibile. Il concetto di "modello utente" interviene attivamente in queste fasi in quanto è l'aspetto che guida la personalizzazione.

I sistemi più conosciuti sono gli *Adaptive Hypermedia Systems* (AHS) [7].

1.2.1 *Adaptive Hypermedia Systems*

Gli AHS sono sistemi che realizzano una forma di personalizzazione che manca ai classici Hypermedia Systems. In particolare, ogni utente possiede una propria vista che ne guida la navigazione:

> *By adaptive hypermedia systems we mean all hypertext and hypermedia systems which reflect some features of the user in the user model and apply this model to adapt various visible aspects of the system to the user.* [26].

Nella maggior parte dei casi si tratta di semplici documenti legati tra loro da links. La particolarità è che in questi sistemi lo user model raccoglie informazioni (conoscenze, goals, esperienze, etc.) che vengono usate attivamente per adattare il contenuto e la navigazione. Ad esempio, ad un utente che abbia una scarsa conoscenza su un certo argomento verranno consigliate letture introduttive prima di addentrarsi in altre più specifiche. Al contrario, ad un esperto verranno presentati subito i documenti che entrano maggiormente nel dettaglio. Quindi, il compito del sistema è quello di presentare l'informazione giusta al momento giusto.

Per farlo si utilizzano diverse tecniche. Una di queste consiste nel disattivare ed attivare i link a seconda che l'utente possieda o meno le conoscenze per comprendere i contenuti della pagina al quale è collegato. Un'altra soluzione è quella di utilizzare la metafora del semaforo: rosso ad indicare un link vietato in quanto l'utente non possiede tutte le precondizioni necessarie, giallo se il link è sconsigliato dal sistema e verde se è consigliato.

L'utilizzo di questi strumenti è tanto più vantaggioso quanto maggiore è il numero di utenti da gestire, ognuno con proprie caratteristiche ed obiettivi.

Esempi di Adaptive Hypermedia Systems sono *ELM-ART*, *INTERBOOK* e *KBS hyperbook system*.

ELM-ART

In questo sistema [15] i concetti sono legati tra loro da prerequisiti ed effetti formando così una rete concettuale. Esso non produce una sequenza di letture da consigliare all'utente, bensì ad ogni passo suggerisce quale sia la prossima pagina da visitare: *next best page*, calcolata considerando i prerequisiti richiesti e le pagine consultate in precedenza dall'utente.

Per la gestione dei link viene usata la metafora del semaforo.

Il successore è ELM-ART II [48]. Le novità introdotte riguardano la rete dei concetti, che viene trasformata in una rete gerarchica organizzata in lezioni, sezioni, sottosezioni e pagine. Un bottone *next best* aiuta l'utente suggerendogli la prossima pagina da visitare.

INTERBOOK

INTERBOOK [14] fa uso di un modello utente e di un modello del dominio. Sulla base di quest'ultimo viene costruito un glossario che rappresenta la struttura didattica del dominio: ogni voce corrisponde ad un concetto e rimanda, mediante un collegamento, ad ogni sezione del libro che tratta quell'argomento.

Anche in questo caso si utilizza la metafora del semaforo. In breve il funzionamento è il seguente:

- Il sistema calcola un punteggio che indica il livello di conoscenza per ogni concetto. Sulla base di questo stabilisce se una conoscenza possa ritenersi acquisita o meno.

- Dopo di che, stabilisce quali pagine debbano essere suggerite e quali, invece, siano vietate in quanto non sono stati ottenuti tutti i prerequisiti necessari.

- Infine, il sistema sceglie una pagina tra quelle che introducono argomenti nuovi e per le quali non ci sono precondizioni insoddisfatte.

KBS Hyperbook System

In questo sistema si distinguono due livelli tenuti separati: quello delle informazioni riferite alla visualizzazione attuale, cioè legate ad un certo utente, e quello delle conoscenze. Quest'ultime sono memorizzate nel *knowledge model*, che mantiene le informazioni riguardanti precondizioni ed effetti di un determinato concetto. Le dipendenze tra le varie conoscenze vengono espresse nella forma $k_1 < k_2$ ad indicare che k_1 deve essere appreso prima di k_2.

KBS Hyperbook System [26] permette di trovare una sequenza di risorse che consenta all'utente di raggiungere in più passi il learning goal da lui desiderato. Questa è la differenza con ELM-ART, che propone una pagina alla volta.

Viene generato un glossario ed alle varie voci vengono legati esempi, unità correlate all'argomento in questione ed eventualmente pagine di altri libri elettronici disponibili in rete. Aumenta così l'indipendenza dalla provenienza

Caratteristiche	Conoscenza	Obiettivi	Esperienza	Velocità di apprendimento
ELM-ART	X		X	
INTERBOOK	X	X	X	
KBS	X	X		X

Tabella 1.1: Caratteristiche dell'utente prese in considerazione dal sistema

delle informazioni e dal loro formato (si possono combinare pagine HTML reperite dalla rete e pagine interne al sistema).

Come si può notare dalla Tabella 1.1, che definisce quali caratteristiche dell'utente vengano prese in considerazione dai vari sistemi durante la fase di pianificazione, KBS hyperbook system è l'unico ad utilizzare la velocità di apprendimento. L'utente, infatti, può specificare cosa e quanto apprendere al passo successivo. Se il sistema constata, proponendo degli esercizi, che il livello raggiunto non è soddisfacente allora presenta allo studente esempi simili che contengano poche informazioni nuove. Viceversa se il livello raggiunto è buono, il sistema permette di procedere velocemente addentrandosi in nuovi argomenti come richiesto.

Capitolo 2

Course Sequencing per la composizione di corsi

Fino ad ora si è posta particolare attenzione ai vantaggi che derivano dalla personalizzazione, dal riuso delle risorse ed all'importanza di descriverle in modo tale da poter effettuare ricerche rapide ed efficienti.

Tuttavia, nella maggior parte dei casi la composizione dei learning object è un procedimento fatto manualmente dal designer. In letteratura si trovano alcune proposte di come questo compito possa essere automatizzato. Non solo ma in [16, 24] vengono proposte tecniche che permettono di stabilire se in un dato momento lo studente abbia effettivamente acquisito le conoscenze che dovrebbe avere. Se così non fosse, si innesca un meccanismo di ripianificazione del piano di studi.

Brusilovsky e Vassileva in [16, 17] descrivono l'applicazione del *Course Sequencing* nel settore large-scale web-based education, come strumento per la generazione di corsi personalizzati. In particolare vengono proposti due modelli: DCG e CoCoA.

Il *Course Sequencing* è una delle tecnologie più utilizzate nell'ambito dell'*ITS (Intelligent Tutoring Systems)*; consiste nel generare corsi personalizzati per lo studente, facendo in modo che nella composizione, in ogni momento, si selezioni la tecnica di insegnamento[1] migliore, dove per migliore

[1] Alcune tecniche di insegnamento potrebbero essere: presentazione, esempio, domanda e problema.

si intende quella che porta l'utente più vicino al suo obiettivo finale. Questo meccanismo utilizza un modello di studente che considera le sue conoscenze iniziali ed il suo learning goal. Su di esso viene simulata la sequenza di apprendimento, associando ad ogni concetto il livello a cui è stato acquisito. Gli autori sottolineano l'importanza ed i vantaggi di riutilizzare le risorse in questo processo, attingendo da insiemi di *Domain Knowledge Elements*[2] (DKEs) precedentemente definiti.

Tuttavia, l'utilizzo del course sequencing presenta dei problemi quando il numero delle risorse aumenta, quindi in ambito di "large-scale Web-based education" dove è necessario gestire da decine a centinaia di corsi per migliaia di studenti raggruppati in classi. Il problema principale è quello del mantenimento delle risorse e per questo ci si affida a sistemi come i CMS (Content Management Systems) già menzionati in precedenza. Questi sistemi, però, nella maggior parte dei casi non offrono la possibilità di applicare il *dynamic sequencing*, ma permettono la sola costruzione di sequenze statiche, dove le preferenze dello studente guidano solo in parte la composizione. Un approccio di tipo dinamico, invece, è in grado di adattarsi meglio alle esigenze dell'utente in quanto compone le risorse "al volo", presentando allo studente un'attività alla volta, valutando i risultati e procedendo di conseguenza nella pianificazione della sequenza.

È possibile sfruttare queste tecniche anche in contesti "large-scale" ed in particolare in [16] vengono presentati tre approcci.

Course Sequencing e CMS. Il primo approccio suggerisce di utilizzare il course sequencing nei sistemi CMS. L'idea è che, poiché questa tecnica è in grado di pianificare sequenze scegliendo in ogni momento l'attività migliore per lo studente, allora può essere utilizzata per valutare una sequenza data, determinando in ogni istante se il passo successivo, scelto dall'autore, sia o meno il migliore possibile. In questo modo è possibile trovare degli errori nella sequenza, ad esempio perché ad un certo punto si richiedono conoscenze che non sono ancora state fornite.

La semplicità di questa tecnica consente di applicarla a qualsiasi corso

[2]Questo termine viene usato in modo generico per indicare qualsiasi concetto, conoscenza, argomento, learning objects, learning outcome, etc.

o sequenza esistente. Lo svantaggio è che i benefici della pianificazione dinamica, che offre la possibilità di adattare il corso alle esigenze di un particolare studente, non vengono applicati. Questa tecnica viene utilizzata nel sistema CoCoA (descritto nella Sezione 2.2).

Adaptive Courseware Generation. Questo secondo approccio mette in evidenza l'importanza di comporre corsi riutilizzando materiale esistente, permettendo così l'abbattimento dei tempi per la definizione delle risorse. La personalizzazione viene realizzata fornendo ad utenti diversi versioni differenti dello stesso corso.

La generazione non avviene in modo incrementale ma in un unico passo. Si tratta quindi di una definizione statica della sequenza e per questo può essere gestita da un CMS. Inoltre, rappresenta un modo per fornire efficacemente corsi personalizzati per un gruppo omogeneo di studenti considerando le conoscenze iniziali ed i learning goals complessivi. Ovviamente, però, il livello di personalizzazione non è elevato come quello che si ottiene con tecniche incrementali.

Questo approccio viene utilizzato nel sistema DCG descritto nella Sezione 2.1.

Dynamic Courseware Generation. Come nell'adaptive courseware generation, l'obiettivo è quello di generare corsi sulla base del learning goal e delle conoscenze iniziali dello studente. Questo si realizza in modo dinamico adattando e modificando la sequenza secondo l'apprendimento dello studente. Vengono, quindi, proposti dei test in determinati punti e se i risultati sono inferiori a quelli attesi ha inizio il processo di ripianificazione.

Rispetto alle due tecniche precedenti, quest'ultima permette di ottenere livelli di personalizzazione superiori in quanto considera quanto è stato effettivamente appreso dallo studente.

DCG permette questo tipo di composizione.

Di seguito verranno presentati i due sistemi: DCG e CoCoA. L'obiettivo degli autori è quello di mostrare come le tecniche appena descritte possano essere utilizzate e sfruttate all'atto pratico.

2.1 DCG: Dynamic Course Generation System

DCG è un sistema che permette di generare e pianificare corsi appositamente per un utente. In particolare consente allo studente di raggiungere il suo learning goal tenendo in considerazione le sue conoscenze iniziali, il modo e la velocità di acquisizione dei concetti. Per farlo si simula il processo di apprendimento per mezzo di un modello (uno per ogni studente) che tiene traccia delle conoscenze che vengono via via acquisite.

Durante la fase di apprendimento vengono proposti dei test, se i risultati ottenuti sono inferiori alle attese significa che lo studente non ha raggiunto il livello necessario per proseguire nella sequenza. È inevitabile, quindi, la ripianificazione del corso in modo da colmare le lacune.

Il dominio viene rappresentato mediante un grafo in cui i nodi sono i concetti e le varie relazioni tra essi vengono espresse dagli archi. Per esprimere, ad esempio, che due conoscenze α e β sono entrambi necessari per definire γ si rappresenta quest'ultimo in un nodo con due archi uscenti, uno verso α ed uno verso β, in relazione di AND tra loro. Se non si specifica alcuna relazione tra questi, allora α e β vengono considerati in alternativa tra loro. Ad ogni arco può, quindi, essere associata una semantica di aggregazione (α e β *insieme* definiscono il concetto γ), di generalizzazione (γ è un concetto generale le cui possibili istanze sono α *oppure* β), causale, temporale e così via.

La costruzione del grafo può avvenire partendo da porzioni indipendenti tra loro successivamente composte in un'unica struttura che rappresenta l'intero dominio. Inoltre, grazie alle relazioni che gli archi identificano, è possibile leggere le diverse conoscenze a livelli di dettaglio diversi (aumentando o diminuendo il grado di aggregazione).

Ad ogni nodo viene associato un metodo di insegnamento (test, esercizio, presentazione, motivazione, esempio, etc.). La componente che si occupa della pianificazione non è in grado di stabilire quale sia la tecnica di insegnamento migliore per ogni concetto. Per questo è necessaria un'altra componente il cui compito è appunto quello di definire il *presentation plan*. Il grafo utilizzato è molto simile a quello definito per la rappresentazione delle conoscenze con la differenza che in questo caso i nodi rappresentano

metodi di insegnamento e gli archi definiscono il modo in cui questi metodi possono essere scomposti.

Per selezionare una tecnica di insegnamento si usano le *Teaching Rules*, che sono in grado di selezionare un metodo tenendo in considerazione lo stile di apprendimento dell'utente. Nella maggior parte dei casi si tratta di regole generiche e quindi indipendenti dal dominio.

Se durante la presentazione del corso lo studente risponde ai test in modo soddisfacente, allora nessun cambiamento deve essere attuato. Se invece i test mostrano che il concetto non è stato appreso come avrebbe dovuto, si effettua una ripianificazione. In un primo momento si cerca di modificare il metodo con cui il concetto è stato presentato. Se questo non è sufficiente si interviene ripianificando anche i contenuti del corso, partendo dalle conoscenze in quel momento in possesso dello studente e costruendo una sequenza che gli permetta di colmare le lacune.

Questa modalità di pianificazione è detta **dynamic planning** proprio perché è in grado di tener conto dell'effettivo apprendimento dei concetti da parte dell'utente. Tuttavia, il sistema è in grado di escludere questa possibilità operando in modalità **one-shot**: la pianificazione avviene considerando il learning goal dello studente e le sue conoscenze iniziali, una volta generato il corso, però, non si possono effettuare modifiche.

2.2 CoCoA: Concept-based Courseware Analysis

CoCoA è un sistema CMS (Content/Course Management System) sviluppato al Carnegie Technology Education, Carnegie Mellon University. Si occupa della verifica della consistenza e qualità dei corsi ed assiste lo sviluppatore in alcune operazioni di routine. Per fare ciò si sfruttano le tecniche di course sequencing.

I concetti che compongono il dominio vengono organizzati secondo una struttura eterarchica in cui ogni nodo figlio è legato ad un unico nodo padre attraverso un link il cui significato è quello di "part-of" e "attribute-of". Questo significa che il concetto padre è definito dall'unione di tutti i concetti figli.

Ogni *teaching operation* (termine con cui si indicano le componenti che

saranno organizzate in un corso) viene definita mediante l'insieme delle precondizioni e l'insieme degli outcomes che essa fornisce, in accordo con quello che viene chiamato **plain indexing**. Ogni concetto può quindi rivestire il ruolo di precondizione, in alcuni casi, e di effetto in altri.

In realtà CoCoA utilizza un'estensione del plain indexing in quanto la rappresentazione delle teaching operations comprende anche:

Typed items. Il tipo permette al sistema di distinguere tra i diversi metodi di insegnamento. I metodi considerati sono quattro: presentazione, esempio, compito e domande a risposta multipla.

Regole sui concetti. Si distinguono quattro tipi di regole: light prerequisite, strong prerequisite, light outcome e strong outcome. Questi rappresentano il livello di dettaglio a cui una certa conoscenza è richiesta o fornita: *light* significa livello non molto approfondito, *strong*, invece, indica dettaglio maggiore.

Il problema di questa estensione è che può portare a dilatare i tempi necessari per la definizione del materiale. Tuttavia, essa permette di ottenere soluzioni migliori, più vicine all'utente e di verificare che i corsi rispettino alcuni vincoli.

Un primo tipo di verifica molto importante è quella che riguarda i prerequisiti: **prerequisite checking**. Si ottiene simulando il processo di apprendimento da parte di uno studente: seguendo l'ordine stabilito dall'autore ad ogni passo si controlla che lo studente possieda le competenze richieste in ingresso ed in tal caso si aggiungono le conoscenze fornite in uscita, prima di passare al passo successivo.

Poiché in questo sistema sono previsti diversi metodi di insegnamento, nella verifica delle precondizioni si distingue tra: *Presentation prerequisite*, *Question prerequisite*, *Example prerequisite* ed *Exercise prerequisite*. Essi differiscono per il significato delle precondizioni: ad esempio specificare un concetto come precondizione di una presentazione significa che è necessario per comprenderne i contenuti; una precondizione di una domanda, invece, significa che arrivati a quel punto il concetto è stato insegnato e quindi lo studente dovrebbe essere in grado di rispondere.

Tra i prerequisiti di una certa conoscenza possono comparire solo i concetti ad essa strettamente legati e non relativi, invece, al metodo di insegnamento scelto (detti "legami indiretti"). Supponiamo, ad esempio, che l'insegnamento di α richieda come precondizione β, ma supponiamo che questo sia dovuto solamente al particolare metodo di insegnamento scelto e non al concetto α in sé, allora β non compare tra le precondizioni di α.

Complessivamente il numero di vincoli che si può esprimere è molto elevato. Da qui il grosso vantaggio nell'utilizzo di sistemi automatici per il mantenimento della consistenza e per la verifica.

Per risolvere problemi dovuti al fatto che lo studente non è in possesso di un determinato concetto quando questo viene richiesto come precondizione di un altro, in alcuni casi è sufficiente riorganizzare temporalmente i vari elementi all'interno del corso. In altri casi, invece, tale mancanza può essere risolta solo aggiungendo del materiale che prima non faceva parte del corso.

Altri tipi di verifica sono basati sul fatto che una certa conoscenza non debba essere insegnata più di una volta in modo approfondito[3]. Diverso è il caso degli esempi, in cui è preferibile avere più esempi a fronte di uno stesso concetto.

Inoltre, il sistema è in grado di individuare dove un certo argomento venga trattato in modo approfondito e di aggiungere, in quel punto, una serie di domande. Se il corso viene ricostruito e certi concetti vengono spostati, le domande vengono ridistribuite automaticamente restando sempre legate all'argomento a cui si riferiscono. Questo stretto legame permette di controllare che le domande non siano concentrate in un unico punto ma siano più o meno distribuite fra tutti i concetti trattati. Inoltre, se un certo numero di potenziali domande, con argomento in comune, non viene assegnato potrebbe significare che quel concetto non viene affrontato.

Lo stesso ragionamento può essere fatto per gli esempi e gli esercizi, in particolare si deve verificare che non ve ne siano né troppi né pochi[4] a fronte di uno stesso concetto.

Il course designer potrebbe introdurre un **design document** che rap-

[3]In generale una conoscenza può essere presentata alcune volte in modo superficiale prima di essere insegnata un'unica volta in modo approfondito.

[4]In questo caso si dice che la copertura è bassa.

presenta quali sono i concetti essenziali ed in quale sezione debbano essere introdotti. Quello che fa il sistema è confrontare il corso ed il design document controllando quando viene introdotto un concetto rispetto a quando sarebbe dovuto essere presentato secondo le specifiche dell'autore.

Infine, l'ultimo vincolo presentato in [16] riguarda il problema di come stabilire la difficoltà di un esercizio. Occorre tener presente, infatti, che la difficoltà non può essere espressa da una costante, bensì è relativa al momento in cui viene proposto l'esercizio ed al bagaglio culturale dello studente in quell'istante. Anche il numero di esercizi che lo studente ha già affrontato su quell'argomento influisce sulla difficoltà che avrà nel risolverne un altro. Tutti questi fattori devono essere tenuti in considerazione nel momento in cui si verifica che un esercizio non sia troppo semplice o troppo difficile.

Il sistema è stato realizzato in Java ed è stato completato nel 1999. È stato testato in alcuni casi reali e ha premesso di evidenziare diversi problemi nella definizione dei corsi.

Un problema riscontrato nella prima versione è che il sistema segnalava molti errori che in realtà non erano tali. Può succedere, infatti, che il professore cerchi di stimolare gli studenti con esempi su argomenti ancora prima che questi vengano presentati e spiegati. Tuttavia, il sistema non permetteva queste eccezioni. Nelle versioni successive questo è stato risolto evidenziando in verde questi casi nel report finale ed in rosso quelli più gravi, che non possono essere trascurati e devono essere corretti.

2.3 DCG e CoCoA: osservazioni e conclusioni

I due sistemi appena descritti presentano alcune caratteristiche in comune. La più evidente è che entrambi costruiscono i corsi componendo *Domain Knowledge Elements (DKE)*, che possono essere visti come l'insieme di conoscenze (o learning object) in cui è organizzato il dominio. Il modo più semplice per rappresentarli è senza link tra i vari elementi, tuttavia questi possono essere legati al fine di comporre una *rete semantica* (realizzata tramite l'AND-OR graph in DCG e tramite una struttura eterarchica in CoCoA). Quindi, il dominio viene rappresentato tramite quello che

viene chiamato *domain model* i cui nodi rappresentano un'astrazione dei concetti. Alcuni di essi saranno legati ai *learning item* cioè ai DKE (Domain Knowledge Element).

Il problema della programmazione dinamica adottata in DCG è che si adatta molto bene per personalizzazioni riguardanti singoli studenti, ma non per personalizzazione a livello di classi di utenti. Il motivo è che in quest'ultimo caso tutti gli appartenenti a quella classe devono apprendere le stesse cose nello stesso tempo e devono rispondere nello stesso modo ai test che vengono via via presentati. Una soluzione migliore, in questo caso, è quella di generare i corsi in modalità one-shot sulla base di uno studente che rappresenti l'intera classe.

Un limite che riguarda entrambi i sistemi è che la definizione iniziale della conoscenza è molto costosa. In particolare in DCG per produrre un corso personalizzato è necessario avere a disposizione un elevato numero di materiale da comporre e quindi tra cui scegliere. CoCoA, invece, ha costi iniziali inferiori in quanto questa fase può essere affrontata in modo incrementale aggiungendo, di volta in volta, nuovi elementi al database. Quindi, in contesti molto grandi (large-scale) quest'ultimo approccio è quello consigliato.

Secondo Brusilovsky e Vassileva [16], il course sequencing è una tecnica destinata a prendere sempre più piede nel contesto large-scale web-based educational systems in quanto si sta diffondendo la tendenza di definire i learning object mediante metadati. Quindi aumenta il numero di risorse che possono essere riutilizzate, con conseguente abbattimento dei costi iniziali per la definizione delle risorse.

2.4 Dynamic Assembly per la generazione di corsi

Farrell, Liburd e Thomas in [24] propongono un processo per la generazione automatica di corsi, chiamato *Dynamic Assembly*. Esso si occupa di selezionare un certo numero di learning objects sulla base di alcuni parametri specificati dall'utente, come ad esempio livello di dettaglio, tempo a disposizione, parole chiave, etc.

Si tratta quindi di fornire uno strumento che eviti all'utente sprechi di tempo alla ricerca di informazioni difficili da trovare a causa dell'eccessiva

quantità di materiale tra cui cercare. A tale scopo occorre tener presente due principi, oltre a quello di modularizzazione descritto a pag 15:

Personalizzazione. Assemblare learning object in una sequenza coerente che soddisfi le esigenze dell'utente.

Controllo all'utente. Deve essere l'utente a guidare la composizione secondo le proprie esigenze. Inoltre, questa caratteristica rappresenta anche una responsabilità per lo studente, che deve essere motivato e attento ai propri obiettivi ed alle proprie scelte.

Il sistema sviluppato permette all'utente di indicare le proprie preferenze attraverso una pagina chiamata *Course Assembly Page*, mediante la quale può specificare gli argomenti d'interesse, la durata, livello di difficoltà, etc. Inoltre sono possibili due modalità per la ricerca: *indepth*, quindi limitata alle sole preferenze specificate nella pagina, oppure *overview*, se si desidera includere anche argomenti correlati a quelli indicati.

Sulla base delle informazioni che il sistema ottiene dalla Course Assembly Page è in grado di costruire una nuova pagina in cui vengono elencati i learning object trovati, organizzati in una sequenza di lezioni numerate. Se l'utente non è soddisfatto dell'ordine può cambiarlo a proprio piacimento spostando i vari elementi all'interno della pagina.

I corsi definiti in questo modo vengono memorizzati nel catalogo personale dell'utente, utilizzando come titolo la query inserita nella Course Assembly Page. Una volta generato, un corso può essere modificato in qualunque istante.

La *Search Engine* è la componente del sistema che si occupa della ricerca tra i learning objects, i quali sono classificati in base all'argomento. Trovata la sequenza ad opera della *Dynamic Assembly Engine* questa viene passata, tramite file XML, al *Course Player* che si occupa di eseguirla. Quest'ultimo offre un meccanismo di autenticazione dell'utente, tiene traccia del profilo (che può essere modificato in qualsiasi momento) e visualizza il corso personalizzato.

Le varie componenti comunicano tra loro per mezzo di tre standard: IEEE Learning Object Metadata (IEEE LOM), IMS Content Package, Resource Description Framework (RDF).

Ci sono diversi modi per combinare tra loro risorse in modo da ottenere sequenze che permettano di raggiungere un certo obiettivo. Occorre, però, verificare che queste abbiano un senso dal punto di vista educativo. In [24] gli autori specificano che si tratta di una questione interessante ma ancora aperta. Tuttavia suggeriscono di confrontare i corsi generati automaticamente con corsi che presentino gli stessi argomenti ma che siano costruiti manualmente da un designer.

Il Dynamic Assembly di learning objects descritto in questa Sezione ed il Dynamic Courseware Generation (trattato ad inizio capitolo) non sono da confondere. Il primo, infatti, si occupa della ricerca e della presentazione di learning object in un certo ordine, in base alle esigenze dell'utente. Il secondo, invece, viene utilizzato come strumento per riorganizzare il corso, qualora ce ne fosse bisogno, sulla base delle conoscenze effettivamente acquisite dallo studente.

Capitolo 3

Il *curriculum planning problem* come problema CSP

Fino ad ora sono state presentate alcune soluzioni al problema di come comporre LOs al fine di ottenere corsi che rispettino vincoli e che portino al raggiungimento di un certo obiettivo. Ora verrà discusso un aspetto che può essere visto come una generalizzazione: il *curriculum planning problem*. Consiste nell'estrazione di un insieme di corsi per un determinato periodo, ad esempio un semestre o un trimestre, in modo da rispettare i vincoli accademici (quando un corso è disponibile, prerequisiti, durata dei corsi...) e le preferenze ed esigenze dello studente (quale corso desideri seguire in un particolare semestre, quali aspetti voglia approfondire, quale specializzazione intenda prendere...).

Il problema del curriculum planning non è da confondere con quello che viene definito "problema del calendario" (timetabling). Quest'ultimo, infatti, nasce dall'esigenza di trovare delle sequenze che permettano agli studenti di frequentare i corsi senza sovrapposizioni e garantendo la disponibilità di un'aula e del materiale necessario affinché si possa svolgere la lezione.

La pianificazione di curricula di studi è un lavoro che generalmente viene fatto a mano da un gruppo di professori e che pertanto, oltre ad essere un lavoro noioso, può essere soggetto ad errori. Quindi, è interessante e può portare a notevoli benefici lo studio di soluzioni che permettano di automatizzare queste operazioni.

3.1 MI-CSP: mixed-initiative constraint satisfaction problem

In [49] gli autori propongono una soluzione che vede il problema come *mixed-initiative (MI) constraint satisfaction problem (CSP)*.

Essi individuano due fasi:

1. Viene costruito un piano iniziale utilizzando tecniche di backtracking.

2. Alla soluzione iniziale viene applicato un algoritmo per realizzare l'interazione con l'utente, permettendogli di apportare le modifiche direttamente sul piano mediante un'apposita interfaccia grafica.

Il piano viene presentato graficamente in una tabella (Figura 3.1), dove le colonne rappresentano i semestri e le righe i corsi seguiti in quel periodo. Si noti come già il numero di righe in sé esprima un vincolo: il numero massimo di corsi che possono essere seguiti in un semestre.

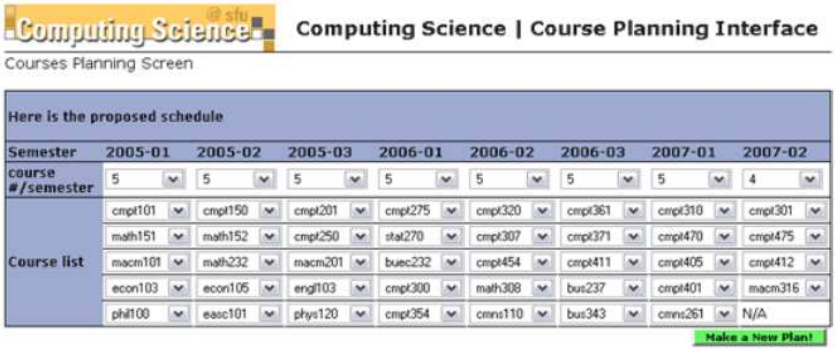


Figura 3.1: Presentazione del piano all'utente

Ogni cella può essere vista come una variabile il cui dominio è un insieme di corsi. Lo studente può cambiarne il contenuto secondo le proprie preferenze. A fronte di una modifica il sistema provvederà a riorganizzare il piano apportando i cambiamenti fatti e propagandoli ai semestri successivi.

Un approccio di tipo "*mixed initiative*" consiste nel fare in modo che sia il sistema, sia l'utente giochino un ruolo attivo nella soluzione di un problema. Non significa, quindi, che il sistema debba avere il controllo sul flusso della

conversazione. Infatti, quando il problema riguarda un task ed un dominio così specifico come quello che stiamo discutendo ora, non è importante per l'utente capire chi abbia "l'iniziativa", questa può essere vista, passo dopo passo, come il modo in cui un certo problema di pianificazione viene risolto.

Le tecniche di programmazione a vincoli sono state ampiamente utilizzate per risolvere problemi di planning e di schedule. Si individuano tre categorie principali di algoritmi:

Metodi Completi o *Systematic algorithms*. Si occupano di esplorare l'intero spazio di ricerca in modo da trovare tutte le possibili soluzioni. Se non viene trovata alcuna soluzione è possibile che non ne esistano oppure l'insieme di vincoli potrebbe essere non consistente.

Metodi Incompleti Fanno uso di *mataeuristiche* al fine di migliorare l'esplorazione limitandola alle sole aree interessanti (interessanti rispetto all'obiettivo che si vuole raggiungere). A differenza degli algoritmi precedenti, le soluzioni trovate non sono tutte, ma sono alcune tra quelle possibili. Gli approcci più comuni riguardano algoritmi evoluzionistici ed algoritmi che fanno uso di strategie di ricerca locali (*Local search algorithms*).

Hybrid search algorithm Si ottengono dalla combinazione dei precedenti. Wu e Havens in [49] propongono l'utilizzo di questi algoritmi per la fase di interazione con l'utente.

Prima di approfondire la descrizione del sistema occorre precisare quali vincoli vengono gestiti.

3.2 Vincoli gestiti dal sistema

I vincoli che vengono presi in considerazione dal sistema sono:

All different. Un corso non può comparire due volte in un piano.

Prerequisite. I prerequisiti di cui si parla in questo approccio sono a livello di corso: un corso può avere come prerequisito un altro corso, oppure un certo numero di crediti.

Mandatory requirement. Alcuni corsi sono obbligatori al fine di ottenere un certo riconoscimento accademico o una qualche certificazione.

Equivalent course. Se un corso è equivalente ad un altro lo studente non può seguirli entrambi con il solo scopo di accumulare crediti.

Breadth. I corsi offerti a livello accademico riguardano aree differenti e livelli differenti. Uno dei vincoli che generalmente viene imposto è che lo studente frequenti corsi di un certo livello in discipline diverse al fine di garantire una conoscenza più ampia, che non riguardi esclusivamente il settore di interesse (crediti interdisciplinari).

Depth. Tra i corsi interdisciplinari scelti per soddisfare i Breath-constraints lo studente deve sceglierne alcuni da approfondire, frequentando corsi di livello più alto. Questo per avere una conoscenza generale su altre discipline che non sia di tipo superficiale, ma che per alcune di esse sia sufficientemente approfondita.

Maximum-load. Il carico di uno studente non deve superare il *maximum load* fissato.

3.3 Il sistema

Il modello è composto da un *system agent* ed uno *user agent* (l'utente). Il primo si occupa di controllare i vincoli del sistema e di mantenere un insieme si soluzioni valide. Inoltre propaga le decisioni prese dall'utente e costruisce, in modo incrementale, la soluzione finale.

Lo user agent, invece, ha il compito scegliere tra piani alternativi e ritrattare le scelte che non hanno portato a soluzioni valide. Anche le richieste fatte dall'utente vengono trasformate in vincoli che vengono aggiunti a quelli di sistema. Tuttavia, questi ultimi hanno priorità più alta e non possono assolutamente essere violati. Infatti, se non vengono trovare soluzioni valide gli unici vincoli che possono essere ritrattati sono quelli che riguardano le decisioni dell'utente.

Come già accennato in precedenza, il sistema usa un approccio a due fasi: dopo aver trovato una prima soluzione la presenta all'utente. Questo

può esprimere le proprie richieste intervenendo direttamente sul piano, cambiando il valore delle celle (si veda la Figura 3.1). Queste modifiche vengono registrate dal sistema come vincoli imposti dallo user agent ed il piano viene nuovamente validato. Quest'ultimo passaggio è necessario in quanto si deve verificare che le nuove modifiche apportate non violino alcun vincolo.

3.4 Formalizzazione del problema

Il problema viene risolto utilizzando la programmazione a vincoli. Un *CSP* (Constraint Satisfaction Problem) è una tripla (V, D, C) dove V è l'insieme delle variabili, D è il dominio di valori associati ad una variabile v (con $v \in V$) e C è l'insieme di vincoli che restringono l'insieme dei valori assegnabili alle variabili.

Ogni cella della tabella (Figura 3.1) viene rappresentata da una variabile. Quindi V è organizzata come una matrice di valori con p colonne (numero di semestri) e r righe (massimo numero di corsi che lo studente può seguire per ogni semestre). D, invece, è l'insieme di tutti i corsi disponibili e ha cardinalità m. Quindi la coppia $(v_{ij}, d_j), \forall v_{ij} \in V$ indica che il valore d_j è stato assegnato alla variabile v_{ij}.

La soluzione consiste in una matrice di coppie *(variabile, valore)*, in modo che tutti i vincoli siano rispettati.

La formalizzazione dei vincoli che il sistema prevede è:

All different. Date due variabili v_{ij} e v_{st}, dove $1 \leq i \leq r$, $1 \leq s \leq r$, $1 \leq j \leq p$ e $1 \leq t \leq p$, se $v_{ij} \neq v_{st}$ allora per le corrispondenti coppie di assegnazione (v_{ij}, d) e (v_{st}, d_2) è sempre vero che $d \neq d_2$.

Prerequisite. Caso 1: dati un insieme $D' : \{d_r, .., d_s\} \in D$ ed un valore $d \in D$ ma $d \notin D'$, dove $1 \leq r \leq m$ e $1 \leq s \leq m$. Si considerino le variabili alla i-esima colonna $V' : \{v_{1i}, .., v_{ri}\}$ dove $1 \leq i \leq p$, una variabile $v \in V_i$ può avere come assegnamento la coppia (v, d) se e solo se è vero che $\forall d_k \in D'$ dove $r \leq k \leq s$ tale per cui $\exists v_{xy} \in V_j$, dove $V_j : \{(v_{11}, .., v_{r1}), ..., (v_{1y}, .., v_{ry})\}$ con $1 \leq y < i \leq p$ allora esiste una coppia di assegnamento (v_{xy}, d_k) nella soluzione.

Caso 2: questo è il caso in cui la precondizione sul corso è espressa dal numero di crediti che lo studente deve possedere. Per un certo valore $d \in D$, relativo ad una soglia ϵ ogni volta che la soglia viene raggiunta dopo l'assegnamento ad una variabile (v_{ij}, d_i), il valore d può essere assegnato ad una variabile $v_{xy} \in V$ dove $y > j$

Mandatory requirement. Un sottoinsieme di valori $D^{'} : \{d_1, .., d_j\} \in D$ con $j < m$, $\forall d_k \in D^{'}$, $\exists v \in V$ tale che esiste un assegnamento nella soluzione dato dalla coppia (v, d_k)

Equivalent course. In un sottoinsieme di valori $D^{'} : \{d_1, .., d_j\} \in D$ se $\exists d, d \in D^{'}$ tale che esiste un assegnamento (v, d) con $v \in V$ allora per ogni altra variabile $x \in V$ con $x \neq v$, l'assegnamento è (x, d_x) con $d_x \notin D^{'}$

Breadth/Depth. Un sottoinsieme $D^{'}$ del dominio di valori è diviso in k gruppi, tale che $D^{'} = \{D^{'}_1, D^{'}_2, .., D^{'}_k\}$. $\forall D^{'}_i \in D^{'}$, $\exists d \in D^{'}_i$ tale che esiste un assegnamento $(v, d), v \in V$ nella soluzione.

Maximum-load. Siano $c_1, .., c_i$ il numero di crediti rispettivamente per $d_1, .., d_i$ e sia Max la costante definita nel sistema, è sempre vero che

$$\sum_{r=1}^{i} c_r \leq Max$$

3.5 Metodi per la ricerca

Per la pianificazione si parte da un piano di studi vuoto, che rappresenta la situazione di una matricola che non ha ancora alcun corso nel proprio piano. Quello che accade normalmente è che lo studente inizi a scegliere i corsi che intende seguire semestre per semestre. Questo è il processo che il sistema cerca di simulare, collocando i corsi nei semestri in modo graduale.

Il metodo usato è basato sul *Dynamic Backtracking* di Ginsberg, al quale sono stati apportati due cambiamenti. Il primo è quello di non riordinare le variabili quando avviene il backtracking ed il secondo è quello di aggiungere il *farward checking*. I vincoli vengono propagati da un semestre (a partire dal primo) ai semestri successivi in ordine temporale. L'utilizzo del forward

checking è stato introdotto per anticipare la verifica dei vincoli, eliminando i valori che non portano a soluzioni valide, cercando, in questo modo, di diminuire il numero complessivo di backtracking.

Per la seconda fase, quella in cui l'utente assume un ruolo attivo ed è chiamato ad esprimere un'opinione sulla soluzione trovata dal sistema, si utilizzano algoritmi di ricerca locale (*local search algorithm*). In particolare, ad ogni passo l'algoritmo cerca una soluzione massimale[1] e verifica che tutti i vincoli siano rispettati. Poiché il sistema è di tipo interattivo, i tempi di risposta sono cruciali, per questo tiene traccia della soluzione migliore trovata fino a quel momento (anche se questa non dovesse rispettare tutti i vincoli imposti dall'utente). Se il tempo limite stabilito per la ricerca viene raggiunto prima che si sia ottenuta una soluzione, allora il sistema restituisce la soluzione migliore trovata fino a quel momento.

3.6 Il problema della simmetria

Questo problema consiste nel fatto che i corsi che compongono un semestre possono essere permutati tra loro. Per il sistema queste rappresentano soluzioni diverse, ma nella realtà non lo sono. Gli autori affrontano questo problema stabilendo un ordinamento parziale dei vincoli. I corsi vengono raggruppati in classi in base all'argomento che trattano. Ad ogni classe viene assegnato un valore che indica l'ordinamento di quella classe rispetto alle altre. A questo punto all'insieme di vincoli ne viene aggiunto uno: il *symmetry-breaking constraint* che impone l'ordinamento rispetto alla classe di appartenenza per ogni corso di una colonna.

Imporre l'ordinamento tra classi e non sugli elementi del dominio evita backtracking sostanzialmente inutili. Si consideri, ad esempio, il caso un cui nella colonna i vi siano tre variabili v_{1i}, v_{2i}, v_{3i}, i possibili valori siano quattro $d_1 < d_2 < d_3 < d_4$ e le classi siano tre $A_1 < A_2 < A_3$, con $d_1 \in A_1$, $d_2 \in A_2$ e $d_3, d_4 \in A_3$. Si supponga che la soluzione proposta dal sistema sia $\{(v_{1i}, d_1), (v_{2i}, d_2), (v_{3i}, d_3)\}$ e che l'utente la cambi assegnando a v_{2i} il valore d_4. In questo caso un ordinamento sui valori del dominio com-

[1] Una variabile a ha un assegnamento massimale d se per la funzione di valutazione f vale che $\neg\exists a \in D$ tale che $f(d) \leq f(a)$

porta backtracking in quanto l'ordinamento relativo non è più rispettato ($d_1 \leq d_4 > d_3$), mentre un ordinamento rispetto alle classi no, in quanto l'ordinamento relativo non cambia $A_1 < A_2 < A_3$.

Questo esempio aiuta a comprendere perché i symmetry-breaking constraints vengano valutati considerando l'ordine relativo tra le classi e non tra i valori del dominio.

I vincoli introdotti in questa sezione possono facilmente essere estesi in modo da costruire curricula che rispettino anche altre caratteristiche. Un esempio è quello dei $BACP$[2] di cui si da una breve descrizione nella Sezione successiva. Si è deciso di trattare la descrizione di tale problema in modo superficiale per non scendere nel dettaglio di quella che è la risoluzione di uno tra i possibili vincoli a cui può essere sottoposto un piano di studi. Tuttavia, in bibliografia si possono trovare i riferimenti per ulteriori approfondimenti.

3.7 Curricula accademici bilanciati

Il problema di disegnare curricula di studio "bilanciati" consiste nell'esigenza di distribuire i vari corsi, che costituiscono il piano dello studente, lungo la durata prevista per conseguire la certificazione. Anche in questo caso il problema viene affrontato definendo opportuni vincoli che riguardano il numero ed il tipo di corsi seguito in ogni periodo accademico. Ad esempio vengono stabiliti un limite massimo ed un limite minimo sul numero di crediti che lo studente può conseguire in un certo periodo[3]. Questi vengono definiti *vincoli di carico* (load). Oltre a questi ci sono ovviamente altri vincoli che riguardano le precedenze tra corsi (prerequisiti).

In [34] viene proposta una soluzione al problema detta "ibrida" (si veda la classificazione degli algoritmi data a pag. 34, Capitolo 3.1), in quanto propone di sfruttare i benefici che possono derivare integrando algoritmi completi ed incompleti al fine di risolvere istanze complesse in modo efficiente.

[2] Balanced Academic Curriculum Problem

[3] Il numero di CFU (crediti formativi universitari) è calcolato in proporzione al numero di ore che si stima lo studente debba spendere per imparare gli argomenti di quel corso e superare l'esame in modo proficuo.

In particolare, si sfruttano algoritmi di risoluzione basati sulla propagazione dei vincoli e algoritmi genetici [38].

3.7.1 Parametri da considerare

Di seguito vengono riportate le definizioni di alcuni parametri, per la verifica di curricula, che vengono presi in considerazione nell'approccio proposto dagli autori in [19, 34].

Academic Curriculum Un curriculum accademico è definito come un insieme di corsi ed un insieme di precedenze tra questi.

Number of periods Indica il numero di periodi tra i quali si vogliono organizzare i vari corsi. Possono essere rappresentati in trimestri, semestri, etc.

Academic load Ad ogni corso viene associato un certo numero di crediti che rappresenta le ore di studio e di lezione richiesti per poter superare l'esame in modo vantaggioso.

Prerequisites Vengono definiti in termini di corsi. Quindi un corso può avere come prerequisito un altro corso.

Minimum academic load Per ogni periodo è richiesto un numero minimo di crediti per poter considerare lo studente come iscritto a tempo pieno.

Maximum academic load Per ogni periodo viene anche fissato un limite massimo sul numero di crediti.

Minimum number of curses Per considerare uno studente come iscritto a tempo pieno non è sufficiente specificare un numero minimo di crediti, ma viene indicato anche un numero minimo di corsi per ogni periodo.

Maximum number of curses Viene specificato un numero massimo di corsi per ogni periodo.

3.7.2 Modello matematico del problema

Sulla base di questi parametri e dei vincoli di bilanciamento del piano, il modello matematico risulta essere il seguente:

Parametri

- m: numero di corsi
- n: numero di periodi accademici
- α_i: numero di crediti per il corso i, $\forall i = 1, .., m$
- β: Minimo numero di crediti ammesso per ogni periodo
- γ: Massimo numero di crediti ammesso per ogni periodo
- δ: Minimo numero di corsi per ogni periodo
- ϵ: Massimo numero di corsi per ogni periodo

Variabili

Le variabili definite sono:

- Un insieme di variabili rappresenta i corsi per ogni periodo:

$$x_{ij} = \begin{cases} 1 & \text{se il corso } i \text{ è assegnato al periodo } j;\ \forall i = 1, .., m\ \forall j = 1, .., n \\ 0 & \text{altrimenti} \end{cases}$$

- c_j: numero di crediti per il periodo j; $\forall j = 1, .., n$
- c: numero di crediti massimo per ogni periodo

Funzione obiettivo

Si tratta di un problema di minimo, dove quello che si cerca di minimizzare è il carico massimo dello studente associato ad un periodo.

- $Min\ c\ =\ Max\{c_1, .., c_n\}$

Vincoli

- Il numero di crediti per il periodo j è definito nel seguente modo:

$$c_j = \sum_{i=1}^{m} \alpha_i \times x_{ij} \ \forall j = 1, .., n$$

- Ogni corso i deve essere assegnato ad un periodo j

$$\sum_{j=1}^{n} x_{ij} = 1 \ \forall i = 1, .., m$$

- Il fatto che un corso b abbia come precondizione un corso a si esprime nel seguente modo:

$$x_{bj} \leq \sum_{r=1}^{j-1} x_{ar} = 1 \ \forall j = 2, .., n$$

- Il massimo numero di crediti è definito come:

$$c = Max\{c_1, .., c_n\}$$

  Che può essere rappresentato con il seguente insieme di vincoli lineari:

$$c_j \leq c \ \forall j = 1, .., n$$

- Il numero di crediti acquisiti nel periodo j deve essere maggiore o uguale al minimo richiesto:

$$c_j \geq \beta \ \forall j = 1, .., n$$

- Allo stesso tempo il numero di crediti acquisiti nel periodo j deve essere minore o uguale al massimo ammesso:

$$c_j \leq \gamma \ \forall j = 1, .., n$$

- Il numero di corsi in un certo periodo j deve essere maggiore o uguale al limite inferiore imposto:

$$\sum_{i=1}^{m} x_{ij} \geq \delta \ \forall j = 1, .., n$$

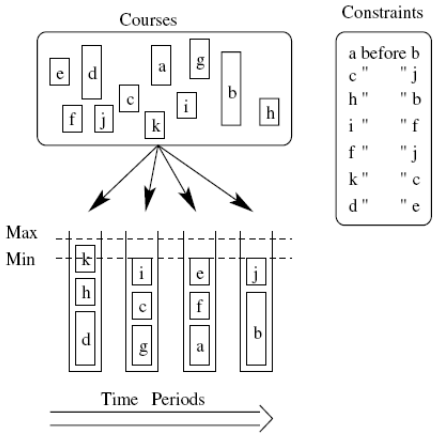


Figura 3.2: Esempio di assegnazione dei corsi ai vari periodi

- Il numero di corsi in un certo periodo j deve essere minore o uguale al limite massimo stabilito:

$$\sum_{i=1}^{m} x_{ij} \leq \epsilon \ \forall j = 1, .., n$$

Quindi una soluzione consiste nell'assegnare i corsi ai vari periodi, come rappresentato in Figura 3.2.

Per ulteriori dettagli su tale approccio si veda [19, 33, 34]. In essi ed in [18, 32] gli autori forniscono anche confronti e risultati sperimentali.

Parte II

DCML: Declarative Curricula Model Language

Capitolo 4

Il sistema Wlog per la generazione di curricula

Come precisato in [16], è importante che un curriculum rispetti tre punti fondamentali:

1. Deve guidare l'apprendimento dell'utente fino al raggiungimento di tutte le conoscenze specificate nel suo learning goal, cioè deve permettere di raggiungere lo scopo per cui è stato disegnato.

2. Nel realizzare il punto precedente bisogna fare in modo che ogni volta che viene introdotto un nuovo concetto, lo studente sia in possesso delle conoscenze necessarie per apprenderlo appieno. Non vi devono essere, cioè, *competency gap*.

3. Infine, bisogna essere sicuri che il curriculum rispetti i *design goals*. Si tratta di obiettivi che definiscono lo scopo educativo del piano di studi.

Si pensi, infatti, ad un curriculum che non permetta all'utente di raggiungere le conoscenze che desidera acquisire o non sia conforme alle regole stabilite dall'ente che eroga i corsi e pertanto non consenta di raggiungere alcuna certificazione. Altrettanto controproducente sarebbe presentare un argomento all'interno di un corso (o addirittura un intero corso) che lo studente non è in grado di capire in quanto richiede conoscenze che non ha. In tutti questi casi sarebbe un piano di studi di scarsa utilità.

Nonostante l'importanza, tali verifiche vengono fatte manualmente da un gruppo di professori (così come avviene in alcuni casi per la composizione di learning object, Capitolo 2), ma non si tratta un compito facile in quanto sono richieste informazioni su ogni singolo studente (carriera precedente, conoscenze in possesso, obiettivi, etc.) e su ogni corso in modo da poter offrire delle alternative all'utente. Inoltre, è un'attività che richiede un notevole dispendio di tempo. Un aiuto viene dal settore della ricerca in cui si stanno studiando tecniche che permettano di automatizzare questo processo. Nei capitoli precedenti sono state descritte alcune soluzioni per la verifica di vincoli quali numero minimo e numero massimo di crediti e di corsi, corsi che devono comparire una sola volta e corsi obbligatori, caratteristiche dei curricula bilanciati, problemi di timetable, etc.

Possiamo così distinguere tra *vincoli di fruizione* e *modello delle competenze*. Nel primo gruppo rientrano necessità legate ad orario, numero di crediti, bilanciamento del carico e così via. Cioè tutti quegli aspetti che riguardano la struttura generale che i curricula devono avere.

Il modello delle competenze, invece, raccoglie i vincoli che rappresentano i design goals e altri tipi di relazioni tra le conoscenze che non sono catturabili con le sole condizioni su precondizioni-effetti con cui si descrivono i corsi.

Particolarmente interessante è la proposta descritta in [9] che prevede che ogni corso sia visto come un'azione che un agente può compiere qualora le precondizioni siano rispettate. Gli effetti vengono aggiunti a quello che rappresenta lo stato mentale dello studente.

Di seguito si cercherà di descrivere meglio questa visione del problema, approfondendo la soluzione proposta dagli autori e cercando di evidenziarne vantaggi e limiti. Quest'ultimi riguardano soprattutto la fase di verifica di cui questa tesi proporrà, nei capitoli seguenti, un approccio diverso con l'obiettivo di superare i problemi riscontrati.

4.1 Il paradigma ad azioni per la rappresentazione dei corsi

Generalmente, quando si definisce un corso (che verrà poi utilizzato per formare una sequenza) oltre ad elencare gli argomenti trattati, viene specificato l'insieme delle precondizioni richieste affinché possa essere compreso e seguito con profitto. Si tratta di conoscenze che possono essere espresse in termini di altri corsi. Ad esempio, si può dire che i corsi "Programmazione I" ed "Introduzione ai Linguaggi di Programmazione" sono precondizioni del corso "Programmazione ad Oggetti". L'approccio proposto in [9] prevede, invece, di lavorare in termini di *competenze*, che vengono utilizzate per descrivere il learning goal, le conoscenze iniziali dello studente ed i corsi. Esse e le varie relazioni che le legano vengono mantenute all'interno di opportune ontologie che rappresentano il *knowledge model*.

Per riprendere l'esempio di prima, è come se si descrivesse il corso di "Programmazione ad Oggetti" dicendo che le sue precondizioni sono, ad esempio, "Programmazione Iterativa", "Utilizzo della Ricorsione", "Principali Strutture Dati", e gli effetti sono "Programmazione ad oggetti" (il cui nome richiama quello del corso ma in questo caso è un concetto), "Strutture dati complesse", "Ereditarietà" e così via.

Quindi, ogni risorsa può essere semanticamente annotata indicando anche il livello a cui una certa competenza è richiesta o fornita. Ad esempio:

resource_name: db_per_biotecnologie,
preconditions: (db_relazionali, beginner)
effects: (db_scientifici, advanced)

dove "db_per_biotecnologie" è il nome della risorsa, che richiede "db_relazionali" a livello *beginner* come precondizione e fornisce "db_scientifici" a livello avanzato come effetto.

In quest'ottica, ogni corso può essere visto come un'azione applicabile se sono rispettate le precondizioni e la generazione di un piano di studi è il compito di un agente razionale [43] che deve trovare una sequenza di azioni che permettano all'utente di raggiungere il suo learning goal. Questo problema prende il nome di *curriculum sequencing*.

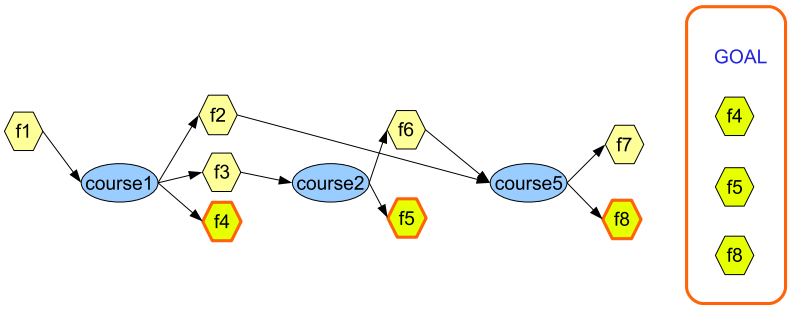


Figura 4.1: Rappresentazione di un curriculum di studi: esso deve rispettare i vincoli imposti dalle precondizioni e dagli effetti forniti dai vari corsi e deve portare lo studente al raggiungimento del suo Learning Goal

Ogni azione viene descritta in termini di precondizioni ed effetti che corrispondono rispettivamente alle conoscenze richieste in ingresso ed a quelle fornite dal corso. La loro esecuzione avviene in uno stato interno dell'agente che viene utilizzato per simulare il processo di apprendimento dello studente.

Oltre a permettere la pianificazione di nuovi curricula, il sistema realizzato è in grado di verificare che un piano di studi proposto da un utente sia valido, controlla cioè che la sequenza di corsi sia corretta (non vi siano precondizioni non soddisfatte) e che permetta di raggiungere il learning goal per cui è stata composta. In caso di fallimento della verifica l'agente ne fornirà una motivazione.

La Figura 4.1 rappresenta un curriculum di studi che rispetta i vincoli di tipo precondizioni ed effetti e permette il raggiungimento del Learning Goal[1]. Sostanzialmente si tratta di una sequenza di corsi, ognuno dei quali richiede un insieme di conoscenze in ingresso e fornisce alcune competenze in uscita. Si noti, quindi, che ogni concetto può essere, a seconda di come è utilizzato, una precondizione o un effetto.

Il linguaggio di programmazione utilizzato dagli autori è *Dynamic in LOGic* [10, 41]. Esso permette di ragionare sugli effetti di un'azione in un ambiente che cambia dinamicamente. Il sistema realizzato, invece, prende il nome di Wlog [9]. L'interazione con esso avviene mediante pagine web

[1]La rappresentazione dei vincoli imposti dal designer verrà trattata nel capitolo successivo.

costruite dinamicamente, realizzando così una sorta di conversazione con l'utente.

4.2 Il linguaggio di programmazione Dynamic in LOGic

Dynamic in LOGic è un linguaggio di programmazione per agenti che consente di ragionare su azioni ed effetti. È ispirato alla logica modale.

Gli agenti sono entità che interagiscono con l'ambiente compiendo delle azioni descritte tramite precondizioni ed effetti. Nel caso di curricula di studi, ogni curriculum rappresenta una sequenza di corsi che, sulla base di quanto detto precedentemente, corrisponde ad una sequenza di azioni. L'agente possiede uno stato che mantiene le conoscenze in quel momento in possesso dello studente. L'esecuzione di un'azione provoca un cambiamento che consiste nell'aggiungere le competenze acquisite[2]. Si assume, infatti, che il dominio sia monotono cioè che la conoscenza possa solo aumentare. Rappresenta la situazione in cui lo studente "impara" senza mai dimenticare.

In questo linguaggio si possono esprimere due modalità: $[a]\alpha$, cioè α vale dopo ogni esecuzione dell'azione a, e $\langle a \rangle \alpha$, il cui significato è che esiste una esecuzione dell'azione a, dopo la quale α vale.

Le azioni complesse vengono definite in termini di procedure a partire da altre azioni.

$$p_0 \textbf{ is } p_1, ..., p_n (n \geq 0)$$

dove p_0 rappresenta il nome della procedura. Vale il seguente assioma:

$$\langle p_1 \rangle \langle p_2 \rangle ... \langle p_n \rangle \varphi \supset \langle p_0 \rangle \varphi$$

che significa che se in uno stato esiste una possibile esecuzione di p_1 seguita da una possibile esecuzione di p_2 e così via fino a p_n, allora in quello stato esiste una possibile esecuzione di p_0.

Inoltre, il sistema è in grado di realizzare una sorta di interazione con l'utente nel caso in cui vi sia un'alternativa tra più strade possibili, ognuna che porta al raggiungimento del learning goal. In questi casi, anziché prendere

[2]Competenze e conoscenze vengono usati come sinonimi.

una decisione, l'agente propone le diverse soluzioni direttamente all'utente che è quindi chiamato ad operare una scelta.

Con questo linguaggio è possibile realizzare tre tipi di ragionamento:

temporal projection. Metodo utilizzato per ragionare sugli effetti a partire dalle cause. Lo scopo è quello di predire gli effetti di azioni che non sono ancora state eseguite, avendo a disposizione le sole conoscenze, eventualmente parziali, sullo stato iniziale.

temporal explanation. L'agente considera alcuni fatti come effetti di azioni che sono state eseguite e ragiona su quali possano essere state le cause.

planning. Probabilmente è il più conosciuto tra i tre. L'obiettivo è quello di trovare una sequenza di azioni che eseguite a partire da un certa situazione, conduca ad un nuovo stato in cui certe condizioni valgono.

La costruzione di curricula viene interpretata come un problema di pianificazione di procedure, mentre la validazione di un dato piano rientra nel primo tipo di ragionamento descritto, il temporal projection.

4.3 Costruzione di curricula personalizzati mediante procedural planning

In generale, il problema della pianificazione consiste nel chiedersi se esista una sequenza di azioni che applicate ad uno stato iniziale s porti ad uno stato in cui valgano determinate condizioni.

Nel linguaggio Dynamic in LOGic lo stesso processo avviene mediante la seguente query: $\langle p \rangle \ Fs$, che si legge "Data la procedura p, esiste un'esecuzione di p che termina e che porta dallo stato iniziale in uno stato in cui Fs vale?".

Questo viene definito *procedural planning* proprio perché la procedura p restringe lo spazio in cui viene cercata la soluzione. Nell'ambito dei curricula di studi che stiamo considerando, la procedura rappresenta il modo in cui lo studente può acquisire la figura professionale richiesta, mentre Fs rappresenta il suo learning goal. In Figura 4.2 vengono rappresentati i piani trovati dalla procedura. In particolare, vi sono dei punti di scelta (in questi

casi sarà l'utente a decidere quale percorso intraprendere) ed uno dei piani non è valido poiché supera il limite di crediti posto come vincolo iniziale.

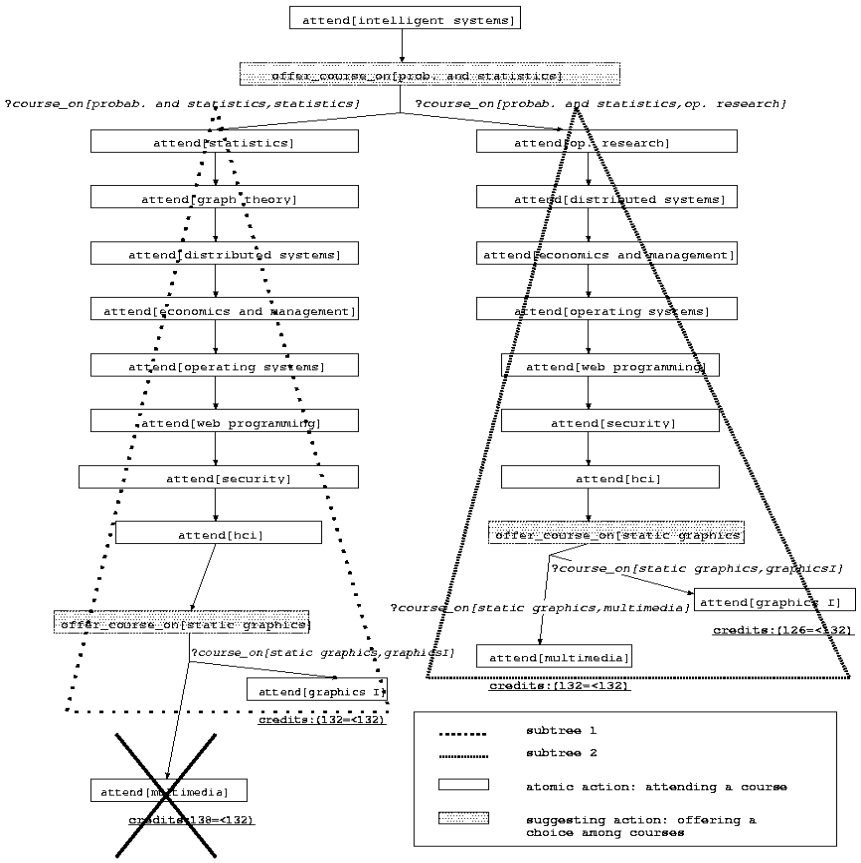


Figura 4.2: Piani ottenuti mediante il procedural planning

La formulazione di una query restituisce una traccia di esecuzione composta da azioni primitive. Questa può essere lineare oppure un piano condizionale qualora vi fossero delle alternative. Tuttavia, è da sottolineare che vengono restituite le sole esecuzioni valide (non vi sono precondizioni mancanti ed ogni strada porta al raggiungimento dell'obiettivo).

4.4 Validazione di curricula per mezzo del temporal projection reasoning

La validazione si presenta nel caso in cui l'utente proponga dei piani di studio da lui costruiti. Come già detto, infatti, la pianificazione operata dal sistema restituisce solo sequenze valide. Ma nel caso in cui l'utente decida di disegnare un curriculum secondo le proprie preferenze ed esigenze, si può dire che questo soddisfa tutte le dipendenze del dominio e che lo porterà ad acquisire le conoscenze richieste?

A questa domanda è possibile rispondere con il temporal projection, data, cioè, una sequenza di azioni $a_1, ..., a_n$ ci si chiede se al termine la condizione Fs sia soddisfatta. Nel contesto del curriculum sequencing i piani corrispondono a sequenze di corsi $c_1, ..., c_n$, mentre l'obiettivo Fs è l'insieme delle competenze che l'utente desidera avere al termine dell'esecuzione. Occorre verificare, quindi, se, dato il background di competenze dello studente (situazione che rappresenta lo stato iniziale), la successione $c_1, ..., c_n$ porti al raggiungimento dell'insieme di conoscenze Fs.

Nel caso la verifica fallisca, il sistema deve fornire una motivazione, precisando se il problema riguarda la mancanza di una conoscenza richiesta in ingresso ad un corso, oppure se sia dovuto al mancato raggiungimento del learning goal.

Sfruttando la particolarità del dominio di essere monotono, la tecnica utilizzata in [9] basata sul temporal explanation consiste nel restituire l'insieme di conoscenze che lo studente dovrebbe avere e la cui mancanza causa il fallimento della verifica. L'idea è che un piano è sempre applicabile a patto che se alcune precondizioni non vengono soddisfatte dai corsi precedenti nella sequenza, tali conoscenze facciano parte del bagaglio iniziale dell'utente.

4.5 Il Procedural planning e i problemi legati alla sua natura prescrittiva

Il paradigma ad azioni per la definizione dei corsi e l'aver scelto di descrivere le relazioni tra essi in termini di concetti, consente di apportare

facilmente modifiche ed aggiunte, e permette di definire vincoli non esprimibili con un granularità a corso anziché a conoscenza. Ad esempio, è possibile precisare che una certa relazione tra due competenze sussiste qualsiasi siano i corsi che le offrono.

Questo sistema è molto efficiente dal punto di vista della pianificazione. Il procedural planning, infatti, permette di restringere notevolmente lo spazio in cui ricercare un piano valido. È come se le procedure permettessero di definire diverse strategie di composizione e quindi diverse politiche di insegnamento.

Tra tutti i piani possibili, oltre a quelli corretti, troviamo le sequenze che violano dei vincoli di tipo *precondizioni-effetti* e quelle che non permettono il raggiungimento del learning goal. A queste si devono aggiungere tutte quelle che non rispettano le regole imposte dall'ente che offre i corsi. Si supponga, ad esempio, che l'Università degli Studi di Torino offra dei piani che permettono di raggiungere la qualifica di *esperto nella progettazione di applicazioni web*. È quindi interesse dell'Università che qualsiasi curriculum, oltre a rispettare le preferenze dell'utente ed a metterlo in condizioni di comprendere il contenuto di ogni corso, permetta di raggiungere competenze legate alla progettazione di applicazioni web. Ad esempio, imponendo conoscenze nell'ambito della sicurezza informatica, dei database e così via. Inoltre, stabilita una durata complessiva, i corsi vengono distribuiti lungo tale periodo, imponendo che corsi del primo anno debbano essere seguiti prima di corsi del secondo anno. Il procedural planning permette di considerare solo piani che rispettano questi vincoli garantendo il riconoscimento della figura professionale che il curriculum dichiara di fornire. Su questi verranno poi controllate le precondizioni dei corsi ed il raggiungimento del learning goal.

Usando il formalismo introdotto dagli autori possiamo scrivere che:

$$M \vdash \langle p \rangle \, G$$

dove M rappresenta il modello, p la procedura e G il goal. Supponendo di trovare un piano σ che soddisfi sia i vincoli imposti dal modello, sia il learning goal si può scrivere:

$$M \vdash \langle \sigma \rangle \, G$$

Poiché σ è una sequenza di corsi (quindi la sua esecuzione termina ed è deterministica [11]), vale anche che:

$$M \vdash [\sigma]G$$

Quindi, dato un insieme di vincoli e una procedura, il procedural planning permette di trovare in modo efficiente un piano corretto. Il ruolo della procedura è quello di individuare un insieme di sequenze lecite, che rispettino, cioè, le restrizioni imposte dalle istituzioni che forniscono i corsi.

I problemi di tale approccio si hanno in fase di verifica, che consiste nel chiedersi se un piano σ dato (ad esempio proposto dall'utente) sia valido o meno: $M \vdash [\sigma]G$. Sicuramente si può controllare il raggiungimento del learning goal e si possono verificare le precondizioni dei corsi. Tuttavia, il ragionamento temporale non permette di verificare se σ faccia parte o meno dei piani individuati dalla procedura p. Quello che manca, quindi, è la verifica di conformità del piano rispetto al modello.

Inoltre, i vincoli imposti dalla procedura sono troppo restrittivi. Essa cattura la struttura del curriculum ma la sua natura *prescrittiva* impone la precisazione di tutto ciò che è lecito. Ciò che non viene specificato è ritenuto non valido. Questo aspetto, però, non si adatta bene ad ambienti aperti e dinamici come il WWW.

Si immagini, ad esempio, di avere le due competenze *(database, beginner)*[3] e *(database, advanced)*. È intuitivo pensare che questo ordine tra le due debba essere rispettato. Se si aggiunge la conoscenza *(english, advanced)* quale posizione deve occupare? Verrebbe da pensare che questa sia indipendente dalle altre e possa, quindi, essere acquisita prima, dopo oppure tra le due precedenti. In realtà, per ottenere questo risultato occorre specificare esplicitamente ognuna delle tre collocazioni possibili per la conoscenza *(english, advanced)*.

Per superare questo limite, si può ancora una volta ricorrere al paradigma ad azioni. Esso consente di vedere la fase di pianificazione come il compito di un agente razionale [43] che deve trovare una sequenza di azioni che

[3]Le competenze vengono rappresentate mediante una coppia nome competenza, livello a cui deve essere acquisita

permetta di raggiungere uno stato goal, a partire da una certa configurazione dell'istante iniziale.

Nei sistemi ad agenti, l'interazione viene regolata per mezzo di protocolli che definiscono le sequenze di azioni lecite per realizzare una certa comunicazione o per l'adempimento di un certo compito. Tuttavia, questa soluzione limita la flessibilità dell'agente. Per questo motivo, Youlm e Singh in [51] propongono il *social approach*, una soluzione in cui la specifica di un protocollo mira a catturare il significato intrinseco delle azioni. Questo viene modellato attraverso i *social commitments*. Sostanzialmente, si tratta di focalizzarsi sugli obblighi di una parte rispetto ad un'altra. Ogni agente, infatti, in quanto facente parte di una società ha un *ruolo* che comporta obblighi ed impegni verso gli altri componenti.

Ogni azione del protocollo viene vista come un'operazione sui "*commitments*" di un agente. Esso, infatti, può rivedere i propri impegni cancellandone ed aggiungendone alcuni, o adempiendo ad altri.

Definire protocolli in questo modo incrementa la flessibilità con cui un agente può operare in quanto gli consente di ragionare sul proprio comportamento e su quello degli altri al fine di scegliere l'azione migliore da compiere.

A titolo di esempio si consideri il protocollo NetBill per l'acquisto e la vendita di beni su internet [50]. La Figura 4.3 riporta un possibile scambio di messaggi tra un commerciante ed un acquirente. L'interazione inizia con la richiesta di un preventivo da parte del cliente. Ottenuta la risposta, viene inviata la conferma dell'ordine al commerciante, il quale invia la merce, attende il pagamento e termina la comunicazione inviando la ricevuta.

La rigidità di tale protocollo è dovuta al fatto che esso contempla tutte e sole le azioni che i due agenti possono compiere e specifica anche l'ordine con cui devono essere scambiati i messaggi. Non rientrano, ad esempio, i seguenti scenari:

- L'offerente vuole pubblicizzare i propri prodotti, pertanto invia i preventivi senza attendere una richiesta esplicita.

- L'acquirente è deciso a comprare i beni qualsiasi sia il loro valore. Non è interessato, quindi, a ricevere alcun preventivo.

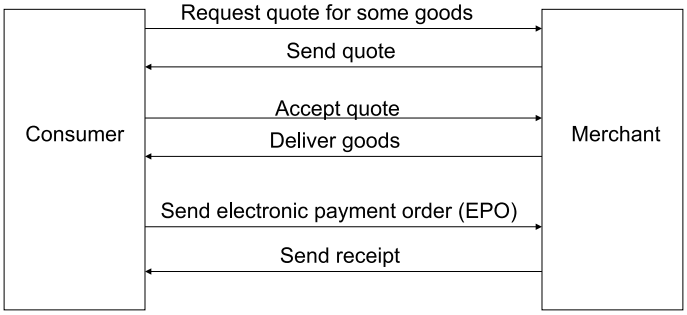


Figura 4.3: Esempio di protocollo di comunicazione tra agenti per l'acquisto e la vendita di beni su internet

- Il commerciante invia la merce senza aspettare alcuna conferma da parte del cliente.

Un protocollo orientato ai "commitments" degli agenti consiste nell'esprimere, ad esempio, che il commerciante ha preso l'impegno di inviare la merce al cliente, oppure che invierà la merce solo nel momento in cui riceverà il pagamento. Come si può notare, con questa scelta non è necessario specificare alcuna azione o stabilire alcun ordine per regolare l'interazione tra gli agenti.

Da tale approccio si può prendere spunto per il dominio dei curricula di studio. In particolare, la caratteristica che si vuole adottare consiste nello specificare solo i vincoli strettamente necessari, anziché elencare tutte le possibili sequenze di concetti, lecite.

In quest'ottica, i vincoli servono per stabilire un ordine relativo di acquisizione tra alcune conoscenze, imponendo, ad esempio, che un certo concetto debba essere affrontato prima di un altro, senza specificare quanto tempo debba intercorrere tra i due o quali altri argomenti possano essere affrontati tra uno e l'altro.

Riprendendo l'esempio precedente, con questo nuovo approccio sarà sufficiente esprimere che *(database, beginner)* deve essere affrontato prima di *(database, advanced)*. Come questo, altri vincoli potranno restringere ulteriormente le sequenze di corsi lecite, ma il fatto che non si esprimano quali competenze debbano essere apprese tra i due concetti comporta una gamma

di piani possibili tra cui l'utente potrà scegliere.

Inoltre, se l'ente che eroga i corsi decidesse di aggiungerne alcuni, l'insieme dei vincoli non dovrebbe essere modificato. Nell'approccio prescrittivo, invece, si sarebbero dovute aggiungere tutte le nuove sequenze valide.

Il formalismo scelto per la definizione dei vincoli si basa sulla logica modale LTL. Verrà presentato nel capitolo successivo insieme a DCML, un linguaggio grafico per facilitarne la definizione [8, 3].

Capitolo 5

Definizione del curricula model

Nei capitoli precedenti sono stati presentati alcuni temi importanti in ambito e-learning come quello del *riuso* e della *personalizzazione* di Learning Objects localizzati in punti diversi della fitta rete che costituisce il Semantic Web. Un aspetto interessante, è sicuramente quello della composizione automatica di tali risorse al fine di poter offrire all'utente una sequenza, che gli permetta di ottenere le conoscenze richieste senza costringerlo a lunghe ed estenuanti ricerche senza la garanzia di avvicinarlo al suo obiettivo.

Il problema della verifica automatica di piani di studio è legato a quello della composizione automatica, ma non si limita ad essa, deve infatti essere applicabile ad una qualsiasi sequenza, ottenuta in un qualsiasi modo. Un buon punto di partenza per affrontare la questione è dare una definizione chiara di quali siano i vincoli che si intende verificare. Come già detto si possono raggruppare in due categorie: vincoli imposti dal modello delle conoscenze e vincoli di fruizione.

In questo lavoro l'attenzione cade principalmente sul primo gruppo. L'intenzione è fornire uno strumento intuitivo per la definizione di quelle che vengono anche chiamate *linee guida* imposte dalle istituzioni. Si tratta di regole, generalmente dipendenze temporali tra conoscenze, che un buon curriculum deve rispettare e che non sono esprimibili né nella definizione dei corsi (conoscenze attese in ingresso ed in uscita), né imponendo vincoli di precedenza

tra questi. Tutto ciò senza dimenticare i vincoli di tipo precondizioni-effetti ed il raggiungimento del learning goal.

Il formalismo scelto è basato sulla logica. Essa, infatti, rappresenta uno strumento indispensabile per la definizione delle proprietà che un certo sistema deve soddisfare, in quanto è, per definizione, "non ambigua". In particolare, verrà adottata la Logica Lineare Temporale LTL (Linear Temporal Logic) di cui viene riportata una breve descrizione nella sezione successiva[1].

5.1 La logica LTL in breve

LTL è l'acronimo di *Linear-time temporal logic* [29]. Si tratta di una logica temporale che modella il tempo come una sequenza lineare di stati (chiamata cammino) che si estendono all'infinito nel futuro. Questa sequenza può essere interpretata in diversi modi: come il susseguirsi di situazioni che rappresentano l'evoluzione di un mondo nel tempo, oppure gli stati attraversati da un programma durante la sua esecuzione.

5.1.1 Sintassi della logica LTL

La sintassi per la logica LTL in Backus Naur Form è la seguente:

$$\phi \ ::= \ \bot \mid \top \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \rightarrow \phi)$$
$$\mid X\phi \mid F\phi \mid G\phi \mid \phi U \phi \mid \phi W \phi \mid \phi R \phi$$

dove p è un qualsiasi atomo proposizionale.

I connettivi X (indicato anche con $\bigcirc$), F (indicato anche con $\diamond$), G (indicato anche con $\Box$), U, R e W sono detti *connettivi temporali* il cui significato è:

- X: lo stato successivo (ne**X**t state)
- F: un qualche stato futuro (in the **F**uture)
- G: in tutti gli stati futuri (**G**lobally)

[1] Il lettore che avesse familiarità con la logica LTL può trascurare le Sezione 5.1 senza compromettere la comprensione degli argomenti che seguono.

- U: **U**ntil[2]
- R: **R**elease
- W: **W**eak until

I connettivi unari, cioè $\neg$, X, F e G hanno priorità più alta rispetto agli altri. Seguono U, R e W, poi $\wedge$ e $\vee$ ed infine $\rightarrow$.

5.1.2 Semantica della logica LTL

I sistemi su cui vengono verificate le formule LTL possono essere modellati come *transition systems*. Si tratta di grafi orientati i cui nodi rappresentano stati (che mantengono alcune informazioni sul sistema valide in un determinato istante) e gli archi rappresentano transizioni (cioè modellano come un sistema può evolvere da uno stato ad un altro).

Definizione 5.1. Un *modello* $M = (S, \rightarrow, L)$ è composto da un insieme di stati S legati da una relazione di transizione $\rightarrow$ (una relazione binaria su S) tale che per ogni $s \in S$ esiste almeno uno stato $s' \in S$ con $s \rightarrow s'$ (cioè per ogni stato esiste almeno un successore). L è la funzione di etichettatura $L : S \rightarrow P(Atomi)$.

In altre parole un modello consiste in un insieme di stati e una relazione $\rightarrow$ che indica come evolve il sistema. Inoltre, ad ogni stato è associato un insieme di atomi $L(s)$ veri.

Come specificato nella Definizione 5.1, un modello non deve avere stati di *deadlock*, cioè ogni stato deve avere almeno un successore. Per i sistemi di transizione che non rispettano questa condizione è sufficiente aggiungere uno stato s_d, chiamato stato di deadlock, con un arco di self-loop. Ogni stato che nel modello precedente non aveva archi uscenti, ora avrà s_d come stato successore.

Definizione 5.2. Un *cammino* in un modello $M = (S, \rightarrow, L)$ è una sequenza infinita di stati $s_1, s_2, s_3, \ldots$ in S tale che per ogni $i \geq 1, s_i \rightarrow s_{i+1}$.

[2]In alcune notazioni, questo viene chiamato Strong Until e viene indicato con il simbolo U per differenziarsi dal Weak Until, che viene rappresentato con lo stesso simbolo scritto in modo diverso: U . Questa è la notazione utilizzata in [28].

Dato un cammino $\pi = s_1 \rightarrow s_2 \rightarrow \ldots$, si indica con π^i il suffisso che inizia allo stato s_i.

Un modello può, quindi, essere rappresentato mediante un grafo orientato o un sistema di transizione. Per rappresentare i cammini, invece, può essere più conveniente utilizzare un *albero di computazione*[3].

Sia $M = (S, \rightarrow, L)$ un modello e sia $\pi = s_1 \rightarrow s_2 \rightarrow \ldots$ un cammino in M. La *relazione di soddisfacimento* di una formula LTL in un cammino π, indicata con $\models$, è definita nel seguente modo:

- $\pi \models \top$
- $\pi \not\models \bot$
- $\pi \models p$ *sse* $p \in L(s_1)$
- $\pi \models \neg\phi$ *sse* $\pi \not\models \phi$
- $\pi \models \phi_1 \wedge \phi_2$ *sse* $\pi \models \phi_1$ *e* $\pi \models \phi_2$
- $\pi \models \phi_1 \vee \phi_2$ *sse* $\pi \models \phi_1$ *o* $\pi \models \phi_2$
- $\pi \models \phi_1 \rightarrow \phi_2$ *sse* $\pi \models \phi_2$ *ogni volta che* $\pi \models \phi_1$
- $\pi \models X\phi$ *sse* $\pi^2 \models \phi$
- $\pi \models G\phi$ *sse* $\forall i \geq 1, \pi^i \models \phi$
- $\pi \models F\phi$ *sse esiste almeno un* $i \geq 1$ *tale che* $\pi^i \models \phi$
- $\pi \models \phi\, U\, \psi$ *sse esiste almeno un* $i \geq 1$ *tale che* $\pi^i \models \psi$ *e per ogni* j, $1 \leq j \leq i-1$, $\pi^j \models \phi$
- $\pi \models \phi\, W\, \psi$ *sse esiste un* $i \geq 1$ *tale che* $\pi^i \models \psi$ *e per ogni* j, $1 \leq j \leq i-1$, $\pi^j \models \phi$; *oppure se per ogni* $k \geq 1$, $\pi^k \models \phi$
- $\pi \models \phi\, R\, \psi$ *sse esiste un* $i \geq 1$ *tale che* $\pi^i \models \phi$ *e per ogni* j, $1 \leq j \leq i$, $\pi^j \models \psi$; *oppure se per ogni* $k \geq 1$, $\pi^k \models \psi$

[3]Si noti che l'albero di computazione sarà infinito in quanto nel modello, a partire dal quale si costruisce l'albero, ogni stato ha almeno un successore.

In particolare L'Until richiede che una certa formula ϕ sia vera fino a quando, ad un certo punto, diventa vera un'altra formula ψ.

Anche il Weak until richiede che una certa formula ϕ valga fino a quando diventa vera un'altra formula ψ, ma quest'ultima potrebbe anche non valere mai. Quindi, in un certo senso il Weak until è una forma più debole dell'Until.

Il Release è molto simile all'Until, tuttavia $\phi R \psi$ richiede che ψ sia vera anche nello stato in cui diventa vera ϕ. $\psi U \phi$, invece, richiede che ψ sia vera almeno fino allo stato precedente a quello in cui ϕ diventa vera.

Si può notare, inoltre, che, fatta eccezione per l'operatore X, per tutti gli altri il concetto di futuro include anche il presente, cioè lo stato attuale. Per questo motivo le formule $Gp \rightarrow p$, $p \rightarrow q \; U \; p$ e $p \rightarrow Fp$ sono valide.

La definizione di relazione di soddisfacimento riportata riguarda i cammini. Rispetto ad un modello diventa:

Definizione 5.3. Sia $M = (S, \rightarrow, L)$ un modello, $s \in S$ e sia ϕ una formula LTL. Si può dire che $M, s \models \phi$ se per ogni cammino π di M che ha come stato iniziale s, vale $\pi \models \phi$.

Sulla base di quanto visto sino ad ora si può dare una definizione che permetta di stabilire quando due formule sono equivalenti.

Definizione 5.4. Due formule LTL ϕ e ψ sono dette *semanticamente equivalenti* (o semplicemente equivalenti) se per ogni modello M e per ogni cammino π di M vale che: $\pi \models \phi$ se e solo se $\pi \models \psi$.

Viene indicato con $\phi \equiv \psi$.

Tra gli operatori della logica temporale esistono alcune relazioni. In particolare, i due operatori temporali F e G sono uno il duale dell'altro e X è il duale di se stesso:

$$\neg G\phi \equiv F\neg\phi \qquad \neg F\phi \equiv G\neg\phi \qquad \neg X\phi \equiv X\neg\phi$$

Anche U ed R sono uno il duale dell'altro:

$$\neg(\phi U \psi) \equiv \neg\phi R \neg\psi \qquad \neg(\phi R \psi) \equiv \neg\phi U \neg\psi$$

Esiste una relazione di equivalenza anche tra Until e Weak until:

$$\phi U \psi \equiv \phi W \psi \wedge F\psi \qquad \phi W \psi \equiv \phi U \psi \vee G\phi$$

Inoltre, valgono le seguenti uguaglianze:

$$\neg(\phi U \psi) \equiv \neg\psi W(\neg\phi \wedge \neg\psi) \qquad \neg(\phi W \psi) \equiv \neg\psi U(\neg\phi \wedge \neg\psi)$$

Dalla definizione e dal significato degli operatori R e W si può dimostrare che:

$$\phi W \psi \equiv \psi R(\phi \vee \psi) \qquad \phi R \psi \equiv \psi W(\phi \wedge \psi)$$

Come si può notare le due sottoformule ϕ e ψ risultano invertite. Inoltre, $\phi W \psi$ non richiede che ϕ sia vero nello stato in cui diventa vero ψ, l'importante è che valga fino allo stato precedente. Tale condizione, invece, è richiesta nel Release e questo giustifica la congiunzione nel passaggio da Release a Weak until e la disgiunzione nel procedimento inverso.

Inoltre, F e G si distribuiscono rispettivamente sulla disgiunzione e sulla congiunzione:

$$F(\phi \vee \psi) \equiv F\phi \vee F\psi \qquad G(\phi \wedge \psi) \equiv G\phi \wedge G\psi$$

Gli operatori F e G possono essere definiti rispettivamente mediante l'Until e mediante il Release:

$$F\phi \equiv \top U \phi^{4} \qquad G\phi \equiv \bot R \phi^{5}$$

Da queste equivalenze si può intuire che alcuni connettivi possono essere definiti per mezzo di altri e quindi la loro presenza è, in un certo senso, ridondante. Per quanto riguarda i connettivi logici si può dire che $\{\bot, \wedge, \neg\}$ forma un insieme adeguato (cioè $\vee$, $\rightarrow$, $\top$ possono essere ottenuti a partire da questi). Per quelli temporali, invece, valgono le seguenti considerazioni:

- X è ortogonale rispetto agli altri connettivi. Quindi, non si può ottenere dalla loro combinazione.
- Ciascun insieme $\{U, X\}$, $\{R, X\}$, $\{W, X\}$ è adeguato. Questo perché, utilizzando le equivalenze sopra elencate:

[4]Poiché $\top$ è sempre vero, questa formulazione dell'until equivale semplicemente a richiedere che prima o poi ϕ diventi vera.

[5]Poiché $\bot$ non diventerà mai vero, ϕ non verrà mai "rilasciato", cioè non diventerà mai falso.

- R e W possono essere definiti in funzione di U
- U e W possono essere definiti in funzione di R
- R e U possono essere definiti in funzione di W

Operatori riferiti al passato

Per esprimere formule che coinvolgono "istanti" passati si possono utilizzare i seguenti operatori:

- Y: che indica lo stato precedente (**Y**esterday)
- S: fino a quando nel passato (**S**ince)
- O: almeno una volta nel passato (**O**nce)
- H: sempre nel passato (**H**istorically)

Diversamente da quanto si potrebbe pensare e da quanto avviene nella logica CTL[6], gli operatori che si riferiscono al passato non aggiungono potere espressivo alla logica LTL. Questo perché permettono di considerare stati che possono comunque essere raggiunti proseguendo in avanti a partire dallo stato iniziale.

5.2 Il linguaggio grafico DCML

DCML è un linguaggio grafico pensato come strumento per la definizione di vincoli tra competenze che un certo insieme di curricula deve rispettare. Per poter eseguire questa verifica, ogni elemento espresso graficamente viene tradotto in una corrispondente formula LTL e testata su di un programma rappresentante i curricula, mediante tecniche di *model checking*.

Per la specifica di tali condizioni si può scegliere tra due tecniche alternative: specificare tutte le combinazioni lecite, oppure rappresentare solo i vincoli strettamente necessari. Come già precisato nella Sezione 4.5, la prima

[6]CTL *Computation Tree Logic* è una logica branching che permette di quantificare sui cammini per mezzo degli operatori A (quantificatore universale) ed E (quantificatore esistenziale). LTL e CTL non sono equivalenti, cioè ci sono formule esprimibili con la prima e non con la seconda e viceversa.

soluzione non si adatta affatto ad ambienti aperti e dinamici come il Web. Per questo, ispirandosi al *social approach* di Singh [44], occorre orientarsi verso la seconda alternativa definendo, ove necessario, condizioni sull'acquisizione delle competenze o sull'ordine relativo con cui debba avvenire.

La rappresentazione grafica scelta è ispirata a *DecSerFlow* (Declarative Service Flow Language) [46], sviluppato per la verifica, la specifica ed il monitoraggio di servizi web. DecSerFlow consente la definizione di vincoli tra attività componenti un servizio, ognuno dei quali corrisponde ad una "politica". In qualsiasi punto, durante l'esecuzione, si può controllare se ogni vincolo è rispettato o violato. Si dice, quindi, che il servizio rispetta o viola la corrispondente politica e si definisce "completamente corretto" se tutti i vincoli sono rispettati.

Similmente, parlando di personalizzazione di curricula di studi, è necessario garantire che un insieme di specifiche imposte sia rispettato. Quindi, come nel dominio dei servizi web, anche in questo caso ogni vincolo deve essere verificato in ogni punto del curriculum. In tal caso si può definire il piano di studi "corretto". È quindi necessario verificare tutti i curricula scartando, tra quelli proposti, quelli non conformi alle specifiche e permettendo all'utente di scegliere tra quelli corretti.

L'utilizzo della logica per la definizione di vincoli offre da un lato uno strumento formale e preciso e dall'altro la possibilità di utilizzare tali formule per verifiche automatiche.

Tuttavia, non è sempre facile intuire quale sia la formula adatta per esprimere un certo concetto che si ha in mente, né è immediato capire se una certa formula esprima esattamente quello per cui è stata pensata.

Una soluzione a questo problema, quindi, consiste nell'isolare un certo numero di vincoli particolarmente significativi ed interessanti in un certo dominio. Ognuno di questi viene quindi mappato in una corrispondente formula logica, accertandosi che esprima esattamente il vincolo che deve rappresentare.

DCML consente al designer di esprimere i vincoli scegliendo la forma grafica che rappresenta ciò che ha in mente, senza dover scrivere alcuna formula logica. La motivazione alla base di questo linguaggio grafico, infatti, è che "disegnare" vincoli dovrebbe essere più facile e dovrebbe fornire

una visione complessiva del modello che si sta creando. In questo modo, chi definisce i vincoli dovrebbe essere facilitato nell'accorgersi di eventuali inconsistenze o di condizioni necessarie a cui non aveva pensato in prima istanza.

Sono stati individuati tre tipi di vincoli principali: *before*, *implies* e *succession*. Questi possono essere combinati con gli operatori *not*, *immediately* e *not immediately*.

Lo scopo dell'immediately è quello di restringere il numero di stati entro il quale si può verificare una certa condizione, ad esempio, a quello immediatamente successivo o immediatamente precedente.

Di seguito ogni costrutto verrà descritto nel dettaglio.

5.3 Rappresentare competenze e vincoli base in DCML

L'obiettivo di DCML è quello di rappresentare le competenze e le relazioni che intercorrono tra esse, nel modo più facile ed intuitivo possibile per l'utente. Occorre, quindi, definire come rappresentare una competenza, come esprimere il livello a cui deve essere acquisita e come può essere legata ad altri concetti.

In questa sezione verranno presentate le forme grafiche utilizzate per definire singole conoscenze, vincoli sul livello, congiunzioni e disgiunzioni.

5.3.1 Rappresentazione delle competenze

In DCML le varie competenze vengono rappresentate mediante una coppia *(k,l)* racchiusa all'interno di un box (Figura 5.1), dove k indica una certa conoscenza ed l è un valore numerico intero che rappresenta il livello a cui k è richiesta in quel determinato vincolo[7].

Lo zero è il più piccolo valore utilizzabile. Esso rappresenta assenza di conoscenza.

[7] È possibile utilizzare altre scale, oltre a quella numerica, per indicare il livello. Ad esempio si possono usare le parole chiave *beginner*, *advanced* ed *expert*. Oppure, *very low low*, *medium*, *high*, *very high*, etc.

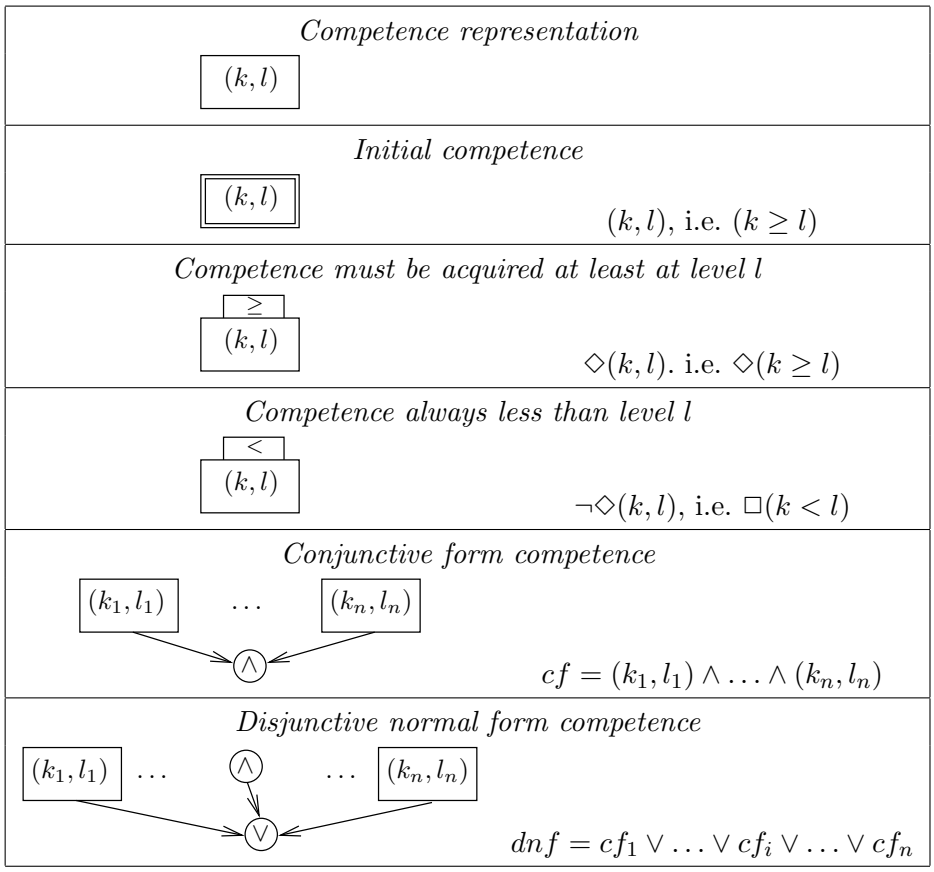


Figura 5.1: Rappresentazione delle competenze e delle relazioni base tra esse in DCML.

L'introduzione del livello è un'evoluzione rispetto al modo in cui era stato concepito inizialmente il linguaggio [5, 12]. Esso, infatti, permette di incrementare l'espressività dei vincoli che non si limitano a condizionare l'acquisizione delle conoscenze, ma anche il livello di dettaglio con cui devono essere affrontate.

L'assunzione di base fatta sul dominio in questione è che sia *monotono*. Il significato che si intende dare a questo termine è che una conoscenza, una volta entrata a far parte del bagaglio culturale dello studente, non può più essere rimossa.

Tale condizione viene espressa dalla formula logica $\neg k_1 \ \mathsf{U} \ \Diamond\Box k_1$ (in cui il simbolo " U " ha il significato di Weak Until cioè non forza l'acquisizione

di k_1).

Poiché si è deciso di associare un livello ad ogni competenza conseguita dallo studente, si deve imporre che tale valore non possa mai decrescere. La formula precedente può, quindi, essere utilizzata solo nel caso in cui la scala sul livello di acquisizione di una certo concetto sia *true*, *false*. Il problema viene risolto inserendo un controllo nel programma che simula il percorso di apprendimento dello studente, facendo in modo che il livello di una certa competenza venga aggiornato solo se il nuovo valore è superiore a quello precedente.

Può succedere, infatti, che lo studente segua prima un corso che fornisce (k_1, l_1) e poi ne affronti un altro che offre (k_1, l_2) con $l_2 < l_1$. Questo è un caso in cui il livello di k_1 non viene modificato con l_2. Intuitivamente, infatti, se lo studente conosce un certo argomento ad un livello abbastanza dettagliato, affrontarlo più volte con un dettaglio minore non aggiunge e non toglie niente alle sue conoscenze.

5.3.2 Esprimere competenze iniziali

Una conseguenza dell'aver scelto di rappresentare le competenze anziché i corsi, è che in questo modo è possibile che l'utente parta da un insieme di conoscenze non vuoto.

Si pensi, ad esempio, al caso in cui il curricula da verificare riguarda la pianificazione dei corsi per il secondo anno di una laurea triennale. In questo caso, è necessario imporre che lo studente abbia un insieme di nozioni che derivano dal percorso seguito l'anno precedente.

La raffigurazione scelta in DCML, per rappresentare che una certa conoscenza deve far parte del bagaglio culturale iniziale dello studente, consiste in un doppio riquadro all'interno del quale si inserisce la competenza ed il livello a cui è richiesta, (come si può vedere in Figura 5.1).

In logica LTL tale vincolo si traduce con la formula $k \geq l$. Come si può notare, in essa non compaiono operatori modali, questo significa che deve essere vera nel primo stato. Inoltre, k è una variabile il cui nome identifica una certa conoscenza ed il cui valore esprime il livello a cui è stata acquisita

dallo studente. Per questo ha senso scrivere disuguaglianze tra k e valori numerici.

5.3.3 Vincoli sul livello delle conoscenze

I vincoli esprimibili sul livello di una competenza sono di due tipi: *at least level* e *always less then level*.

Con il primo si intende imporre che la conoscenza k sia acquisita almeno a livello l. Graficamente si rappresenta aggiungendo il simbolo $\geq$ sopra il box corrispondente ed in logica questo si traduce con la formula $\Diamond(k \geq l)$. Questa richiede l'esistenza di un istante in cui k viene acquisita almeno a livello l. Poiché il dominio è monotono, da quello stato in poi la condizione $(k \geq l)$ sarà sempre rispettata.

Di significato opposto è il vincolo *always less then level*. Questo impone che il livello della conoscenza k sia sempre inferiore ad l. Il simbolo da apporre al box nella rappresentazione grafica è, come è facile intuire, il $<$.

La formula logica differisce dalla precedente, oltre che per il verso della disuguaglianza, anche per l'operatore modale utilizzato: $\Box(k < l)$. In questo caso, infatti, è necessario controllare che la condizione sia rispettata in ogni stato. Il caso particolare in cui $l = 1$, impone che la conoscenza non venga mai acquisita a livello superiore ad uno ma, come specificato precedentemente, il valore zero indica assenza di conoscenza. Di primo acchito, potrebbe sembrare strano esprimere una condizione di questo tipo. Tuttavia, il suo impiego sarà utile per la definizione di relazioni più complesse, che coinvolgono un certo numero di competenze e relazioni temporali tra esse.

Per entrambi, la rappresentazione grafica è riportata in Figura 5.1.

In questo modo, si può imporre,ad esempio, che la conoscenza *database_relazionali* sia acquisita almeno a livello 2 mentre per *database_OO* sia sufficiente un livello introduttivo, cioè almeno pari a 1.

5.3.4 Esprimere vincoli su più conoscenze

Vincoli più complessi possono essere ottenuti esprimendo relazioni su più conoscenze, anziché su una sola. Tuttavia, queste non possono essere

composte in qualsiasi modo, ma la struttura delle formule è limitata a quelle in *Forma Normale Disgiuntiva* DNF (Disjunctive Normal Form). Questo non pone nessuna limitazione alle formule che si possono esprimere, ma ne facilita l'interpretazione, la traduzione e soprattutto la gestione, limitando il numero di combinazioni possibili tra i concetti. In una formula, infatti, non è indifferente trattare prima le congiunzioni o le disgiunzioni.

La DNF è costituita da congiunzioni di competenze che vengono poste in alternativa tra loro: $dnf = cf_1 \vee \cdots \vee cf_n$. Da questa si può estrarre l'insieme dei congiunti che la compongono: $dnf = \{cf_1, \ldots, cf_n\}$. Quindi, utilizzare questa rappresentazione per le formule permette di stabilire un ordine con cui analizzare i diversi componenti dei vincoli: almeno un disgiunto deve essere vero e, perché questo accada, devono valere tutti i congiunti che lo compongono. Graficamente una disgiunzione viene rappresenta mediante il simbolo $\vee$ all'interno di un cerchio. Ad esso possono essere collegati, mediante frecce, sia singoli box, sia congiunzioni di competenze.

La congiunzione, invece, viene rappresentata con il simbolo $\wedge$ all'interno di un cerchio (Figura 5.1). A questo si possono collegare solo box rappresentanti le competenze, non si possono unire simboli di disgiunzione. Si possono, quindi, disegnare congiunzioni di concetti, ma non congiunzioni di disgiunzioni.

Indicando, quindi, con (k, l) il vincolo per cui $k \geq l$ e con $\neg(k, l)$ la condizione $k < l$, una congiunzione cf (conjunctive competence formula) di più competenze risulta essere $cf = (k_1, l_1) \wedge ... \wedge (k_n, l_n)$ e sostanzialmente esprime un vincolo sull'insieme $cf = \{(k_1, l_1), ..., (k_n, l_n)\}$.

Sulla base di questa definizione si possono stabilire le traduzioni per i vincoli di *existence*, *absence*, *negation* e *next*.

existence

L'*existence* viene utilizzato ogni qual volta si voglia specificare che uno solo o un insieme di concetti deve essere acquisito prima del termine del corso di studi. Più che una relazione o un vincolo tra conoscenze, rappresenta una condizione di obbligatorietà. In particolare impone il raggiungimento di un certo livello per una certa nozione. Si consideri, però, il caso

"$\mathsf{existence}(\{(k_1, l_1), (k_2, l_2)\})$", le competenze coinvolte sono solo due, tuttavia si possono presentare alcuni dubbi su quale debba essere la rappresentazione logica. Le possibili interpretazioni sono:

- $\Diamond((k_1, l_1) \wedge (k_2, l_2))$
- $\Diamond(k_1, l_1) \wedge \Diamond(k_2, l_2)$

La prima formula richiede che esista uno stato in cui per le due competenze sia raggiunto il livello richiesto. Nel secondo caso, invece, si richiede semplicemente che le due, prima o poi ed in modo indipendente tra loro, raggiungano il livello specificato. Quest'ultimo è proprio il significato che si intende dare al costrutto, tuttavia con entrambe le traduzioni si otterrebbe lo stesso effetto a causa della monotonicità del dominio.

Nella traduzione in formula logica si è scelto di mantenere la corrispondenza con il significato che si intende dare all'*existence*, cioè:

$$(\exists j, j \geq 1, \pi^j \models k_1 \geq l_1) \wedge (\exists y, y \geq 1, \pi^y \models k_2 \geq l_2)^8$$

La formula risultante è:

$$\mathsf{existence}(cf) = \bigwedge_{(k_i, l_i) \in cf} \Diamond(k_i, l_i)$$

Questo vincolo è molto importante per la certificazione di curricula di studio. Ad esempio, un esperto in *applicazioni web* non può non avere conoscenze a livello approfondito in ambito di sicurezza informatica, database, etc. Mediante il vincolo *existence* si possono specificare tutte le nozioni e il livello a cui debbano essere acquisite.

absence

L'*absence* può essere considerata l'opposto dell'*existence*. Lo scopo è imporre che ognuna delle conoscenze che compaiono in un certo insieme (eventualmente composto da un solo elemento) non venga mai acquisita a livello superiore di quello specificato.

[8] Questa definizione segue la notazione utilizzata da Gerard J. Holzmann in [28].

Per riprendere l'esempio precedente, potrebbe essere interessante imporre che lo studente interessato a diventare esperto in *applicazioni web* non scenda nel dettaglio di argomenti quali teoria della calcolabilità, probabilità e statistica, e così via.

Rispetto al caso precedente, quindi, l'operatore "$\Diamond$" deve essere sostituito con il "$\Box$". Anche così, però, si ripresenta il dubbio sulla traduzione:

- $\Box\neg((k_1, l_1) \wedge (k_2, l_2))$
- $\Box\neg(k_1, l_1) \wedge \Box\neg(k_2, l_2)$

A differenza dell'*existence*, la monotonicità del dominio non rende le due formule equivalenti. La prima, infatti, è verificata anche nei casi in cui $(\Diamond(k_1 \geq l_1) \wedge \Box(k_2 < l_2))$ e viceversa $(\Box(k_1 < l_1) \wedge \Diamond(k_2 \geq l_2))$. Esprimere il vincolo di assenza su una congiunzione di concetti, significa che per ogni k_i si deve mantenere un livello inferiore ad l_i.

La traduzione corretta è la seguente:

$$\mathsf{absence}(cf) = \bigwedge_{(k_i,l_i)\in cf} \Box\neg(k_i, l_i)$$

negation

Il costrutto *negation* non introduce alcun operatore modale. Questo non significa, però, che sia applicabile solo allo stato iniziale. Se così fosse, il suo impiego sarebbe limitato ai soli vincoli riguardanti le conoscenze iniziali dello studente. In realtà esso può essere utilizzato per esprimere condizioni complesse che coinvolgono più concetti. Ad esempio, verrà utilizzato nel vincolo temporale *before*.

L'obiettivo è esprimere che una certa competenza k_i in un determinato istante sia in possesso dello studente ad un livello inferiore a l_i. La differenza con il costrutto *absence* è che questo impone che tale condizione valga sempre. Con il solo costrutto *absence* non sarebbe possibile rappresentare vincoli che esprimono una condizione sull'istante attuale, come ad esempio: "se in questo stato non vale (k_i, l_i), allora deve valere (k_j, l_j)".

$$\mathsf{negation}(cf) = \bigwedge_{(k_i,l_i)\in cf} \neg(k_i, l_i)$$

next

Partendo da un certo stato (non necessariamente quello iniziale) l'operatore *next* esprime una condizione sullo stato successivo. Quindi, come nel caso precedente, la sua utilità è legata soprattutto alla combinazione con i costrutti che verranno descritti nelle sezioni successive di questo capitolo. Ad esempio verrà impiegato per rappresentare l'*immediately before*, per esprimere cioè la condizione per cui se in un certo stato lo studente ha acquisito una certa competenza, in quello immediatamente precedente deve valere una particolare condizione. Pertanto, l'impiego dell'operatore *next* risulterà più chiaro nel prosieguo del capitolo, quando verranno descritte le relazioni che si possono definire tra i concetti. Di seguito se ne riporta la definizione.

$$\mathsf{next}(cf) = \bigwedge_{(k_i,l_i)\in cf} \bigcirc(k_i, l_i)$$

I costrutti definiti fino ad ora non impongono relazioni tra conoscenze o ordini temporali sulla loro acquisizione. È giunto, quindi, il momento di chiedersi quali altri operatori abbia senso definire in questo dominio e quale traduzione in logica LTL sia giusto associare loro.

5.4 Vincoli temporali e relazioni tra le competenze

I vincoli di base descritti precedentemente e le varie formule *dnf* che si possono ottenere a partire dalle conoscenze base, possono essere combinati al fine di ottenere vincoli complessi mediante l'uso di tre tipi di relazioni:

before: ad indicare che un concetto deve essere appreso prima di un altro;

implication: l'acquisizione di una certa conoscenza impone il raggiungimento (prima o poi) di un'altra;

succession: se viene acquisita una certa conoscenza, allora un'altra deve essere acquisita successivamente.

Ognuno di questi può essere affiancato all'operatore immediately, il cui significato è stabilire una successione temporale stretta tra due eventi. Ad esempio, può indicare l'istante immediatamente precedente o successivo all'acquisizione di una certa competenza.

5.4.1 Before

Il *Before* impone un vincolo temporale sull'acquisizione delle competenze. Tuttavia si tratta di un ordine relativo in quanto esprime solamente che un concetto deve essere appreso prima di un altro, senza indicare né quando, né tanto meno quanto prima.

Nella prima parte sono stati descritti altri approcci in cui questo vincolo è riferito a corsi: "il corso A deve essere affrontato prima del corso B". La condizione in questo caso è simile, ma si ricorda che si è scelto di lavorare con una granularità più sottile, considerando le competenze.

Il vincolo "(k_1, l_1) *before* (k_2, l_2)" richiede, quindi, che la conoscenza k_1 almeno a livello l_1 sia acquisita prima del concetto k_2 a livello l_2. Il significato intuitivo è che (k_1, l_1) è un prerequisito di (k_2, l_2).

Inizialmente si potrebbe pensare che la soluzione più adatta sia quella di utilizzare l'Until (inteso come Strong Until nella notazione utilizzata in [28] e riportata nella Sezione 5.1.1): $\neg(k_2, l_2)\ \mathsf{U}\ (k_1, l_1)$. Questo è vero solo in parte. Esso, infatti, permette di esprimere che la condizione $k_1 \geq l_1$ deve verificarsi prima di $k_2 \geq l_2$, ma impone anche l'esistenza di uno stato che verifica la prima condizione, obbligando lo studente ad acquisire la conoscenza k_1. Dopo di che, non importa se ed a quale livello verrà conseguita la competenza k_2.

In questo modo, il vincolo:

$$(sql, l_1)\ \textit{before}\ (trigger, l_2)$$

esprimerebbe una relazione temporale sull'acquisizione dei due concetti, ma obbligherebbe anche lo studente ad imparare il linguaggio *sql*.

Con il costrutto *before*, invece, si vuole definire un legame tra due competenze, senza imporre l'acquisizione di alcuna di esse. Per questo si ricorre al Weak Until, la cui definizione è

$$\exists i \geq 1, \pi^i \models (k_1, l_1) \vee (\pi^i \models \neg(k_2, l_2) \wedge \pi^{i+1} \models \neg(k_2, l_2) \mathsf{U} (k_1, l_1))$$

che, come si può notare, esprime una sorta di condizione: "se lo stato i non verifica (k_1, l_1), allora anche (k_2, l_2) è falso e lo stesso varrà nello stato successivo".

Pertanto, la formula logica scelta per rappresentare tale vincolo è $\neg(k_2, l_2) \mathsf{U} (k_1, l_1)$ (dove il simbolo U rappresenta il Weak Until).

$$dnf_1 \ \textit{before} \ dnf_2 \quad \equiv \quad \bigvee_{cf_i \in dnf_1, cf_j \in dnf_2} \mathsf{negation}(cf_j) \mathsf{U} \ cf_i$$

Come si può vedere in Figura 5.2, la rappresentazione scelta per il vincolo *before* è una freccia che termina con un pallino. Intuitivamente questo dovrebbe rappresentare che la condizione è sul secondo concetto coinvolto. Infatti, (k_1, l_1) *before* (k_2, l_2) impone dei vincoli su k_1 solo se e fino a quando viene acquisita k_2. Altrimenti non vi è alcuna condizione su k_1.

Negative Before

Prima di capire come rappresentare in logica LTL il vincolo *not before*, occorre definire quale condizione si voglia imporre. Una possibile interpretazione è quella di vedere il *not before* semplicemente come duale del *before*. In tal caso, però, perché introdurre un nuovo formalismo quando si può ricorrere semplicemente al precedente?

Il *not before* assume, infatti, un altro significato. In un certo senso viene aggiunta una condizione: "(k_1, l_1) *not before* (k_2, l_2)" impone che k_1 non venga acquisito ad un livello superiore ad l_1 né prima né nello stesso stato in cui viene acquisito k_2 a livello l_2 (o a livello superiore).

La corrispondente formula è: $\neg(k_1, l_1) \mathsf{U} ((k_2, l_2) \wedge \neg(k_1, l_1))$ dove anche in questo caso l'Until è considerato come Weak Until.

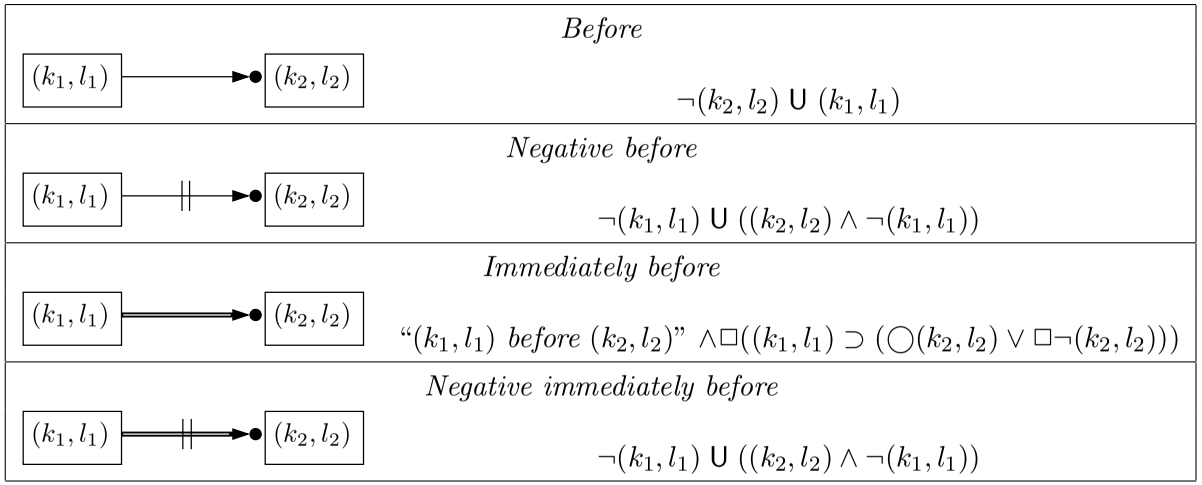


Figura 5.2: Rappresentazione grafica del vincolo Before.

Confrontando tale formula con quella del *before* si può notare che oltre a scambiare k_1, il quale ora compare negato prima del simbolo Until, e k_2, ora in forma positiva, si aggiunge la condizione che se esiste uno stato in cui vale (k_2, l_2), nello stesso stato deve valere $k_1 < l_1$. Anche in questo caso si sfrutta la monotonicità del dominio. Se così non fosse, infatti, il livello di k_1 potrebbe crescere e decrescere liberamente. Il fatto che questo non possa succedere, comporta che se in uno stato vale $(k_2 \geq l_2 \wedge k_1 < l_1)$ allora la competenza k_1 non ha mai raggiunto livello l_1.

Questo risultato non si ottiene semplicemente negando la formula logica che traduce il *before*. Il risultato che si otterrebbe da $\neg(\neg(k_2, l_2)\ \mathsf{U}\ (k_1, l_1))$ sarebbe $\neg(k_1, l_1)\ \mathsf{U}\ ((k_2, l_2) \wedge \neg(k_1, l_1))$ dove il Weak Until è stato trasformato in Until, che impone, quindi, l'acquisizione di (k_2, l_2). Per esprimere il significato che si vuole dare al *not before* è necessario indebolire l'Until riportandolo al Weak Until.

$$dnf_1 \ not\ before\ dnf_2$$

$$\equiv$$

$$\bigvee_{cf_i \in dnf_1, cf_j \in dnf_2} \mathsf{negation}(cf_i)\ \mathsf{U}\ (cf_j \wedge \mathsf{negation}(cf_i))$$

Graficamente la negazione si ottiene con due linee verticali sulla freccia che rappresenta il before, come si può vedere in Figura 5.2. Intuitivamente questo significa "rompere" la relazione.

Immediately Before

Di per sé il *before* rappresenta un ordine relativo sull'acquisizione delle conoscenze. Con l'aggiunta dell'aggettivo *immediate* si vuole limitare il numero di stati entro il quale deve verificarsi una certa condizione. In questo caso, poiché applicato al *before*, si esprime il vincolo secondo cui due concetti devono essere conseguiti in sequenza: (k_1, l_1) *immediate before* (k_2, l_2) richiede che se prima o poi viene acquisito (k_2, l_2), allora (k_1, l_1) deve essere stato acquisito nello stato immediatamente precedente. Si noti che il

raggiungimento di quest'ultimo non pone alcun vincolo di obbligatorietà sul conseguimento del precedente. Quindi, (k_2, l_2) potrebbe anche non valere mai.

Trovare la formula logica corrispondente non è immediato come nei casi precedenti. Sicuramente il solo Until non è sufficiente, deve essere affiancato all'operatore Next che è l'unico a permettere di esprimere una condizione su un particolare stato.

Un primo tentativo di traduzione potrebbe essere

$$(\neg(k_2, l_2) \mathsf{U} (k_1, l_1)) \wedge (\Diamond(k_1, l_1) \rightarrow \bigcirc(k_2, l_2))$$

Il motivo per cui questa formula è inadeguata è che la seconda parte della congiunzione non impone che (k_2, l_2) venga acquisito lo stato successivo a (k_1, l_1), semplicemente richiede che se in un certo stato è vero che prima o poi verrà acquisito (k_1, l_1), allora nello stato successivo deve essere vero (k_2, l_2). Poiché tale conoscenza non può essere acquisita se lo studente non è in possesso di (k_1, l_1), condizione imposta dall'Until, allora in ogni stato deve valere k_1 ed in quello successivo deve valere k_2.

Quindi, supponendo di partire dallo stato che contiene le conoscenze iniziali dello studente, imporre il vincolo:

$$(\mathit{database\_relazionali}, l_1) \ \mathit{immediate\ before}\ (\mathit{database\_OO}, l_2)$$

significherebbe richiedere che se prima o poi viene acquisito il concetto "*database_relazionali*", allora dopo il primo corso (cioè a partire dal secondo stato) lo studente deve avere conoscenze nell'ambito dei database object oriented e *database_relazionali* deve far parte del suo bagaglio culturale iniziale.

Si potrebbe cercare di rimediare spostando l'implicazione nel seguente modo:

$$\Diamond(k_1, l_1) \rightarrow (\bigcirc(k_2, l_2) \wedge (\neg(k_2, l_2) \mathsf{U} (k_1, l_1))$$

Anche così, però, non si riesce ad esprimere il vero significato dell'*immediate before*. Infatti, applicata all'esempio precedente, questa formula impone che se prima o poi viene acquisita la conoscenza *database_relazionali* allora non solo nel secondo stato deve valere la condizione $\mathit{database\_OO} \geq l_2$, ma poiché

deve essere verificato anche l'Until, secondo cui *database_OO* non può essere acquisito se prima non vale *database_relazionali* $\geq l_1$, nel secondo stato devono valere entrambe le conoscenze.

L'errore è chiedersi se esista uno stato in cui viene acquisita una certa conoscenza. Per arrivare alla traduzione corretta è necessario controllare che *in ogni stato* in cui $(k_1 \geq l_1)$, o in quello successivo $(k_2 \geq l_2)$ oppure k_2 non raggiungerà mai il livello l_2:

$$\Box((k_1, l_1) \supset (\bigcirc(k_2, l_2) \vee \Box\neg(k_2, l_2)))$$

Questa formula non controlla l'ordine di acquisizione tra le due competenze, k_2, infatti, potrebbe valere già prima di k_1. Pertanto, si deve aggiungere il vincolo *before*.

$$dnf_1 \textit{ immediate before } dnf_2$$

$$\equiv$$

$$\bigvee_{cf_i \in dnf_1, cf_j \in dnf_2} \text{“}cf_i \textit{ before } cf_j\text{”} \wedge\Box(cf_i \supset (\mathsf{next}(cf_j) \vee \mathsf{absence}(cf_j)))$$

Poiché l'*immediately before* può essere visto come una forma più forte del semplice before, la rappresentazione grafica scelta prevede che alla freccia che rappresenta il before venga aggiunta una linea, come ad enfatizzare la stretta sequenzializzazione con cui devono avvenire i due eventi.

Negative immediately Before

Per capire il significato che si vorrebbe dare a questo operatore occorre riprendere il legame che intercorre tra il vincolo *before* ed il *not before*. Come già precisato, il secondo non rappresenta la semplice negazione del primo. (k_1, l_1) *not before* (k_2, l_2) impone che k_2 venga acquisito prima e non nello stesso stato di k_1. (k_2, l_2) *before* (k_1, l_1), invece, non vincola l'acquisizione di k_2 ad uno stato strettamente precedente a quello in cui si ottiene la conoscenza k_1, ma impone semplicemente che non avvenga in un istante successivo.

Poiché l'*immediate before* mantiene uno stretto legame con il *before*, si vorrebbe conservare la stessa simmetria tra *not before* e *not immediate before*. Pertanto, (k_1, l_1) *not immediate before* (k_2, l_2) non può che voler imporre che (k_1, l_1) venga acquisito prima e nello stesso stato di (k_2, l_2). A ben guardare, questo non è che un caso particolare del *not before*. Poiché il dominio è monotono, infatti, dire che una conoscenza non può essere acquisita prima di un'altra include anche lo stato immediatamente precedente.

Per questo, il *not immediate before* si traduce esattamente allo stesso modo del *not before*.

$$dnf_1 \text{ not immediate before } dnf_2$$

$$\equiv$$

$$\bigvee_{cf_i \in dnf_1, cf_j \in dnf_2} \mathsf{negation}(cf_i) \ \mathsf{U} \ (cf_j \wedge \mathsf{negation}(cf_i))$$

La rappresentazione grafica deve conciliare sia l'aspetto *immediate* della formula, sia quello della negazione. Si ottiene, quindi, aggiungendo una riga alla freccia rappresentante il *before*, ottenendo l'*immediate before*, e due linee verticali per raffigurare il *not*, come si può vedere in Figura 5.2.

5.4.2 Implication

Di natura diversa è l'*implication*. Questo, a differenza del before, non rappresenta un vincolo temporale tra le competenze, ma stabilisce una condizione del tipo "se-allora" (*if-then*).

Proprio come avviene nella logica, nel caso dell'*implies* è l'antecedente a porre delle condizioni sull'acquisizione del conseguente. Infatti, con "(k_1, l_1) *implies* (k_2, l_2)" si vuole esprimere il vincolo per cui se la conoscenza k_1 viene acquisita almeno a livello l_1, allora k_2 deve essere acquisita almeno a livello l_2. Altrimenti, non si pongono vincoli su k_2.

La caratteristica dell'implicazione è che non specifica un ordine tra le due competenze: k_2 potrebbe già essere in possesso dello studente al momento dell'acquisizione di k_1, oppure potrebbe essere conseguito dopo o nello stesso stato.

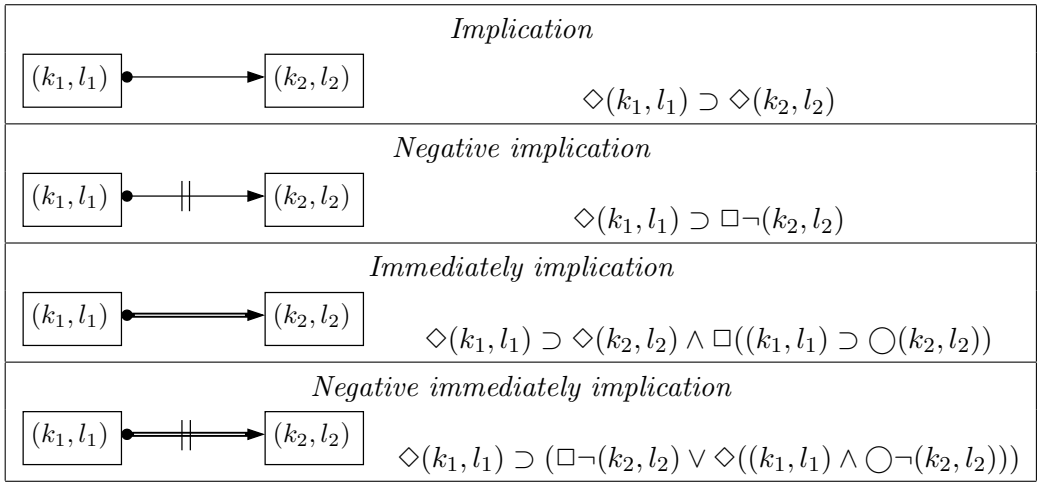


Figura 5.3: Rappresentazione grafica dell'Implication

In questo modo si può imporre che se viene acquista la conoscenza "*algoritmo_A**", prima o poi, per completezza, lo studente dovrà affrontare "l'*algoritmo_IDA**". In realtà, quando lo studente affronta il primo dei due, potrebbe già sapere in cosa consiste il secondo.

In [28] l'autore affronta il problema di come trasformare l'implicazione della logica classica nella corrispondente formula in logica LTL. In quel caso l'esempio viene utilizzato per evidenziare come spesso sia difficile trovare una traduzione adeguata e, soprattutto, corretta. Ripercorrere i vari passaggi potrà essere utile per arrivare alla formula che esprime il vincolo *implies*.

Supponendo di voler tradurre "(k_1, l_1) *implies* (k_2, l_2)" è evidente che la formulazione $(k_1, l_1) \supset (k_2, l_2)$ non può essere quella corretta. In essa, infatti, non compaiono operatori modali e, pertanto, impone una condizione solo sullo stato iniziale.

Anche l'aggiunta dell'operatore "$\Box$", ottenendo così "$\Box((k_1, l_1) \supset (k_2, l_2))$", non porta al raggiungimento del risultato sperato. In questo modo, infatti, si richiede che in ogni stato se vale k_1 valga anche k_2. Questo è vero solo nel caso in cui le due conoscenze vengano acquisite contemporaneamente.

Per risolvere questo problema si può ricorrere all'uso dell'operatore "$\Diamond$" nel seguente modo: $\Box((k_1, l_1) \supset \Diamond(k_2, l_2))$. Questa non rappresenta la tra-

duzione dell'implicazione in logica LTL[9], tuttavia si avvicina a quello che si vuole esprimere con questo vincolo. Essa significa che ogni volta che viene acquisita la competenza (k_1, l_1) deve essere, prima o poi, acquisita anche (k_2, l_2). Poiché il dominio è monotono, è possibile indebolire questa formula, controllando che se esiste uno stato in cui $k_1 \geq l_1$, allora, prima del termine del piano di studi dovrà valere anche $k_2 \geq l_2$. Tale risultato si ottiene con la seguente formula: "$\Diamond(k_1, l_1) \supset \Diamond(k_2, l_2)$".

$$dnf_1 \textit{ implies } dnf_2 \quad \equiv \quad \bigvee_{cf_i \in dnf_1, cf_j \in dnf_2} \mathsf{existence}(cf_i) \supset \mathsf{existence}(cf_j)$$

Graficamente l'*implies* si rappresenta con una freccia che inizia con un pallino per indicare da che parte è posta la condizione. Come si può notare, questa raffigurazione, riportata in Figura 5.3, è simmetrica rispetto a quella del *before*, dove la condizione è posta sui concetti che compaiono nella parte destra del vincolo. Nell'implicazione, invece, è sull'antecedente.

Un aspetto secondario di tale costrutto è che può essere utile per trovare errori nella definizione dei corsi. Si supponga, ad esempio, che il vincolo

$$(\textit{breadth-first\_search}, l_1) \textit{ implies } (\textit{depth-first\_search}, l_2)$$

non sia rispettato da un piano di studi, perché esso fornisce solo una delle due competenze. Supponendo, anche, che queste due strategie di ricerca debbano essere affrontate entrambe nello stesso corso di intelligenza artificiale, allora è evidente che il corso in questione è stato definito in modo non corretto. L'errore potrebbe essere nella descrizione del corso, sia a causa di un'omissione nell'elenco delle conoscenze fornite in uscita, sia perché effettivamente l'argomento non viene trattato.

Per la correzione si può procedere sostituendo il corso in questione perché non idoneo, oppure rimediare alla lacuna aggiungendo l'argomento al programma.

[9] La formula $a \rightarrow b$ corrisponde in logica LTL a "$\Box(a \supset \bigcirc(\Diamond b)) \wedge \Diamond a$".

Negative Implication

Come esprimere la negazione dell'implicazione è un problema che deve essere affrontato iniziando a definire quale situazione si vuole verificare con tale vincolo.

Infatti, $(k_1, l_1) \supset (k_2, l_2)$ richiede che se vale $\Diamond(k_1 \geq l_1)$ allora deve valere $\Diamond(k_2 \geq l_2)$. Considerando che la condizione che si vuole verificare con il costrutto *not implies* non necessariamente corrisponde alla negazione della formula logica che rappresenta il vincolo *implies*, le possibili interpretazioni sono tre:

1. Se l'antecedente non è verificato in alcuno stato, allora deve valere il conseguente: $\neg(k_1, l_1)$ *implies* (k_2, l_2).

2. Se l'antecedente è verificato, allora il conseguente dovrà restare falso fino al termine del corso di studi: (k_1, l_1) *implies* $\neg(k_2, l_2)$.

3. L'ultima alternativa nega l'acquisizione di tutti i concetti coinvolti nel vincolo: $\neg(k_1, l_1)$ *implies* $\neg(k_2, l_2)$.

Il criterio di scelta consiste nell'individuare la condizione più significativa per la rappresentazione di un vincolo, nel dominio dei curricula di studio.

La situazione che si è scelto di rappresentare mediante il costrutto *not implies* è quella delineata dal punto due. Essa, infatti, potrebbe essere usata per esprimere che due competenze sono equivalenti e il conseguimento di entrambe da parte dello studente porterebbe ad un dispendio inutile di tempo, oppure gli permetterebbe di accumulare crediti privandolo della possibilità di affrontare altri argomenti importanti.

La formula logica risultante si ottiene indebolendo la negazione dell'*implies*. Infatti, da:

$$\neg(\Diamond(k_1, l_1) \supset \Diamond(k_2, l_2))$$

si ottiene la formula:

$$(\Diamond(k_1, l_1) \wedge \Box\neg(k_2, l_2))$$

Questa, però, impone l'acquisizione di k_1 almeno a livello l_1. Per eliminare questo vincolo la congiunzione viene trasformata in una implicazione.

L'acquisizione di k_1 diventa così una condizione e non un'imposizione:

$$(\Diamond(k_1, l_1) \supset \Box\neg(k_2, l_2))$$

$$dnf_1 \textit{ not implies } dnf_2$$

$$\equiv$$

$$\bigvee_{cf_i \in dnf_1, cf_j \in dnf_2} \mathsf{existence}(cf_i) \supset \mathsf{absence}(cf_j)$$

Anche in questo caso la rappresentazione grafica si ottiene tramite due linee verticali sulla freccia che rappresenta l'implicazione (Figura 5.3).

Immediately Implication

L'implicazione, dunque, non esprime un vincolo temporale sull'acquisizione delle competenze ma rappresenta una sorta di relazione causa effetto.

Che significato può avere, quindi, l'affiancamento del "rafforzativo" immediately? Sicuramente l'*immediately implication* deve conservare alcune caratteristiche dell'implicazione. Anche in questo caso, quindi, la condizione è posta sull'antecedente. Rispetto a prima, però, ora si limita il numero di stati entro il quale può essere acquisito il conseguente.

Si supponga di avere il vincolo "(k_1, l_1) *immediate implication* (k_2, l_2)", con questo si esprime la condizione per cui se in un certo stato viene acquisito k_1 almeno a livello l_1, allora nello stato immediatamente successivo dovrà valere $k_2 \geq l_2$. Tale nozione potrebbe essere stata conseguita insieme o addirittura prima di k_1 (questo a causa della monotonicità del dominio). L'unico vincolo è che non venga assunto con più di uno stato di ritardo rispetto a k_1.

La semplice formula "$\Diamond(k_1, l_1) \supset \bigcirc(k_2, l_2)$", non rappresenta tale vincolo. Essa, infatti, viene verificata nel primo stato. Se (k_1, l_1) viene acquisita prima o poi impone il conseguimento di k_2 entro il secondo stato.

Aggiungere un operatore modale nel seguente modo non risolve il problema: "$\Diamond(\Diamond(k_1, l_1) \supset \bigcirc(k_2, l_2))$". Così, infatti, l'acquisizione di k_1 continua

ad essere del tutto slegata dallo stato in cui vengono verificate le condizioni "$\Diamond(k_1, l_1)$" e "$\bigcirc(k_2, l_2)$".

La formulazione corretta è:

$$(\Diamond(k_1, l_1) \supset \Diamond(k_2, l_2)) \wedge \Box((k_1, l_1) \supset \bigcirc(k_2, l_2))$$

Questa formula corrisponde all'implicazione con una condizione in più: $\Box((k_1, l_1) \supset \bigcirc(k_2, l_2)$, cioè deve essere sempre vero che, dal momento in cui viene acquisita k_1, dallo stato successivo dovrà valere anche $k_2 \geq l_2$.

$$dnf_1 \textit{ immediate implication } dnf_2$$

$$\equiv$$

$$\bigvee_{cf_i \in dnf_1, cf_j \in dnf_2} \text{“}cf_i \textit{ implies } cf_j\text{”} \wedge \Box(cf_i \supset \mathsf{next}(cf_j))$$

La stretta relazione che intercorre con l'*implies* viene rappresentata graficamente aggiungendo una linea alla freccia che rappresenta il vincolo dell'implicazione, come si può vedere dalla Figura 5.3.

Negative Immediately Implication

Il *negative immediately implication* è un costrutto il cui significato è quello di impedire che una certa competenza sia in possesso dello studente nello stato immediatamente successivo a quello in cui viene acquisita un'altra conoscenza. Più precisamente, "(k_1, l_1) *not immediate implies* (k_2, l_2)" controlla che (k_1, l_1) venga acquisita. Se questo non dovesse succedere, non è importante controllare a che livello e quando verrà acquisita k_2. Se invece la condizione dovesse verificarsi prima del termine del piano di studi, allora nello stato successivo in cui questo avviene deve valere $k_2 < l_2$.

Come per l'*implication* anche in questo caso si controlla il soddisfacimento di una certa condizione su una conoscenza. Dalla negazione dell'*immediately implication*, però, si ottiene la seguente formula:

$$\neg(((k_1, l_1) \textit{ implies } (k_2, l_2)) \wedge \Box((k_1, l_1) \supset \bigcirc(k_2, l_2))) \equiv$$

$$(\Diamond(k_1, l_1) \wedge \Box\neg(k_2, l_2)) \vee (\Diamond((k_1, l_1) \wedge \neg \bigcirc (k_2, l_2)))$$

Come si può vedere questa formula impone che lo studente raggiunga il livello l_1 per la conoscenza k_1, trasformando la condizione in un vincolo di obbligatorietà. Per ottenere l'effetto desiderato, quindi, occorre introdurre l'implicazione. La formula che si ottiene è:

$$\Diamond(k_1, l_1) \supset (\Box\neg(k_2, l_2) \vee \Diamond((k_1, l_1) \wedge \bigcirc\neg(k_2, l_2)))$$

Sostanzialmente essa controlla, nel caso in cui venga acquisita (k_1, l_1), che (k_2, l_2) non venga mai acquisita, oppure che esista uno stato in cui vale $k_1 \geq l_1$ e nello stato successivo $k_2 < l_2$.

$$dnf_1 \textit{ not immediate implies } dnf_2$$

$$\equiv$$

$$\bigvee_{cf_i \in dnf_1, cf_j \in dnf_2} \mathsf{existence}(cf_i) \supset \mathsf{absence}(cf_j)$$

Il vincolo si disegna aggiungendo una linea alla freccia che rappresenta l'implicazione, ottenendo così l'*immediate implication*, e due trattini verticali per rappresentarne la negazione (Figura 5.3).

5.4.3 Succession

Nelle sezioni precedenti sono stati descritti i costrutti *before* e *implication*, le rispettive negazioni e le forme contenenti l'immediately.

In un certo senso, il vincolo *succession* è una combinazione dei due precedenti: come il *before* si tratta di un vincolo temporale e come l'*implication* esprime una condizione sull'acquisizione di una conoscenza.

La necessità di introdurre tale costrutto deriva dal fatto che con i due precedenti non si riesce ad esprimere, in modo semplice, la condizione per cui se lo studente acquisisce una certa conoscenza, allora, in un istante temporale successivo, dovrà apprenderne un'altra.

In questo modo, si può esprimere, ad esempio, che dopo l'acquisizione di una certa competenza, lo studente ne veda un'applicazione o un esempio reale.

Il seguente vincolo:

$$(\mathit{query\_sql}, l_2)\ \mathit{succeeds}\ (\mathit{interrogazione\_database}, l_1)$$

impone, quindi, che dopo aver scoperto cosa significhi interrogare un database lo studente apprenda come eseguire una query con il linguaggio sql.

Con "(k_2, l_2) *succeeds* (k_1, l_1)" si intende esprimere la limitazione per cui se viene acquisita (k_1, l_1) in un certo istante, allora nel futuro si dovrà conseguire anche k_2 almeno a livello l_2. Altrimenti, non è importante se k_2 venga acquisito o meno ed a che livello. Questa è la vera differenza tra il vincolo succession ed il before, dove "(k_1, l_1) *before* (k_2, l_2)" impone che se (k_1, l_1) non viene mai raggiunto, allora neanche la competenza (k_2, l_2) potrà essere affrontata.

Poiché il costrutto *succession* deriva, in un certo senso, dall'unione dell'implicazione e del *before*, la prima soluzione che può venire in mente è di combinare le due definizioni anche per ottenere la formulazione in logica temporale:

$$((k_1, l_1)\ \mathit{implies}\ (k_2, l_2)) \wedge ((k_1, l_1)\ \mathit{before}\ (k_2, l_2))$$

che equivale a:

$$(\Diamond(k_1, l_1) \supset \Diamond(k_2, l_2)) \wedge (\neg(k_2, l_2)\ \mathsf{U}\ (k_1, l_1))$$

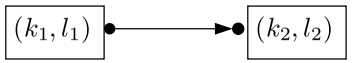
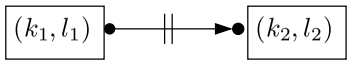
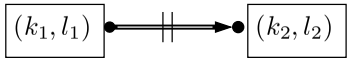
Succession	
	$\Diamond(k_1, l_1) \supset (\Diamond(k_2, l_2) \wedge (\neg(k_2, l_2)\ \mathsf{U}\ (k_1, l_1)))$
Negative succession	
	$\Diamond(k_1, l_1) \supset (\Box\neg(k_2, l_2) \vee$ "(k_1, l_1) *not before* (k_2, l_2)")
Immediately succession	
	"(k_1, l_1) *succession* (k_2, l_2)" $\wedge\Box((k_1, l_1) \supset \bigcirc(k_2, l_2))$
Negative immediately succession	
	$\Diamond(k_1, l_1) \supset (\Box\neg(k_2, l_2) \vee$ "(k_1, l_1) *not before* (k_2, l_2)" $\vee\Diamond((k_1, l_1) \wedge \bigcirc\neg(k_2, l_2)))$

Figura 5.4: Rappresentazione grafica del vincolo temporale Succession

In questo modo, però, si mantiene l'aspetto del *before* secondo il quale, la non acquisizione di (k_1, l_1) forza la non acquisizione anche di (k_2, l_2). Questo si vorrebbe evitare. Tra i due concetti esiste, infatti, una relazione temporale non dovuta al fatto che uno sia precondizione dell'altro, come avviene, invece, nel caso del before. Riprendendo l'esempio di prima, l'utente potrebbe essere in grado di eseguire una query sql su un database con l'obiettivo di reperire informazioni senza essere interessato a come venga soddisfatta la sua richiesta.

Anche in questo caso occorre riscrivere la formula sotto forma di implicazione:

$$\Diamond(k_1, l_1) \supset (\Diamond(k_2, l_2) \wedge (\neg(k_2, l_2) \,\mathsf{U}\, (k_1, l_1)))$$

$$dnf_2 \ succeeds \ dnf_1$$

$$\equiv$$

$$\bigvee_{cf_i \in dnf_1, cf_j \in dnf_2} \mathsf{existence}(cf_i) \supset (\mathsf{existence}(cf_j) \wedge \ \text{“}cf_i \ before \ cf_j\ )$$

Graficamente la succession viene rappresentata con una freccia che inizia e termina con un pallino (Figura 5.4). Intuitivamente questo richiama la rappresentazione dell'implicazione (simile alla succession per il comportamento condizionale) e del before (in questo caso la caratteristica in comune è il comportamento temporale).

Negative succession

Il costrutto *succession* impone dunque un vincolo temporale e condizionale. Con la negazione si intende modificare l'ordine con cui le competenze vengono acquisite. In particolare "(k_2, l_2) *not succeeds* (k_1, l_1)" impone il vincolo per cui se viene conseguita la conoscenza (k_1, l_1), allora o k_2 è stato acquisito prima oppure non raggiungerà mai il livello richiesto.

La formula che si otterrebbe negando il vincolo *succession* è:

$$\Diamond(k_1, l_1) \wedge (\Box\neg(k_2, l_2) \vee \neg((k_1, l_1) \ before \ \ (k_2, l_2)))$$

In questa formula $\neg((k_1, l_1)$ *before* $(k_2, l_2))$ viene interpretato come *negative before*: (k_1, l_1) *not before* (k_2, l_2).

Tuttavia, si può notare che essa impone l'acquisizione della competenza (k_1, l_1). Esso rappresenta un vincolo troppo forte rispetto a quello che si vuole imporre. Pertanto viene indebolito trasformando la congiunzione in un'implicazione.

$$dnf_2 \text{ not succeeds } dnf_1$$

$$\equiv$$

$$\bigvee_{cf_i \in dnf_1, cf_j \in dnf_2} \mathsf{existence}(cf_i) \supset (\mathsf{absence}(cf_j) \vee \text{“}cf_i \text{ not before } cf_j \text{)}$$

La *negative succession* graficamente si ottiene con due trattini verticali sulla freccia che rappresenta il vincolo *succession*. Figura 5.4.

Immediately Succession

Nel caso dell'*immediately succession* il vincolo temporale imposto diventa ancora più restrittivo. Se, infatti, il costrutto *succession* impone che l'acquisizione dell'antecedente avvenga dopo quella del conseguente, con l'aggiunta dell'*immediately* si impone che questo avvenga entro lo stato successivo. Quindi, possono essere acquisiti nello stesso stato, oppure in due stati consecutivi (rispettando comunque l'ordine).

La formula

$$\Diamond(k_1, l_1) \supset (\bigcirc(k_2, l_2) \wedge (k_1, l_1) \text{ before } (k_2, l_2))$$

non è adeguata ad esprimere il concetto “(k_2, l_2) *immediate succeeds* (k_1, l_1)”. Essa, infatti, controllata a partire dallo stato iniziale, verifica se prima o poi lo studente acquisisce la conoscenza (k_1, l_1). Se questo avviene, allora nel secondo stato deve valere $k_2 \geq l_2$ e k_1 deve essere acquisito prima di k_2.

Per trovare la traduzione corretta, è utile partire dalla formula che esprime il costrutto *succession* e aggiunge la condizione che restringa allo stato in cui viene acquisito k_1 e a quello successivo, il limite entro cui lo studente deve conseguire k_2.

La formula risultante è:

$$(k_2, l_2) succeeds (k_1, l_1) \wedge \Box((k_1, l_1) \supset \bigcirc(k_2, l_2))$$

$$dnf_2 \ immediate \ succeeds \ \ dnf_1$$

$$\equiv$$

$$\bigvee_{cf_i \in dnf_1, cf_j \in dnf_2} \text{“}cf_j \ succeeds \ cf_i\text{”} \ \wedge \Box(cf_i \supset \mathsf{next}(cf_j))$$

La rappresentazione grafica, segue quella scelta per gli altri costrutti: è sufficiente aggiungere una linea alla freccia che rappresenta il vincolo *succeeds* (Figura 5.4).

Negative Immediately Succession

L'ultima categoria di vincoli esprimibile tramite il linguaggio DCML è quella del *negative immediately succession*. Questo deve conciliare le caratteristiche del *succession* con quelle della negazione e dell'*immediately*. Esso, quindi, non può che voler rappresentare il caso in cui, se una certa conoscenza viene acquisita, un'altra non potrà essere acquisita nello stato successivo.

Più precisamente, "(k_2, l_2) *not immediate succeeds* (k_1, l_1)" esprime la condizione per cui se (k_1, l_1) viene acquisita in un certo istante, nello stato successivo deve valere $k_2 < l_2$ a meno che k_2 sia stato acquisito prima di k_1.

Dalla negazione dell'*immediately succession* si ottiene la seguente formula:

$$\Diamond(k_1, l_1) \wedge (\Box\neg(k_2, l_2) \vee ((k_1, l_1) \ not \ before \ (k_2, l_2)))$$

Innanzitutto, tale formula deve essere indebolita sostituendo la congiunzione con l'implicazione. Inoltre, bisogna aggiungere una condizione che controlli che l'acquisizione di (k_2, l_2), qualora non sia avvenuta prima di quella di (k_1, l_1), non avvenga proprio nello stato successivo.

La formula risultante è:

$$\Diamond(k_1, l_1) \supset (\Box\neg(k_2, l_2) \vee ((k_1, l_1) \ not \ before \ (k_2, l_2)) \vee \Diamond((k_1, l_1) \wedge \bigcirc\neg(k_2, l_2)))$$

$$dnf_2 \textit{ not immediate succeeds } dnf_1$$

$$\equiv$$

$$\bigvee_{cf_i \in dnf_1, cf_j \in dnf_2} \begin{array}{l} \mathsf{existence}(cf_i) \supset (\mathsf{absence}(cf_j) \vee \text{ ``}cf_i \textit{ not before } cf_j\text{''} \vee \\ \Diamond(cf_i \wedge \mathsf{next}(\mathsf{negation}(cf_j)))) \end{array}$$

Anche in questo caso la rappresentazione grafica si ottiene aggiungendo una linea alla freccia che rappresenta il vincolo *succession* e due trattini verticali per ottenere la negazione (Figura 5.4).

5.5 Caso di studio: esempio di curricula model

Il Curricula Model colleziona tutti i vincoli imposti dal designer, cioè tutte le relazioni e le precedenze temporali che devono intercorrere tra i concetti per poter affermare che il curriculum è corretto (o conforme).

In Figura 5.5 è riportato un esempio di curricula model definito mediante il linguaggio grafico DCML. Ogni vincolo verrà trasformato in una formula logica LTL per poter essere controllato. L'obiettivo è mettere in luce la facilità con cui si possono imporre i vincoli, pertanto il lettore potrebbe essere più o meno d'accordo con quelli imposti in questo esempio. Le relazioni che intercorrono tra i concetti, infatti, sono strettamente dipendenti da chi è incaricato della loro definizione e di quali siano i suoi obiettivi oltre, ovviamente, all'ambito di riferimento. In questo esempio, il curricula model coinvolge due ambiti nel contesto informatico: programmazione in Java e database. Il modello rappresentato nell'esempio, infatti, potrebbe essere diviso in due parti indipendenti, in quanto tra le competenze che compongono le due non intercorre alcun tipo di relazione o di vincolo.

Ogni box contiene almeno un concetto e il corrispondente livello di dettaglio. Se un box racchiude più di una competenza queste sono separate da una virgola, che deve essere interpretata come congiunzione. Nell'esempio tale rappresentazione viene utilizzata per le competenze *(error_handling,2)* e *(exception_handling,2)*. Si tratta, sostanzialmente, di una scorciatoia per rappresentare la congiunzione.

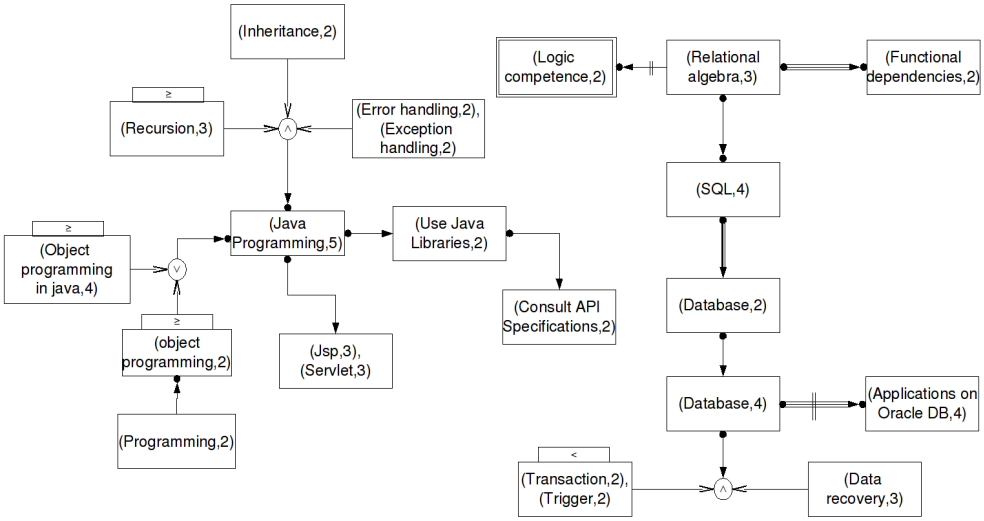


Figura 5.5: Esempio di curricula model

Queste due conoscenze, *(recursion,3)* e *(inheritance,2)* insieme rappresentano una precondizione all'acquisizione della competenza *(java_programming,5)*. Questo viene espresso mediante un simbolo di congiunzione a legare le prime, e un vincolo *before* (freccia che termina con un pallino) verso quest'ultima. Si noti che se la congiunzione non è rispettata non potrà essere acquisita neanche la competenza *(java_programming,5)*. Questa, infatti, è la condizione imposta dal vincolo temporale *before*.

Tale competenza è coinvolta anche in un altro vincolo temporale analogo. Essa, infatti, è collegata con un simbolo di disgiunzione che impone che lo studente debba possedere *(object_programming_in_java,4)* oppure semplicemente *(object_programming,2)*, prima di poter conseguire *(java_programming,5)*. L'obiettivo è imporre che lo studente, prima di affrontare la programmazione in Java ad un livello così approfondito, abbia conoscenze sulla programmazione ad oggetti, eventualmente riferita al linguaggio di programmazione Java.

Sul box che rappresenta la conoscenza *recursion* compare il simbolo "$\geq$". Questo impone che essa sia acquisita almeno a livello 3. Questo vale anche per "*object programming in Java*" e "*object programming*", sulle quali compare lo stesso simbolo.

Diversa è la condizione per le competenze *(transaction,2)* e *(trigger,2)*. Queste compaiono in un unico box, separate da una virgola. Si tratta, quindi, di una congiunzione. Il simbolo "<", però, impone che nessuna delle due raggiunga o superi il livello indicato.

Le altre competenze, per le quali alla rappresentazione non viene aggiunto alcun simbolo di disuguaglianza, compaiono semplicemente perché sono coinvolte in un vincolo, ma non si intende imporre la loro acquisizione.

L'implicazione compare, ad esempio, tra i concetti *(database,2)* e *(database,4)*. Quindi, se lo studente affronta l'argomento ad un livello abbastanza superficiale è obbligato, prima del termine del piano di studi, ad incrementarlo almeno fino al livello 4. In questo caso dovrà anche, prima o poi, ottenere le conoscenze *transaction* e *trigger* ad un livello inferiore a 2 e *(data_recovery,3)*.

Inoltre, l'acquisizione di *database* a livello 4 implica che *(application_on_Oracle_DB, 4)* non possa essere acquisito subito dopo. Come se per riuscire ad affrontare un'applicazione pratica di quanto appreso, lo studente dovesse lasciar trascorre un po' di tempo per avere, in seguito, una visione d'insieme più chiara sul funzionamento dei database.

I vincoli in cui compare il rafforzativo *immediately* usati in forma positiva, invece, sono utili per fare in modo che l'utente non dimentichi nulla di quanto appena appreso. Il dominio, infatti, è monotono, ma nella realtà spesso accade che il tempo tenda a scolorire le conoscenze degli studenti. Questo è il motivo per cui spesso conviene utilizzare vincoli come l'*immediate succession* oppure l'*immediate before*. Ad esempio, il primo viene utilizzato per imporre che la conoscenza *(functional_dependencies,3)* debba avvenire subito dopo quello di *(relational_algebra,3)*.

Tra *(relational_algebra,3)* e *(sql,4)*, invece, la successione non è così stretta ed è sufficiente che il secondo venga acquisito dopo il primo (secondo le regole descritte nella Sezione 5.4.3).

Infine, occorre notare che lo studente non può iniziare con un insieme di conoscenze vuoto, ma sin da subito dovrà avere competenze in logica; *(logic_competence,2)*, infatti, compare in un doppio riquadro.

Occorre ribadire che quello appena descritto è un esempio, il cui obiettivo è mettere in luce uno dei diversi modi con cui i vari vincoli possono

essere utilizzati. Pertanto, il lettore potrebbe non essere d'accordo con alcune delle relazioni espresse o semplicemente, potrebbe vederne altre molto più importanti. L'obiettivo di DCML è appunto quello di facilitare la definizione dei vincoli, e stimolare, mediante una visione d'insieme del modello, cambiamenti e nuovi spunti per la definizione di nuove relazioni.

Parte III

Rappresentazione e Verifica dei curricula di studio

Capitolo 6

Rappresentazione di curricula di studio

Un curriculum di studi è una sequenza di corsi che, a partire da un insieme di conoscenze iniziali, porta lo studente al raggiungimento di un insieme di competenze, senza violare vincoli imposti dal curricula model e senza che vi siano *competency gap*. Inoltre, l'utente deve avere la garanzia che al termine del percorso il suo *learning goal* sarà soddisfatto.

Secondo il paradigma introdotto in [9], ogni corso può essere visto come un'azione. Come tale sarà applicabile se le precondizioni sono soddisfatte e la sua esecuzione cambierà "il mondo", aggiungendo le conoscenze fornite come effetti. Il compito di un agente razionale, quindi, è quello di individuare un piano che, simulando l'apprendimento dello studente, permetta di raggiungere uno stato finale in cui valgano le condizioni specificate nel learning goal, e per raggiungere il quale non occorrano competency gaps.

La soluzione descritta in [9] prevede l'utilizzo del *procedural planning*, una tecnica di pianificazione molto efficiente. Le procedure vengono definite sulla base dei vincoli che un piano deve rispettare per poter fornire competenze adeguate per una specifica figura professionale. In questo modo si restringe lo spazio di ricerca alle sole sequenze che consentono di raggiungere quegli obiettivi.

Questo metodo fornisce una sequenza di azioni primitive che conducono dallo stato iniziale ad uno finale in cui valgono le richieste espresse nel learn-

ing goal dello studente. Tale sequenza può essere *lineare*, oppure può essere un piano *condizionale*, come si può vedere in Figura 4.2, dove ogni possibile strada tra cui l'utente può scegliere, rappresenta un'alternativa lecita, cioè non viola alcun vincolo.

I limiti di tale approccio, già discussi nei capitoli precedenti, sono legati alla sua *natura prescrittiva*, secondo la quale tutto ciò che è lecito deve essere specificato. Devono quindi essere descritte tutte le sequenze con cui i concetti possono essere acquisiti. Questo è tanto più dispendioso e difficile da gestire, quanto più il numero delle risorse è grande.

Pertanto, si è pensato di affrontare il problema da un'altra prospettiva: specificare solo i vincoli strettamente necessari, in analogia con il *social approach* suggerito da Singh e Yolum [50], per la definizione di protocolli per agenti. In questo modo, è sufficiente precisare che un certo concetto debba essere appreso prima di un altro, senza dover elencare tutti i possibili argomenti che possono essere affrontati tra i due.

Nel Capitolo 5 è stato presentato DCML, un linguaggio grafico per la definizione dei vincoli temporali, come il *before*, e delle relazioni che possono intercorrere tra i concetti, come *implication* e *succession*. Questi costituiranno il *curricula model*: un insieme di condizioni che deve essere rispettato da un piano di studi per poter essere ritenuto corretto.

In questa Parte, verrà affrontato il problema di come rappresentare i piani di studio e di come operare la verifica, per poter concludere che un certo curriculum è idoneo: non presenta competency gaps, rispetta il curricula model e permette all'utente di conseguire il suo learning goal.

Si presentano, infatti, due esigenze: quella di presentare i curricula ottenuti in seguito ad un'operazione di pianificazione e quella di permettere ad un utente di sottoporre il proprio piano per la validazione. In analogia con DCML, si è scelto di adottare una rappresentazione grafica. Questo dovrebbe facilitare l'utente molto più di una qualsiasi rappresentazione testuale.

6.1 Caratteristiche di una rappresentazione grafica adeguata

Una rappresentazione grafica adeguata deve permettere di esprimere tutti i possibili legami tra i corsi che compongono il piano. In particolare, essi possono essere disposti a formare una sequenza. In questo caso, quindi, si controlla che le precondizioni di un corso siano rispettare. Se così è, gli effetti dell'azione [9] potranno essere applicati al modello dello studente, utilizzato per la simularne l'apprendimento. Altrimenti è stato individuato un *competency gap*, per cui si può affermare che il piano non è valido.

Oltre alla semplice sequenza temporale con cui si possono susseguire i corsi, potrebbero esserci punti di scelta, originando così, piani alternativi. Tali scelte possono essere risolte dall'utente sulla base dei propri interessi, oppure possono essere vincolate da una condizione; ad esempio, se una certa conoscenza non rientra tra quelle acquisite, è necessario aggiungere uno o più corsi che rimedino alla lacuna, altrimenti questo non è necessario e lo studente può saltare tali lezioni per seguirne altre.

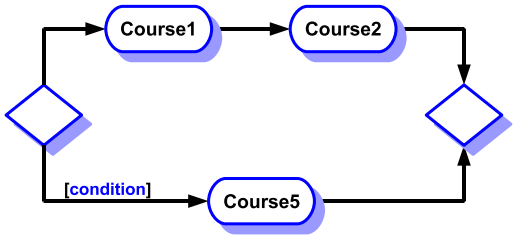
Infine, come normalmente accade nella realtà, più corsi possono essere seguiti in "parallelo". Con questo termine si intende che sono tenuti nello stesso periodo accademico (trimestre, semestre, etc.).

Per poter "disegnare" curricula di studi, quindi, è necessario trovare una rappresentazione che soddisfi queste esigenze. Una soluzione naturale consiste nell'adottare gli *activity diagrams* UML [40]. Essi permettono, infatti, di rappresentare sequenze, punti di scelta e attività in parallelo.

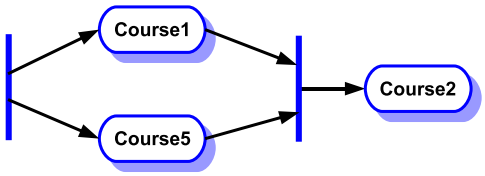
6.2 Rappresentazione dei curricula di studi mediante activity diagram: una prima proposta

Gli activity diagrams rientrano tra i diagrammi definiti dal linguaggio UML (Unified Modeling Language). Vengono generalmente utilizzati per modellare processi di business e per dettagliare alcune funzionalità di un sistema. In esso le attività, rappresentate in rettangoli con gli angoli arrotondati, definiscono una componente atomica all'interno di un flusso che

(a) Sequenza



(b) Alternativa



(c) Parallelismo

Figura 6.1: Possibili disposizioni dei corsi in un curricula di studi

realizza una determinata funzionalità di un sistema. Il flusso viene riprodotto tramite le frecce che collegano tra loro i rettangoli stabilendo, così, un ordine. Tuttavia, esso può essere arricchito con punti di *decisione*, *condizioni*, *fork*, *join*, etc.

Nell'ambito dei curricula di studio la corrispondenza con gli activity diagram è praticamente immediata. In particolare, ogni corso può essere visto come un'attività atomica, la cui esecuzione comporta il successivo adempimento dell'attività ad essa collegata nella sequenza [8].

In questo modo risulta molto semplice rappresentare sequenze di corsi, come si può vedere in Figura 6.1(a): si tratta semplicemente di disegnarli all'interno di rettangoli con gli angoli arrotondati e collegarli con le frecce, rispettando l'ordine che si vuole imporre. Si noti che, in questo modo, il tempo non viene rappresentato esplicitamente ma è dettato dalla durata del corso stesso.

Anche i punti di decisione vengono rappresentati utilizzando la notazione propria degli activity diagrams: un rombo divide il flusso in ingrasso verso le varie alternative. Ognuna di queste è costituita da un certo numero di corsi

e deve essere indipendente dalle altre. Per ogni ramo si possono annidare un numero indefinito di punti di decisione o di fork, oppure vi possono essere semplici sequenze di corsi. In Figura 6.1(b) è riportato lo schema generale. In esso compare anche una condizione su uno dei due rami. In questo modo si possono precisare condizioni su alcune conoscenze richieste allo studente. Quindi, se la condizione è rispettata lo studente potrà affrontare il corso etichettato come "Course5", altrimenti dovrà intraprendere l'altro ramo dell'alternativa. Al termine, i vari percorsi si uniranno in un altro rombo con un unico arco in uscita. Infatti, ad ogni punto di decisione deve corrispodere un rombo di chiusura dei rami alternativi. Nel caso non venga precisata alcuna condizione sui rami, significa che la scelta di quale percorso intraprendere è demandata all'utente. Non è, cioè, vincolata da alcun tipo di conoscenza.

Infine, come si può vedere in Figura 6.1(c) è possibile specificare che due corsi debbano essere affrontati contemporaneamente. Perciò, quando il processo di verifica raggiunge la linea verticale, che rappresenta una *fork*, dovranno essere controllate le precondizioni di tutti i corsi da affrontare in parallelo. In modo simile, in corrispondenza della linea verticale che rappresenta il *join* dei diversi rami, dovranno essere aggiunte tutte le competenze ottenute nei vari corsi. Come per la decisione, anche in questo caso, ad ogni simbolo di *fork* deve corrispondere un simbolo di *join* che ricomponga i due flussi in uno solo.

6.3 Una scelta adeguata per gli istanti di tempo

Gli activity diagrams permettono di rappresentare tutti gli elementi necessari per descrivere un piano di studi, vale a dire *sequenza*, *parallelismo* e *alternative*.

Il problema principale, della soluzione adottata per la rappresentazione è legato al tempo. In essa, infatti, non compare una divisione temporale esplicita, ad esempio in mesi o periodi accademici, ma la successione degli eventi è determinata solo dall'inizio e dal termine delle attività. Poiché ad ogni attività corrisponde un'azione (in accordo con l'approccio descritto nella Sezione 4.1), l'azione di frequentare il corso rappresenta anche l'unità minima di tempo.

Essa dunque, può essere utilizzata per la rappresentazione di sequenze e di semplici casi di alternative o di parallelismi. Si immagini, però, la seguente situazione:

corso A. Inizio: primo periodo. Durata: 2 periodi.

corso B. Inizio: secondo periodo. Durata: 2 periodi.

Questa non può essere rappresentata senza spezzare le due attività, in modo da rappresentare che il parallelismo avviene solo tra una porzione dei due corsi. Supponendo, quindi, di dividere il corso **A** in **A1**, per indicare la parte tenuta nel primo periodo, e **A2**, quella del secondo e in modo analogo dividere il corso **B**, una rappresentazione possibile è riportata in Figura 6.2.

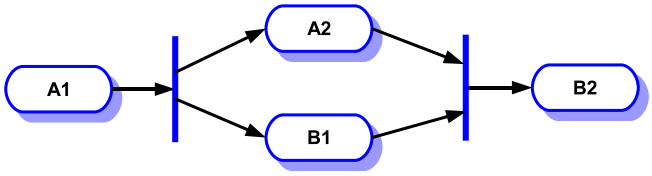


Figura 6.2: Rappresentazione del parallelismo tra due corsi della durata di due periodi. Questo esempio mette in luce la difficoltà di rappresentare il parallelismo tra corsi in questo approccio.

Questa soluzione risolve il problema, o così sembra, ma in questo modo si perde il concetto di "corso". In una situazione simile, infatti, si dovrebbe chiedere al professore di determinare l'elenco delle competenze attese in ingresso e quelle fornite in uscita, per ognuna delle parti in cui è stato diviso il suo corso. Inoltre, in questo modo il diagramma finale risulta meno leggibile e più difficile da definire.

Altrettanto complesso da risolvere è il problema simile applicato all'alternativa. Nel caso dei curricula di studio, infatti, con questo meccanismo si vogliono rappresentare veri e propri curriculum tra cui l'utente può scegliere. Si immagini, ad esempio, di voler rappresentare un curriculum informatico, per cui se lo studente vuole diventare esperto in *servizi web* dovrà seguire un certo percorso. Qualora decidesse di specializzarsi in *database*, invece, dovrebbe seguirne un altro. A questi si deve aggiungere una sequenza di corsi obbligatori per entrambi.

Con questo approccio è difficile rappresentare corsi che appartengono alla stessa specializzazione, ma che si tengono in momenti distanti nel tempo. In particolare, risulta complicato riportare la condizione per cui se l'utente nel primo periodo ha seguito un certo corso (ad esempio per il curriculum in database), allora dovrà affrontare certi argomenti collocati al terzo periodo, altrimenti dovrà intraprendere altri corsi tenuti nel secondo e nel quarto [1]. Tutto questo raffigurando anche i corsi obbligatori distribuiti lungo tutti i periodi.

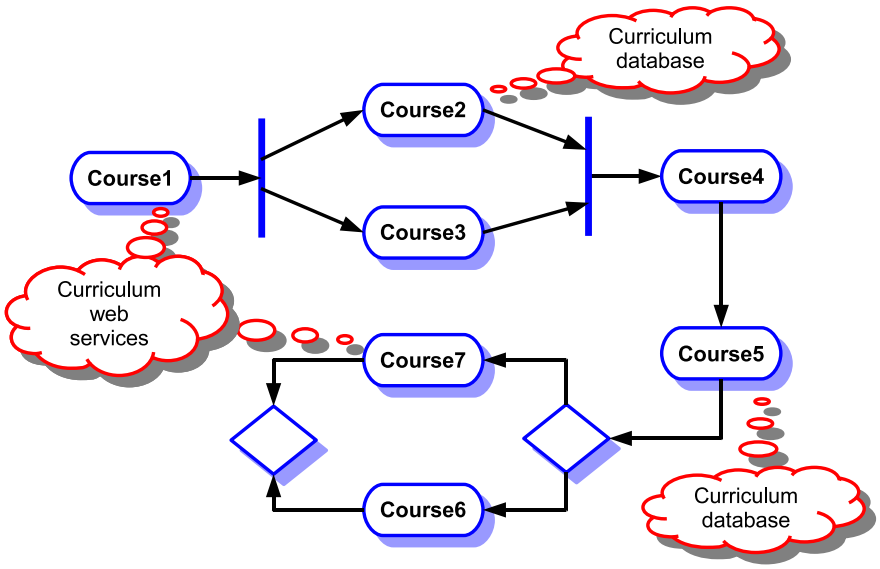


Figura 6.3: Esempio che illustra la difficoltà nel definire curricula alternativi con gli activity diagrams.

In Figura 6.3 è riportato un diagramma che rappresenta i due indirizzi *web services* e *database*, in concomitanza con i corsi obbligatori per entrambi. Le nuvolette indicano il curriculum di appartenenza del corso, ma non fanno parte delle rappresentazioni previste dagli activity diagrams. Pertanto, risulta difficile esprimere che lo studente deve affrontare i corsi 2 e 5 solo nel caso in cui abbia scelto di specializzarsi in database. In questo caso si

[1] Si ricorda che, poiché la fase di verifica opera a livello di competenze e non di corsi, le condizioni possono essere imposte solo su conoscenze che lo studente possiede o non possiede.

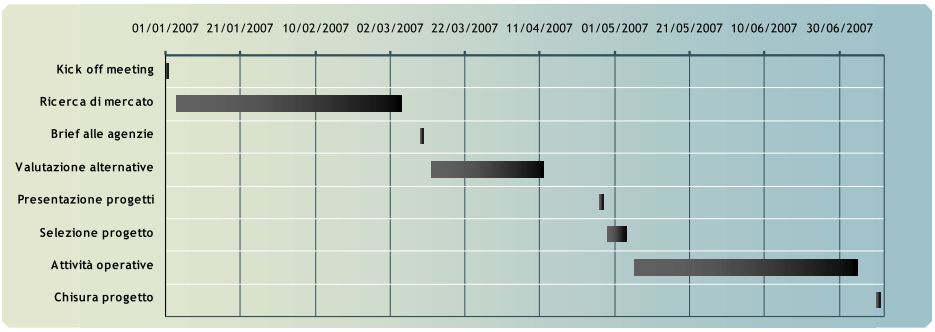


Figura 6.4: Esempio di un diagramma di Gantt per la pianificazione di un progetto aziendale.

può parlare di parallelismo per i corsi 2 e 3. Altrimenti, se è stato scelto l'altro indirizzo, si viene a formare una sequenza tra i corsi 1, 3 e 4. In tal caso, più avanti nella sequenza, anche il corso 5 dovrà essere saltato. Inoltre, solo in questo caso il rombo rappresenta un vero e proprio punto di scelta.

L'utilizzo degli activity diagram per la raffigurazione dei curricula presenta numerosi vantaggi. Innanzitutto, la corrispondenza tra attività e corsi semplifica la rappresentazione. Inoltre, questo tipo di diagramma prevede formalismi grafici appositi per la definizione di punti di decisione, condizioni e fork. Tuttavia, dai casi descritti in precedenza si deduce che la soluzione proposta non è sufficiente ed è necessario arricchirla con informazioni supplementari legate al tempo.

Uno spunto interessante può venire da un altro tipo di diagramma: i diagrammi di Gantt.

6.3.1 Diagrammi di Gantt

Il *diagramma di Gantt* è uno strumento di supporto alla gestione dei progetti in ambito aziendale. Esso viene utilizzato principalmente nelle attività di project management. È costituito da un asse orizzontale, che rappresenta il tempo suddiviso in fasi incrementali (ad esempio, giorni, settimane, mesi), e da un asse verticale che raffigura le mansioni o le attività che costituiscono il progetto.

Per rappresentare le sequenze, la durata e l'arco temporale di ogni singola

attività del progetto si utilizzano delle "barre" orizzontali. Queste possono sovrapporsi, indicando la necessità di svolgere alcune attività in parallelo. Un esempio è riportato in Figura 6.4. La semplicità di tale rappresentazione permette di capire immediatamente quando verrà intrapresa una certa fase del progetto e quale sia la sequenza con cui queste verranno affrontate.

Un diagramma di Gantt permette, dunque, la rappresentazione grafica di un calendario di attività, utile al fine di pianificare, coordinare e tracciare specifiche attività in un progetto, dando una chiara illustrazione dello stato d'avanzamento del lavoro.

La facilità con cui si possono esprimere attività che devono essere svolte in parallelo, induce a pensare che l'elemento mancante alla rappresentazione tramite activity diagrams, descritta nelle Sezioni precedenti, sia una suddivisione temporale esplicita. L'esempio in Figura 6.5 mostra un curriculum che non può essere rappresentato tramite activity diagrams. Tuttavia, il modo in cui sono stati disegnati i corsi nella parte alta della figura, non può essere considerato un vero e proprio diagramma di Gantt. Le risorse, infatti dovrebbero essere rappresentate lungo l'arco verticale, e la loro collocazione temporale dovrebbe essere espressa mediante le barre orizzontali. Aggiungere le attività, cioè i corsi, direttamente all'interno del periodo desiderato permette di semplificare la definizione e la lettura del diagramma finale.

Questa nuova rappresentazione permette di visualizzare in modo chiaro un curriculum di studi costituito da corsi in sequenza e in parallelo. Non è altrettanto facile esprimere punti di scelta e condizioni sulle conoscenze dello studente in un determinato istante. Questo tipo di diagramma, infatti, non prevede simboli per rappresentare queste situazioni.

In generale, quindi, i diagrammi di Gantt costituiscono un ottimo strumento per rappresentare il parallelismo tra i corsi ma non le alternative. Viceversa, gli activity diagrams permettono di esprimere percorsi alternativi e condizioni sugli archi, ma non contemplano meccanismi per la collocazione temporale delle varie attività.

La soluzione ideale sarebbe quella di mantenere gli aspetti vantaggiosi di entrambe le rappresentazioni. In effetti gli activity diagrams includono la possibilità di utilizzare partizionamenti orizzontali e verticali (*swimlanes* in UML [40]). Tale estensione permette di integrare i benefici della flessibilità

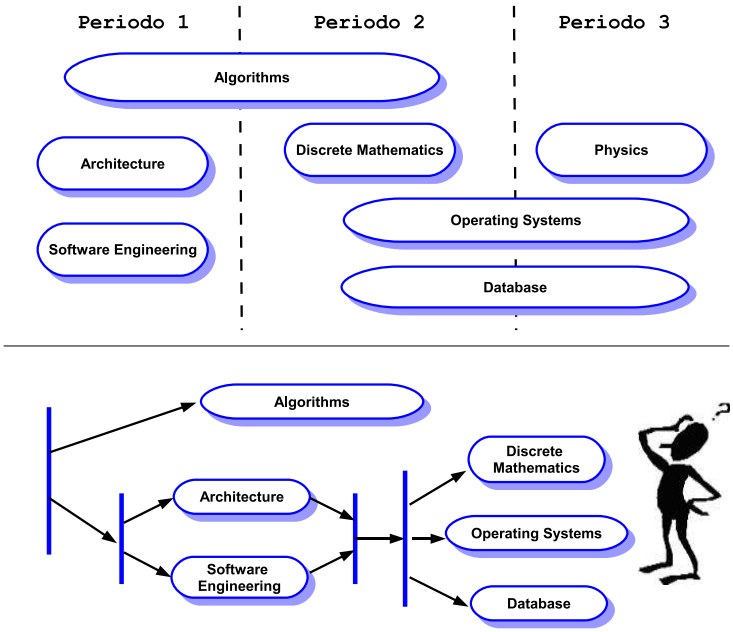


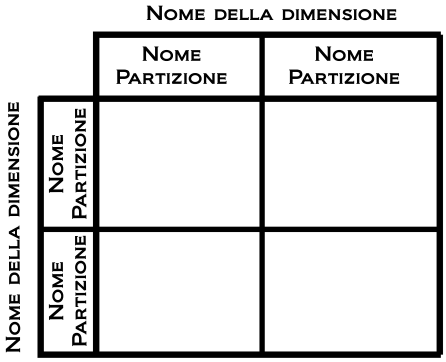
Figura 6.5: Esempio di curriculum non rappresentabile tramite activity diagrams

degli activity diagram con la facilità di collocare i corsi temporalmente dei diagrammi di Gantt, proprio ciò che serve per una corretta rappresentazione dei piani di studio.

6.4 Rappresentazione dei piani di studio mediante activity diagram: utilizzo di swimlanes e milestones

Prendendo spunto dai diagrammi di Gantt, si può pensare di introdurre delle partizioni verticali e orizzontali [2] alla rappresentazione descritta nella Sezione 6.2. Queste prendono il nome di *swimlanes* e sono solitamente utilizzate nei diagrammi per raggruppare le attività facenti parte dello stesso processo, oppure dipendenti dallo stesso responsabile. Si tratta di vere e proprie partizioni che possono essere organizzate in modo gerarchico e su più dimensioni, come si può vedere in Figura 6.6.

(a) Partizione gerarchica



(b) Partizione multidimensionale

Figura 6.6: Partizionamento di attività negli activity diagrams: utilizzo delle swimlanes.

Per la rappresentazione dei curricula di studio si utilizza una partizione multidimensionale, dove la suddivisione verticale fraziona la durata del curriculum in periodi. La suddivisione scelta, generalmente, segue quella con cui vengono forniti i corsi: trimestri, quadrimestri, semestri, etc. La linea verticale che rappresenta la partizione prende il nome di *milestone* e identifica il punto in cui deve essere applicata la verifica. Quindi, se nella proposta precedente era il tempo a dipendere dai corsi, ora sono i corsi ad essere distribuiti nel tempo.

Un corso può occupare più di una milestone nel caso in cui la sua durata sia superiore ad un periodo. Per ognuno, le precondizioni dovranno essere verificate in corrispondenza della milestone di inizio e gli effetti dovranno essere aggiunti solo in corrispondenza di quella di fine. Anche nel caso in cui il corso si estenda su più periodi, gli effetti potranno essere aggiunti solo al

termine dello stesso. Questo è in accordo con quanto previsto dalle specifiche UML [40]. Queste prevedono che, nel caso in cui un'attività si trovi a cavallo di una partizione, venga associata alla partizione immediatamente superiore nella gerarchia. Nella rappresentazione scelta non si utilizzano partizionamenti gerarchici, tuttavia un corso che occupa più milestones si può idealmente associare ad un periodo più lungo, composto da più sottoperiodi. La gerarchia, quindi, è implicitamente legata al tempo.

Come possono essere rappresentati, invece, i vari curriculum alternativi? Ogni curriculum può essere diviso in due parti: corsi obbligatori e corsi scelti dall'utente. Quelli obbligatori verranno estratti dal piano e rappresentati in una swimlane dedicata, etichettata come "*mandatory*". In essa possono comparire anche parallelismi e punti di verifica sulle conoscenze dello studente. Ogni altra swimlane orizzontale nel diagramma rappresenta una "porzione" di curriculum tra cui l'utente può scegliere. Tale piano, insieme a quello rappresentante i corsi obbligatori, costituisce il curriculum finale a cui lo studente dovrà adempiere.

Occorre notare che un corso può comparire in più piani alternativi, questo è possibile in quanto lo studente ne potrà scegliere uno solo tra questi. La stessa cosa non vale se il corso in questione compare tra quelli obbligatori. In questo caso non potrà comparire in nessuno degli altri curriculum.

Infine, è necessario precisare che la divisione temporale scelta rappresenta l'unità minima di tempo. In essa, quindi, non si possono rappresentare situazioni in cui lo studente debba seguire più di un corso, se non in parallelo. In altre parole, si escludono tutti quei casi (compresi punti di scelta a parallelismi) che porterebbero lo studente a seguire più di un corso in sequenza, all'interno della stessa milestone.

6.4.1 Caso di studio

L'introduzione di swimlanes e milestones permette di superare i limiti degli activity diagram nella rappresentazione di curricula di studio. Le prime consentono di distinguere chiaramente tra corsi obbligatori e piani tra cui l'utente può scegliere. Le milestones, invece, introducono esplicitamente il tempo all'interno del diagramma.

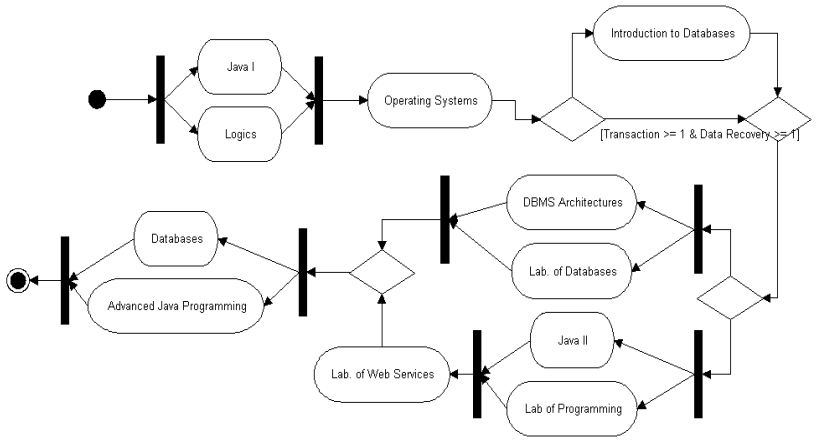


Figura 6.7: Rappresentazione di un curricula mediante activity diagram: i corsi non sono collocati nei vari periodi accademici e le alternative tra cui l'utente può scegliere si confondono tra i corsi obbligatori.

La Figura 6.7 riporta un piano di studi in cui compaiono punti di scelta, check point (con la verifica di una condizione), corsi in parallelo e sequenze. Osservando semplicemente la figura, è impossibile capire quali siano i vari piani in alternativa. Pertanto può essere ristrutturato con l'aggiunta delle swimlanes e delle milestones. Il risultato è visualizzato in Figura 6.8.

La prima swimlane, etichettata come "mandatory", rappresenta la porzione di piano obbligatoria. Ogni studente dovrà scegliere uno e uno solo tra i piani "additional". Sostanzialmente, quindi, ogni swimlane individua una porzione di curriculum, completamente indipendente dalle altre.

Per passare da una rappresentazione all'altra, non vi sono delle regole. La posizione in cui inserire un corso dipende dal corso stesso, dall'ente che eroga le lezioni, dal professore, etc.

Nello specifico, la prima alternativa fornisce competenze orientate ai database. La condizione "$transaction \geq 1 \wedge data\_recovery \geq 1$" controlla che tali competenze siano in possesso dello studente. In tal caso potrà saltare il corso "Introduction to databases". Per tale curriculum non sono previsti corsi aggiuntivi il secondo periodo, mentre al terzo saranno affrontati in parallelo quattro corsi: due del curriculum scelto e due *mandatory*. Si noti, dunque, che i corsi che si sovrappongono, pur non appartenendo alla stessa

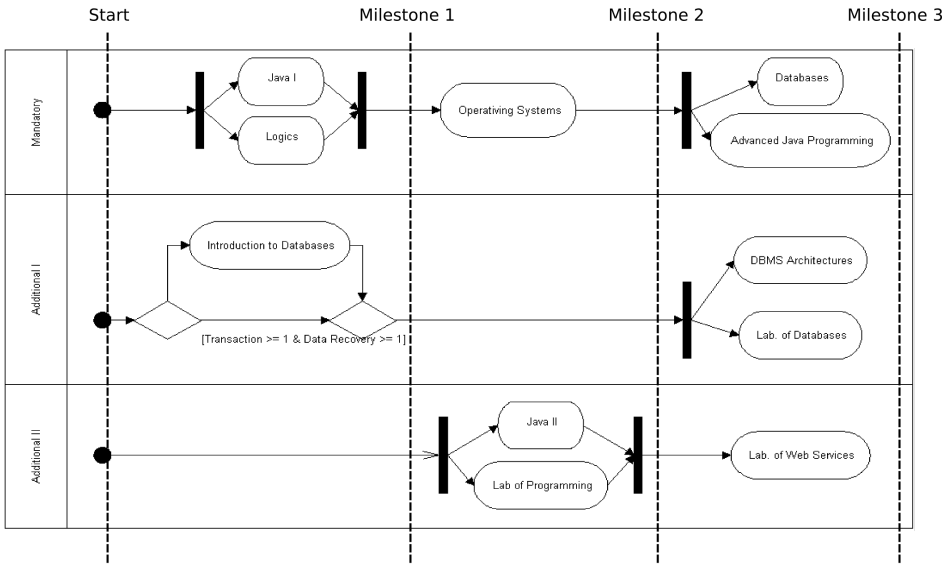


Figura 6.8: La rappresentazione finale dei curricula di studio prevede l'utilizzo di swimlanes, per distinguere i vari curriculum, e di milestones per rappresentare la collocazione temporale dei corsi.

swimlane, devono essere seguiti in parallelo.

Se lo studente dovesse scegliere l'alternativa "*additional II*" non avrebbe corsi aggiuntivi durante il primo periodo, ne avrebbe due il terzo periodo e uno l'ultimo.

La grande facilità di rappresentare curricula in questo modo, è che permette di distinguere nettamente tra i corsi obbligatori e i restanti. Quest'ultimi verranno composti in diversi modi formando diverse alternative tra cui l'utente può scegliere.

Capitolo 7

Verifica dei curricula di studio mediante il model checker SPIN

La verifica di piani di studio rappresenta l'obiettivo finale di questo lavoro. Essa consiste nel controllare (a) che un certo curriculum sia conforme rispetto al curricula model, (b) che permetta di raggiungere il learning goal definito dall'utente, (c) che non presenti competency gaps.

Il curricula model, espresso graficamente tramite il linguaggio DCML, fornisce un insieme di formule logiche LTL, nessuna delle quali deve essere falsificata nel piano in esame, pena la non idoneità di quest'ultimo. In tal caso, prima di poter essere sottoposto nuovamente alla verifica, il piano dovrà essere modificato al fine di rimediare agli errori riscontrati. A tale scopo, è necessario che lo strumento scelto fornisca un aiuto all'utente per l'individuazione del problema e non si limiti a rispondere circa la sua validità o meno.

La verifica deve essere preceduta da una fase di traduzione dei vincoli dal linguaggio grafico alla logica LTL. L'utilizzo di quest'ultima, infatti, è indispensabile per poter controllare in modo automatico la conformità dei piani rispetto al curricula model.

Il procedimento di verifica proposto in questa tesi, avviene mediante il ben noto model checker SPIN. In particolare verrà utilizzato XSPIN, che

offre all'utente la possibilità di interagire con il model checker per mezzo di un'interfaccia grafica. Esso permette, oltre alla verifica di comportamenti concorrenti tra processi, anche quella di proprietà esprimibili tramite formule logiche LTL [13]. Anche per i curricula, rappresentati graficamente mediante activity diagrams, occorre una traduzione verso il linguaggio PROMELA. Quest'ultimo, infatti, è il linguaggio utilizzato in SPIN per definire il sistema che si intende controllare.

In letteratura si possono trovare alcune proposte per la traduzione di diagrammi UML [30, 35], e nello specifico di activity diagrams [22, 25], nel linguaggio PROMELA. Tuttavia, la traduzione che sono in grado di offrire viene utilizzata per il debugging di diagrammi UML e non si adegua alla traduzione necessaria per la rappresentazione di curricula di studi. In questo caso, infatti, l'obiettivo è simulare il processo di acquisizione delle competenze dettato dai corsi che compongono il piano.

L'utilizzo del model checking offre numerosi vantaggi. Esso, infatti, in caso di fallimento restituisce un controesempio che può essere utilizzato per capire quale sia il problema del curriculum in questione. Ad esempio, ci si potrebbe accorgere che aggiungere un corso in un determinato punto permetterebbe di risolvere il problema.

Inoltre, poiché permette la verifica di formule LTL su sequenze rappresentanti lo stato di un programma, non occorrono fasi ulteriori di traduzione dei vincoli e ogni stato può essere interpretato come l'insieme delle conoscenze in possesso dello studente in quell'istante.

Un possibile punto a sfavore all'utilizzo del model checking in questo contesto, è dato dalla dimensione degli automi che vengono generati per la verifica, essa è strettamente dipendente dalla dimensione e dalla complessità (in termini di numero di operatori) della formula LTL da verificare. A tale proposito, però, occorre notare che le formule raccolte nel curricula model sono in congiunzione tra loro (vale a dire, devono essere tutte verificate per poter concludere che un curriculum è corretto). Pertanto, è possibile controllarne una per vola, ottenendo in questo modo automi di dimensioni contenute.

Prima di affrontare la traduzione di curricula di studi dal linguaggio grafico al linguaggio PROMELA, verrà descritto brevemente il model checking

e il funzionamento del model checker SPIN. Il lettore non interessato potrà saltare questa trattazione e passare direttamente alla Sezione 7.3.

7.1 Model checking in breve

Il *model checking* si occupa di verificare, in modo automatico, che un dato modello a stati finiti di un sistema rispetti un insieme di proprietà, descritte in un certo formalismo (ad esempio in logica temporale).

Oltre al model checking, altre tecniche di verifica molto conosciute e utilizzate sono il *testing*, la *simulazione* e le *verifiche formali*.

La *simulazione* si basa su un modello che descrive il possibile comportamento del sistema. Il "simulatore" è lo strumento di supporto in questa fase ed è in grado di determinare il comportamento del sistema in alcuni scenari. L'utente è così in grado di capire come il sistema reagisce quando gli vengono proposti alcuni stimoli. In genere viene utilizzato nelle fasi iniziali per una rapida analisi delle qualità di un disegno del sistema. Tuttavia, la sua utilità per la ricerca di errori significativi resta limitata in quanto è impossibile rappresentare tutti gli scenari rilevanti.

Il *testing* consiste nell'osservare le risposte del sistema a fronte di input opportunamente scelti (le scelte possono essere ad-hoc sull'implementazione che si sta analizzando, oppure si possono determinare mediante l'uso di euristiche). Quindi, si verifica che gli output prodotti siano conformi alle specifiche date. Il testing è il metodo più utilizzato, ma poiché si basa sull'osservazione, è possibile controllare un numero limitato di comportamenti del sistema. Non si tratta, quindi, di una tecnica completa: *"Il testing può mostrare solo la presenza di errori, mai la loro assenza"* (Dijkstra).

Un procedimento complementare è quello che va sotto il nome di *verifica formale*. L'idea è quella di costruire un modello che rappresenti i possibili comportamenti del sistema che si vuole studiare. I requisiti che esso deve soddisfare rappresentano il comportamento desiderato, mentre il modello del sistema rappresenta il comportamento reale. Si tratta, perciò, di verificare quali siano, se vi sono, i punti d'incontro tra le due specifiche. Questo è quello che avviene con questo tipo di verifica, che fornisce una vera e propria

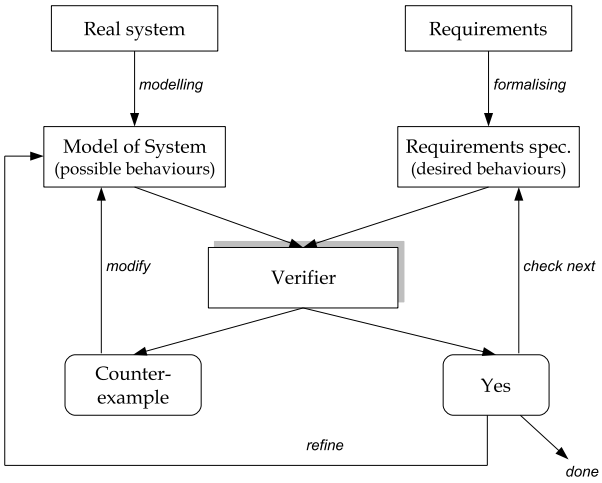


Figura 7.1: Processo di verifica nel model checking

dimostrazione di correttezza o di non correttezza del sistema, utilizzando un insieme di regole (*proof roles*).

Il *model checking* è un tipo di verifica formale basata su logica temporale [20, 31]. L'idea è quella di utilizzare algoritmi che permettano di verificare la correttezza di un sistema. L'utente, quindi, dovrà fornire al model checker, una descrizione del modello del sistema e una descrizione delle specifiche richieste. Se qualche proprietà viene violata, il sistema di verifica restituisce un controesempio che permette all'utente di capire il motivo per cui la verifica è fallita. Il controesempio consiste in uno scenario in cui il sistema si comporta in modo indesiderato. Questo processo è rappresentato in Figura 7.1.

Una prima distinzione riguarda il modo in cui vengono descritti i requisiti del sistema: si parla di *approccio logico o eterogeneo* nel caso in cui si faccia uso di una logica appropriata (in genere si tratta di una logica modale); si parla di *approccio basato sul comportamento o omogeneo* se i comportamenti possibili e quello desiderato vengono dati nella stessa notazione, ad esempio sotto forma di automi. Occorre fornire anche un criterio di equivalenza che i due automi devono rispettare. Quest'ultimo esprime relazioni del tipo "si comporta come" e "si comporta almeno come". Una definizione di questo tipo non è del tutto soddisfacente, in quanto può essere soggetta a diverse

interpretazioni. Per questo sono state date diverse nozioni di equivalenza. Una delle più usate è quella di *bisimulazione*, secondo cui due automi sono bisimili se uno riesce a simulare ogni passo dell'altro e viceversa. La nozione di *inclusione*, invece, stabilisce che un certo automa è incluso in un altro se ogni esecuzione accettata dal primo è accettata anche dal secondo. Infine, un sistema è considerato **corretto** se il comportamento desiderato e quello possibile sono equivalenti rispetto al criterio di equivalenza dato.

Un grosso vantaggio che deriva dall'utilizzo delle tecniche di model checking è che queste permettono verifiche parziali, considerando cioè solo un sottoinsieme dei requisiti iniziali. Ad esempio si può pensare di tralasciare le proprietà meno rilevanti al fine di velocizzare la verifica. Inoltre, queste tecniche possono essere applicate a qualsiasi dominio (Multi-agent systems, protocolli di comunicazione, software engineering, etc.) ed il loro utilizzo nella fase di disegno non richiede più tempo per la definizione di quanto ne richiedano il testing o la simulazione.

Gli svantaggi possono derivare dal fatto che la verifica viene fatta su di un modello e non sul sistema reale. Il fatto che un modello soddisfi certe proprietà, infatti, non garantisce che anche l'implementazione finale le rispetti e a volte trovare le necessarie astrazioni per esprimere i requisiti (ad esempio in logica temporale) può rivelarsi più complicato di quanto sembri. Tuttavia, la definizione di un modello consente di focalizzare l'attenzione sulle caratteristiche salienti del sistema tralasciando i dettagli implementativi. Inoltre, modificare un modello, nel caso in cui si trovino degli errori, è molto più facile e meno dispendioso rispetto a correggere una versione funzionante del sistema. Infine, occorre considerare il potenziale problema noto come *state-space explosion* che si verifica quando il numero di stati diventa troppo grande. Questo è più frequente quando si modellano sistemi paralleli in cui, nel caso peggiore, il numero degli stati è pari al prodotto del numero degli stati possibili per i due sistemi considerati separatamente. Quando il model checking fu introdotto (1981) era possibile modellare sistemi con al più alcune migliaia di stati. Con lo sviluppo di algoritmi più efficienti e di particolari strutture dati (attraverso rappresentazioni implicite), oggi si possono trattare sistemi con un numero di stati potenzialmente molto elevato, addirittura più di quanto sia effettivamente necessario per le verifiche che

devono essere fatte.

Riassumendo, quindi, le fasi di modellazione e di verifica attraverso model checking sono:

Definizione di un modello formale del sistema. Per il model checking il formalismo utilizzato è in genere definito mediante una struttura di Kripke.

Specifica di un insieme di proprietà mediante logiche temporali (LTL o CTL) formulate su attributi del modello definito precedentemente.

Verifica automatica delle proprietà mediante appositi strumenti (detti model checker).

7.1.1 Model checking di formule LTL

Gli algoritmi di model checking per formule LTL si occupano di stabilire, dati un modello $M = (S, \rightarrow, L)$, uno stato $s \in S$ e una formula LTL ϕ, se $M, s \models \phi$, cioè se ϕ è soddisfatta lungo tutti i cammini di M che iniziano in s. Se ϕ risulta essere non soddisfatta viene restituito un controesempio che dimostri che $M, s \not\models \phi$. Si tratta di un prefisso di lunghezza finita di un cammino di lunghezza infinita in cui ϕ non vale.

L'algoritmo presentato utilizza l'approccio basato su automi. In particolare si sfrutta il fatto che ogni formula LTL ϕ può essere rappresentata da un automa di Büchi non deterministico [29]. L'idea principale è quella di trovare un cammino π tale che $\pi \models \neg\phi$. Se esiste un cammino di questo tipo un prefisso di π viene restituito come controesempio. Se tale cammino non esiste, allora si può concludere che $M \models \phi$.

Lo schema in Figura 7.2 riassume i passi principali per la verifica di formule LTL.

Tali passi si possono riassumere nel seguente modo:

1. Costruire un automa per la negazione della formula, cioè per $\neg\phi$. Tale automa prende il nome di $A_{\neg\phi}$.

   La costruzione dell'automa ha la proprietà che per ogni cammino π del modello, $\pi \models \neg\phi$ se e solo se l'esecuzione del cammino π è accettata

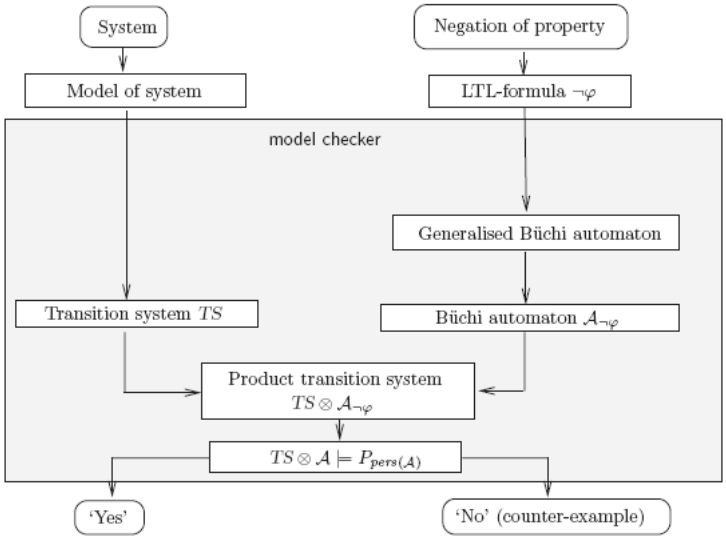


Figura 7.2: Model Checking in LTL

dall'automa $A_{\neg\phi}$. In altre parole, $A_{\neg\phi}$ accetta tutte e sole le esecuzioni che soddisfano $\neg\phi$.

2. Ottenuto l'automa $A_{\neg\phi}$, il passo successivo è quello di combinarlo con il transition system che rappresenta il modello del sistema. Il risultato è un transition system i cui cammini sono sia cammini dell'automa, sia cammini del sistema.

3. A questo punto occorre verificare se esiste un cammino, sul transition system ottenuto al passo precedente, che inizia in uno stato che deriva da s. Se esiste può essere interpretato come un cammino in M che inizia in s e che non soddisfa ϕ. Se un cammino di questo tipo non esiste, invece, possiamo concludere che $M, s \models \phi$

7.2 Il model checker SPIN

Insieme a NuSMV, SPIN è uno tra i model checker più conosciuti. Fu sviluppato da Gerard J. Holzmann [28] per la verifica di protocolli di comunicazione. Nel 2001 ricevette il premio ACM Software Systems Award.

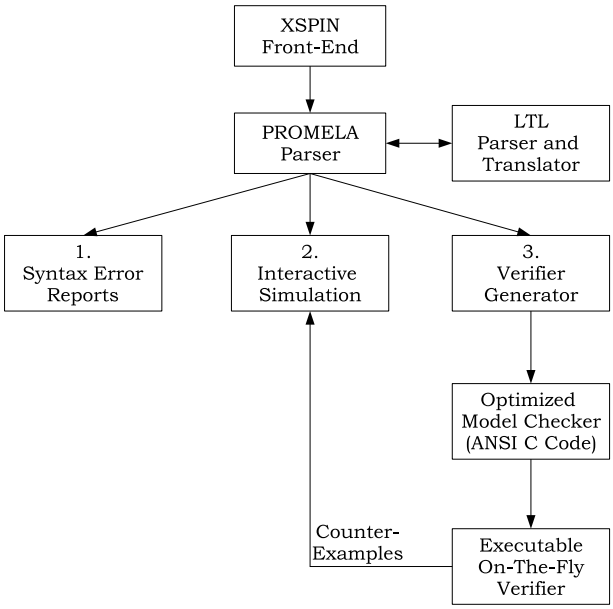


Figura 7.3: Verifica e Simulazione in SPIN

Il suo utilizzo è legato a momenti importanti dello sviluppo industriale. Ad esempio, tra il 1999 e il 2001 fu usato presso i Bell Laboratories per verificare il funzionamento di un nuovo switch telefonico. Inoltre, SPIN viene utilizzato per la verifica di algoritmi chiave nell'ambito di missioni spaziali della NASA.

SPIN è uno strumento che supporta la verifica automatica utilizzando la logica LTL. I modelli vengono descritti tramite il linguaggio PROMELA (PROcess MEta LAnguage) che permette di definire le interazioni sincrone o asincrone tra processi concorrenti, utilizzando lo scambio di messaggi, le variabili condivise oppure una combinazione delle due modalità. La sintassi è molto simile a quella del linguaggio C [13, 28].

La Figura 7.3 riporta uno schema che descrive le varie fasi di verifica e simulazione ad opera di SPIN.

L'utilizzo di questo strumento è facilitato dalla possibilità di interagire utilizzando un'interfaccia grafica (come ad esempio XSPIN[1] o JSPIN[2]). In questo modo, l'utente è agevolato nella definizione del modello. Dopo aver

[1]http://spinroot.com/spin/whatispin.html

[2]http://stwww.weizmann.ac.il/g-cs/benari/jspin/

verificato che non vi siano errori sintattici, è possibile passare ad una fase di simulazione interattiva, grazie alla quale l'utente può convincersi che il sistema è stato modellato correttamente, cioè si comporta come ci si aspetta. Infine, si possono verificare proprietà di alto livello.

In generale, il funzionamento di SPIN [27] è il seguente: l'utente può specificare delle procedure tramite la parola chiave "*proctype*". SPIN traduce ognuna di queste in un automa a stati finiti. Per ottenere il comportamento globale del sistema, viene fatto il prodotto degli automi utilizzando una politica di tipo asincrono e ad interleaving. Il risultato è un nuovo automa. Esso prende il nome di *state space* oppure, poiché può essere rappresentato come un grafo, di *reachability graph*.

Per effettuare la verifica, ogni formula LTL viene a sua volta trasformata in un automa di Büchi. Poiché verificare una certa formula significa controllare ogni possibile comportamento del sistema, quello che si verifica in realtà è che la negazione della formula non sia mai soddisfatta. Cioè, si controlla che l'intersezione tra il linguaggio riconosciuto dall'automa che rappresenta il sistema e quello identificato dalla negazione della proprietà, sia vuoto.

Il meccanismo usato, quindi, lavora con la negazione delle formule che si intende verificare e ne controlla il soddisfacimento mediante l'intersezione tra i linguaggi riconosciuti dagli automi.

7.3 Rappresentazione delle competenze e dei corsi nel linguaggio PROMELA

Per poter effettuare la verifica dei curricula, occorre definire un processo preciso e rigoroso di traduzione in linguaggio PROMELA. L'obiettivo è quello di simulare il processo di apprendimento dello studente tenendo traccia, nei vari istanti, delle conoscenze apprese e del rispettivo livello.

In questo procedimento è possibile individuare una struttura generale, che il programma deve avere e che si ripete indipendentemente dal curricula da rappresentare. In particolare, si possono riconoscere la fase di dichiarazione delle variabili, la definizione dei corsi e l'organizzazione di questi in milestone. I primi due aspetti possono essere definiti una sola volta

ed essere riutilizzati per la verifica di diversi piani. Di seguito, si affronterà con maggior dettaglio questo procedimento. La definizione delle milestone, invece, verrà trattata nella Sezione 7.4.

Per quanto riguarda la definizione dei vari concetti coinvolti nel percorso di apprendimento dello studente o nella definizione dei corsi, la rappresentazione scelta deve consentire la memorizzazione del livello a cui una certa competenza è stata acquisita (eventualmente zero, ad indicarne l'assenza). Pertanto, la soluzione adottata prevede la definizione di ogni competenza con una variabile intera inizializzata a zero, il cui nome indica il concetto che deve rappresentare e il valore indica il livello a cui è in possesso dello studente. Questa rappresentazione facilita anche la gestione della monotonicità del dominio (una volta acquisite, le conoscenze non possono essere rimosse) che impone che i valori non possano decrescere.

```
1 int
2   competenza_1 = 0,
3   competenza_2 = 0,
4   ...
5   competenza_n = 0;
```

Lo stato iniziale viene rappresentato mediante l'inizializzazione delle variabili, per cui solo quelle che fanno parte sin da subito del bagaglio culturale dello studente avranno valore maggiore di zero. Ad esempio, se il programma descrive un piano di studi per il secondo anno di università, l'insieme delle conoscenze iniziali conterrà sicuramente tutte quelle fornite dai corsi superati l'anno precedente. Tuttavia, esiste anche la possibilità di partire da un insieme di conoscenze iniziali vuoto. Questa situazione rappresenta, ad esempio, i casi in cui non si abbiano informazioni sull'utente. Quello che si vuole verificare in tal caso, è la conformità di un curriculum rispetto al curricula model, indipendentemente da chi ne usufruirà.

La procedura "*SetInitialSituation*" viene invocata una sola volta e serve proprio per modificare il valore delle variabili per le quali il valore iniziale è superiore a zero, cioè quelle che definiscono lo stato iniziale.

```
1 inline SetInitialSituation(){
2     atomic{
3         SetCompetenceState(competenza_1, livello_1);
```

```
4           SetCompetenceState(competenza_2, livello_2);
5           ...
6       }
7   }
```

Per quanto riguarda i corsi, invece, in accordo con il paradigma ad azioni, si possono distinguere due fasi importanti: la verifica delle precondizioni e l'applicazione degli effetti. La prima consiste semplicemente nel verificare che le conoscenze richieste in ingresso abbiano, nell'istante in cui viene affrontato il corso, un valore almeno pari a quello necessario. Questo viene realizzato mediante una semplice *assert*:

```
1 inline preconditions_course_nomeCorso() {
2   assert(competenza >= livelloRichiesto);
3 }
```

Il meccanismo alla base di questo confronto è che non è possibile asserire una condizione falsa. Pertanto, se la competenza dovesse essere in possesso dello studente ad un livello inferiore a quello richiesto, questa semplice istruzione farebbe terminare la verifica con un fallimento. Nel caso in cui il corso non abbia precondizioni sarà sufficiente sostituire il confronto con "*true*", se, invece, dovessero essere più di una, verrebbero rappresentate tutte all'interno della stessa assert, legate da una congiunzione.

Gli effetti dei corsi agiscono direttamente sullo stato, modificando i valori delle variabili. In particolare, per ognuno viene richiamata la procedura "*SetCompetenceState(concetto,livello)*" il cui compito è quello di controllare che la variabile rappresentante il concetto abbia, nello stato attuale, un valore inferiore a quello fornito dal corso. In tal caso il livello viene aggiornato, altrimenti non viene apportata alcuna modifica.

```
1 inline effects_course_nomeCorso() {
2   SetCompetenceState(competenza_1, livello_1);
3   SetCompetenceState(competenza_2, livello_2);
4   ...
5 }
6
7 inline SetCompetenceState(competence, lev){
8     if
```

```
9       :: (competence < lev) ->
10              competence = lev;
11      :: else -> skip;
12      fi;
13  }
```

La traduzione di un ipotetico corso di database potrebbe essere la seguente:

```
1  inline preconditions_course_database_I() {
2          assert(true);
3  }
4
5  inline effects_course_database_I() {
6          SetCompetenceState(DBMS_architecture,3);
7          SetCompetenceState(entity_relationship_model,3);
8          SetCompetenceState(b_trees,3);
9          SetCompetenceState(indexes,3);
10         SetCompetenceState(relational_model,3);
11         SetCompetenceState(sql_language,2);
12 }
```

Come si può notare, un corso definito in questo modo, non richiede alcuna competenza in ingresso, ma fornisce, come effetto, sei conoscenze, ognuna ad un determinato livello di dettaglio.

La definizione dei corsi segue, quindi, uno schema rigido e preciso, ma soprattutto indipendente dalle successive fasi di verifica. Per questo motivo, si può pensare di anticipare l'operazione di definizione delle procedure per la verifica delle precondizioni e per l'aggiunta degli effetti. Una volta definiti, infatti, i corsi non dovrebbero subire variazioni frequenti. Anche perché, in tal caso, utenti e professori resterebbero spiazzati e dovrebbero spendere molto tempo per restare al passo delle modifiche. Tali procedure potranno essere utilizzate in fase di verifica. Pertanto, la parte più importante della traduzione di curricula di studio consiste nella definizione della successione con cui si alternano i vari corsi.

7.4 Traduzione delle milestones che compongono i curricula

La rappresentazione vera e propria avviene, come sottolineato in precedenza, con la traduzione delle *milestone*. In generale, infatti, queste rappresentano punti in cui deve essere effettuata una qualche verifica: in questo contesto, ogni milestone rappresenta il periodo che si sta concludendo.

Innanzitutto, quindi, occorre analizzare la rappresentazione grafica del curricula è stabilire il numero di periodi in cui è organizzato. Quindi, la struttura generale del programma viene definita dalla seguente procedura:

```
1 proctype CurriculumVerification() {
2     milestone_1();
3     milestone_2();
4     ...
5     milestone_n();
6     LearningGoal();
7 }
```

Essa consiste nell'elenco delle milestones e nella successiva verifica del learning goal.

Ora è importante analizzare come vengano tradotte le varie milestones.

In particolare si possono individuare due fasi distinte: la verifica delle precondizioni e l'applicazione degli effetti dei corsi. Queste due devono avvenire esattamente in questo ordine.

Il processo di traduzione inizia, come è facile aspettarsi, dalla prima milestone in ordine cronologico, quindi quella a sinistra nel diagramma.

7.4.1 Controllo delle precondizioni

La verifica delle precondizioni inizia dai corsi che si trovano nella swimlane *mandatory*. In base alla loro disposizione la traduzione cambia. In particolare ci si può trovare di fronte a tre casi principali:

Corso singolo

Rappresenta il caso in cui nella milestone in questione vi sia un solo corso (Figura 7.4). In tal caso, è sufficiente richiamare la procedura corrispondente

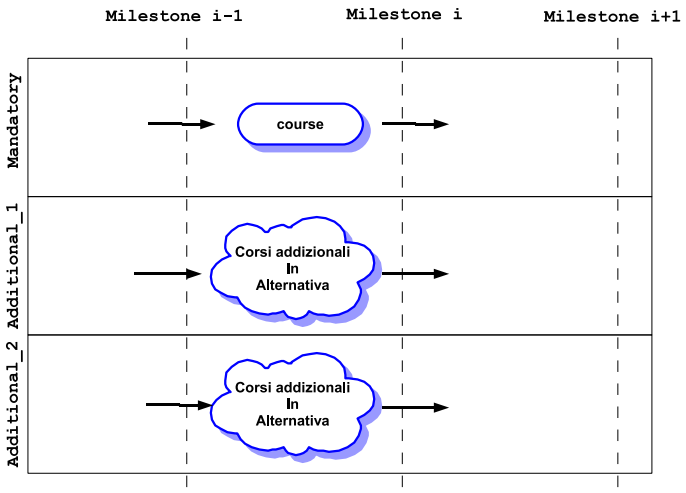


Figura 7.4: Rappresentazione di un corso nella swimlane mandatory

per la verifica delle precondizioni. A tal proposito, occorre sottolineare che la suddivisione temporale scelta, impone che in una milestone non vi siano corsi da seguire in sequenza: per ogni milestone lo studente potrà seguire al più un corso, oppure sequenze di lunghezza uno in parallelo.

```
1 proctype milestone_i() {
2     preconditions_course_1();
3
4     (precondizioni corsi in alternativa)
5     (effetti corsi obbligatori)
6     (effetti corsi in alternativa)
7 }
```

Parallelismo tra corsi

In questa situazione, vi sono più corsi di cui è necessario verificare le precondizioni (Figura 7.5). L'ordine con cui questo avviene non è importante, tra una verifica e l'altra, infatti, lo stato non cambia, in quanto non viene applicato alcun effetto. Quindi, ogni verifica viene fatta sullo stesso insieme di competenze.

```
1 proctype milestone_i() {
2     preconditions_course_1();
```

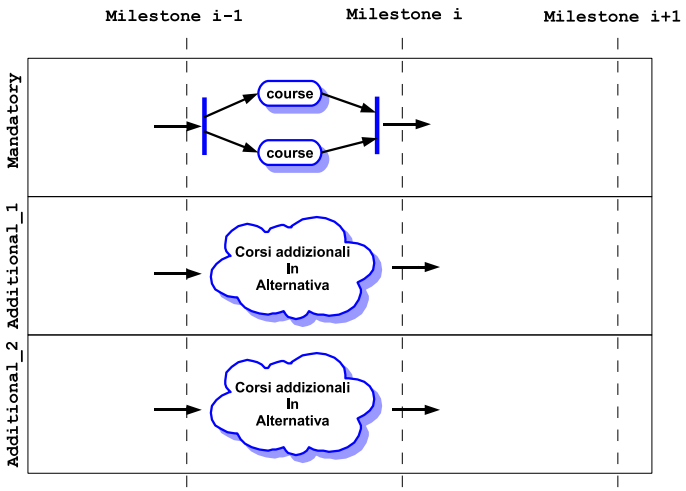


Figura 7.5: Rappresentazione di corsi in parallelo nella swimlane mandatory

```
3       preconditions_course_2();
4       ...
5       preconditions_course_n();
6
7       (precondizioni corsi in alternativa)
8       (effetti corsi obbligatori)
9       (effetti corsi in alternativa)
10  }
```

Punti di decisione

Questo è il caso più complesso da trattare. Ad ogni punto di decisione, infatti, può essere associata una condizione riferita al livello di una qualche conoscenza in possesso dello studente. In base al valore di ogni condizione la strada da intraprendere cambia. Tuttavia, bisogna considerare che i vari punti di decisione possono essere annidati a formare strutture più o meno complesse. Inoltre, il valore di una certa condizione in un determinato punto della verifica può avere effetti anche su milestone successive.

Per tener traccia del ramo da seguire, via via che si incontrano punti di decisione e che vengono verificate le condizioni, si è scelto di utilizzare un array di interi, in cui ad ogni posizione corrisponde un punto di deci-

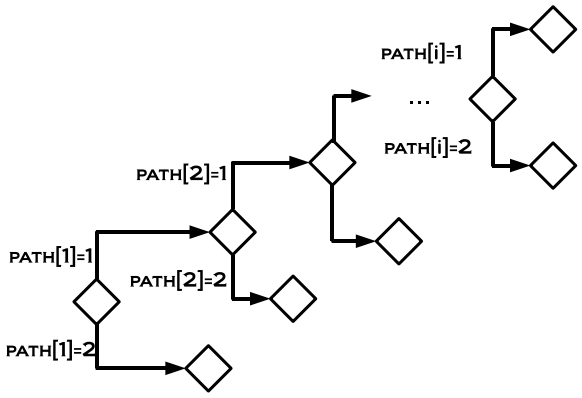


Figura 7.6: Utilizzo della variabile path nel caso di punti di decisione annidati.

sione. In base al valore della condizione verrà settato il numero del ramo che si deve seguire. In Figura 7.6 è riportato un esempio in cui le condizioni partizionano il cammino in due strade alternative. In generale, però, le possibili diramazioni potrebbero essere anche di più. Questo significa semplicemente, che la numerazione dei rami prosegue oltre il valore due. Pertanto, sarà sufficiente associare alla i-esima posizione dell'array, il valore del ramo corrispondente.

```
1  proctype milestone_i() {
2      if
3      ::(cond_1) -> preconditions_course_1();
4                    ...
5                    preconditions_course_n();
6                    path_mandatory[1] = 1;
7      ::else ->
8              path_mandatory[1] = 2;
9              if
10             ::(cond_2) -> preconditions_course_2();
11                           ...
12                           preconditions_course_m();
13                           path_mandatory[2] = 1;
14             ::else -> preconditions_course_3();
15                       ...
16                       preconditions_course_k();
```

```
17                          path_mandatory[2] = 2;
18                  fi;
19          fi;
20
21          (precondizioni corsi in alternativa)
22          (effetti corsi obbligatori)
23          (effetti corsi in alternativa)
24  }
```

La rappresentazione grafica corrispondente a questo esempio, è riportata in Figura 7.7.

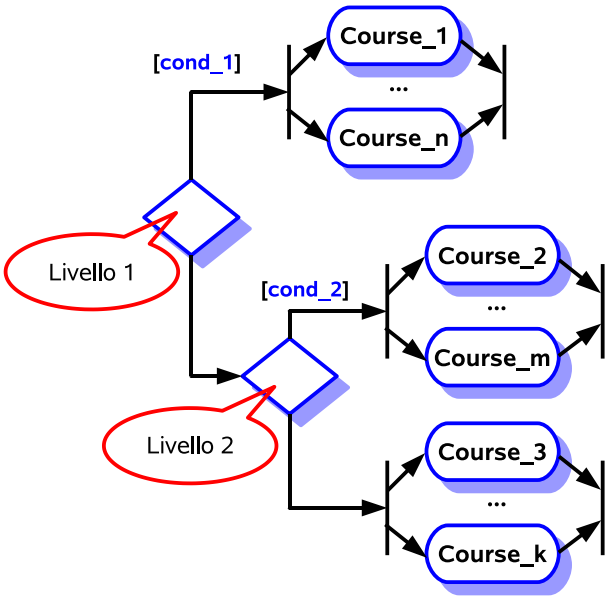


Figura 7.7: Rappresentazione di una porzione di piano in cui compaiono più punti di decisione, a diversi livelli di annidamento

È importante notare che questo ragionamento vale anche nel caso in cui la decisione si prolunghi su più milestones. In generale, supponendo di trovarsi in una milestone in cui vi siano rami originati da punti di decisione precedenti, è sufficiente controllare il livello di annidamento a cui ci si trova, sia questo i. A partire dalla posizione uno dell'array (la posizione zero indica che ci si trova nella swimlane dei corsi obbligatori) si ripercorrono le varie decisioni prese fino alla i-esima. In questo modo si determina quale sia il

corso di cui bisogna verificare le precondizioni nella milestone in cui ci si trova.

L'array *path_mandatory* viene utilizzato per realizzare questo meccanismo. Trattandosi di un array occorre definirne la dimensione. Per questo, prima di iniziare la traduzione occorre controllare il livello massimo di annidamento raggiunto nel diagramma.

Terminata la verifica delle precondizioni per i corsi obbligatori, occorre passare alle swimlanes dei corsi *additional*. In questo caso, la traduzione per la prima milestone differisce leggermente dalle altre. Arrivati a questo punto della verifica, infatti, non è ancora stato scelto alcun percorso tra quelli addizionali. In SPIN esiste la possibilità di creare una scelta non deterministica mediante un *if* con più condizioni verificate. Quello che occorre fare, quindi, è definire tante condizioni a *true*, quante sono le swimlanes addizionali. In questo modo, nella fase di simulazione SPIN ne sceglierà una in modo non deterministico. In fase di verifica, invece, questa scelta permette di verificare tutti i piani addizionali.

```
1  proctype milestone_1() {
2      (precondizioni corsi obbligatori)
3
4      if
5      ::(true) ->(precond. corsi_add_1);
6                  path_additional[0]=1;
7      ::(true) ->(precond. corsi_add_2);
8                  path_additional[0]=2;
9      ...
10     ::(true) ->(precond. corsi_add_n);
11                 path_additional[0]=n;
12     fi;
13
14     (effetti corsi obbligatori)
15     (effetti corsi in alternativa)
16 }
```

Per le restanti milestone, invece, il valore *true* sarà sostituito da una condizione espressa sulla variabile che tiene traccia del cammino scelto.

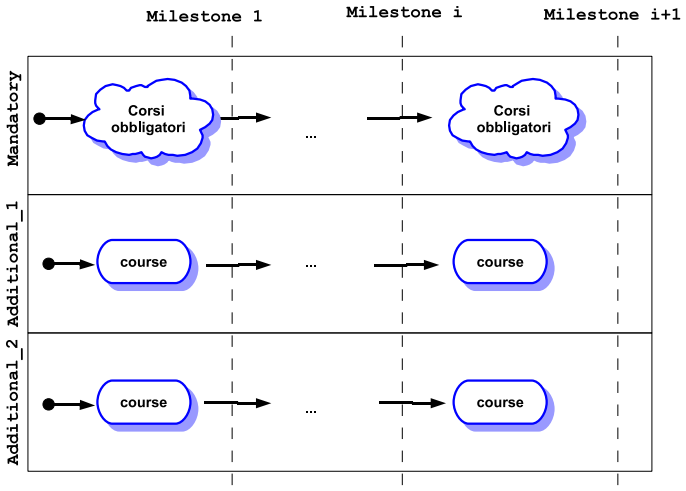


Figura 7.8: Rappresentazione di un corso nella swimlane additional

```
1  proctype milestone_i() {
2      (precondizioni corsi obbligatori)
3
4      if
5      ::(path_additional[0]==1) ->(precond. corsi_add_1);
6      ::(path_additional[0]==2) ->(precond. corsi_add_2);
7      ...
8      ::(path_additional[0]==n) ->(precond. corsi_add_n);
9      fi;
10
11     (effetti corsi obbligatori)
12     (effetti corsi in alternativa)
13 }
```

Per ogni cammino addizionale definito, occorre esprimere quali precondizioni debbano essere verificate. Il procedimento è simile a quello descritto per i corsi mandatory, in quanto i casi possibili sono gli stessi:

Corso singolo all'interno di una swimlane additional

Come nel caso precedente, è sufficiente verificare le precondizioni di tale corso, badando di essere all'interno della condizione che identifica il path corretto. Dopo di che, se ci si trova nella prima milestone, occorre memo-

rizzare quale ramo addizionale è stato scelto. Per questo, in modo simile a quanto avveniva per i corsi mandatory, nel caso di punti di decisione, si utilizza un array di interi: *path_additional*. La prima posizione è riservata alla memorizzazione del cammino addizionale scelto, mentre quelle successive verranno utilizzate per gestire i punti di decisione. La Figura 7.8 rappresenta due piani in alternativa tra loro, in cui compaiono corsi singoli all'interno delle milestones.

Corsi in parallelo all'interno di una swimlane additional

La situazione in cui su un ramo additional compaiano corsi da seguire in parallelo, come rappresentato in Figura 7.9, è sufficiente controllare in sequenza (senza vincoli di ordine) le precondizioni dei corsi in questione. Nel caso ci si trovi nella prima milestone, al termine della verifica delle precondizioni, occorre memorizzare nella prima posizione dell'array *path_additional[0]*, il numero identificativo del cammino scelto.

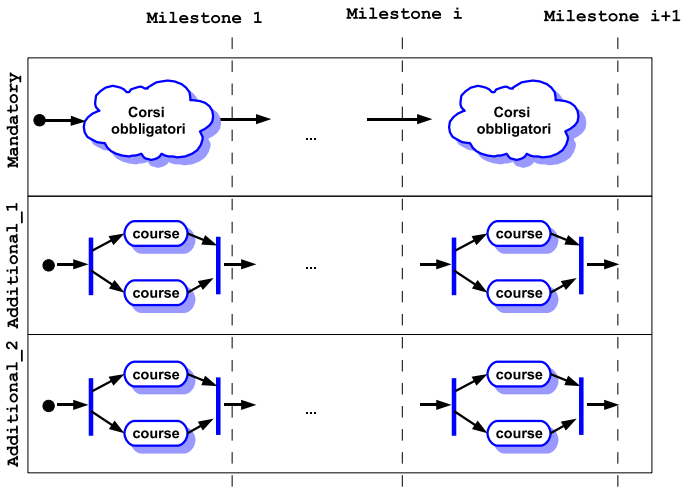


Figura 7.9: Rappresentazione di corsi in parallelo nella swimlane additional

Punti di decisione all'interno di una swimlane additional

Questo caso è molto simile a quello descritto per i corsi mandatory. Occorre notare, però, che si dispone già di una variabile per tener traccia del

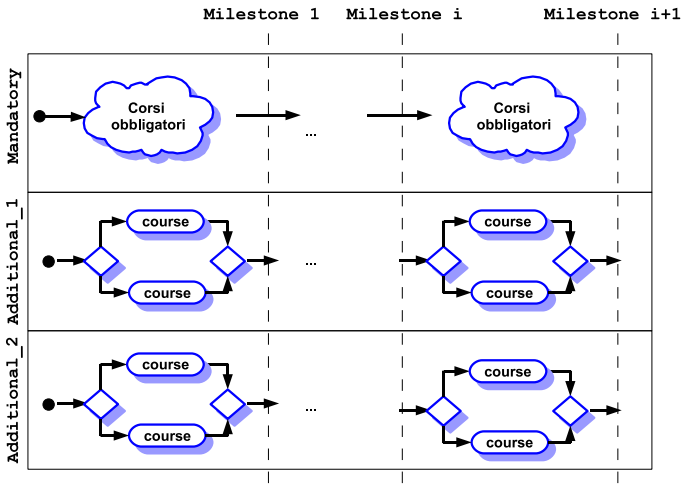


Figura 7.10: Rappresentazione di corsi in parallelo nella swimlane additional

valore delle condizioni che possono susseguirsi in diversi livelli di annidamento. L'array *path_additional*, infatti, memorizza nella posizione "zero" il valore del cammino scelto. I restanti indici vengono utilizzati, esattamente come nel caso dei corsi mandatory, per tener traccia dei rami da seguire in seguito alla verifica delle diverse condizioni. Lo schema di tale situazione è riportato in Figura 7.10.

7.4.2 Applicazione degli effetti

Terminata la fase di verifica delle precondizioni, è possibile procedere con l'aggiornamento dello stato in conseguenza agli effetti dei vari corsi. In generale, verranno aggiunti solo gli effetti dei corsi che terminano in corrispondenza della milestone che si sta considerando.

In particolare, per ogni corso della swimlane "mandatory" che termina all'interno della milestone in questione, viene invocata la corrispondente procedura per l'applicazione degli effetti. Questo ad eccezione dei corsi che si trovano su rami originati da punti di decisione. In tal caso, infatti, occorre utilizzare la variabile *path_mandatory* e ripercorrere le decisioni via via intraprese, fino ad arrivare alla milestone in questione. A questo punto, si può procedere verificando quale ramo debba essere considerato e, nel caso in cui

vi siano corsi che terminano entro la milestone, se ne potranno aggiungere gli effetti.

Per quanto riguarda i cammini additional, invece, è già stata effettuata una scelta, memorizzata nella prima posizione dell'array *path_additional*. Prima di tutto, quindi, occorre controllare quale sia il cammino aggiuntivo da considerare. Questo viene fatto mediante un *if* con una condizione per ogni cammino a scelta. Dopo di che, ogni caso viene trattato separatamente, aggiungendo gli effetti dei corsi che terminano entro la milestone e controllando il ramo in cui ci si trova in caso di punti di decisione. Questo processo è molto simile a quello descritto per la verifica delle precondizioni. Esso differisce poiché, anziché considerare solo i corsi che iniziano nella milestone, vengono considerati solo quelli che terminano nel periodo.

Di seguito viene riportata la struttura generale per la definizione delle prime due milestone. Esse differiscono nel modo in cui viene trattata la verifica delle precondizioni nei cammini addizionali. Nella prima, infatti, le condizioni sono tutte a *true* e, dopo aver effettuato la verifica, viene settato il cammino scelto nella prima posizione della variabile *path_additional*. Nella *milestone_2*, invece, la scelta del percorso è già stata fatta, pertanto sarà sufficiente controllarne il valore.

```
1  proctype milestone_1() {
2      (precondizioni corsi obbligatori)
3      if
4      ::(true) -> (precond. corsi_add_1);
5                  path_additional[0]=1;
6      ::(true) -> (precond. corsi_add_2);
7                  path_additional[0]=2;
8      ...
9      ::(true) -> (precond. corsi_add_n);
10                 path_additional[0]=n;
11     fi;
12     (effetti corsi obbligatori)
13     if
14     ::(path_additional[0]==1) -> (effetti corsi_add_1);
15     ::(path_additional[0]==2) -> (effetti corsi_add_2);
16     ...
```

```
17      ::(path_additional[0]==n) -> (effetti corsi_add_n);
18      fi;
19  }
20  proctype milestone_2() {
21      (precondizioni corsi obbligatori)
22      if
23      ::(path_additional[0]==1) -> (precond. corsi_add_1);
24      ::(path_additional[0]==2) -> (precond. corsi_add_2);
25      ...
26      ::(path_additional[0]==n) -> (precond. corsi_add_n);
27      fi;
28      (effetti corsi obbligatori)
29      if
30      ::(path_additional[0]==1) -> (effetti corsi_add_1);
31      ::(path_additional[0]==2) -> (effetti corsi_add_2);
32      ...
33      ::(path_additional[0]==n) -> (effetti corsi_add_n);
34      fi;
35  }
```

7.5 Verifica dei piani di studio

L'esecuzione di un programma PROMELA così come è stato definito fino ad ora, non produce alcun effetto. Inizialmente, infatti, viene eseguito un solo processo, quello il cui nome è *init*. Questo deve essere sempre definito all'interno di un programma. In particolare, nel conteso dei curricula di studi, la prima cosa da fare è definire lo stato iniziale dello studente. Dopo di che si potrà procedere con l'esecuzione del curriculum.

Quindi, il processo *init* risulta essere:

```
1  init {
2      atomic{SetInitialSituation();}
3      run CurriculumVerification();
4  }
```

La parola chiave *run* permette di creare il processo che si occuperà della verifica dei corsi.

SPIN mette a disposizione diversi strumenti per la verifica di un programma. Ad esempio, è possibile simulare l'esecuzione di uno tra i possibili piani alternativi. In questa modalità, verrà selezionato in modo casuale un possibile processo di esecuzione. Operando in modalità verifica, invece, verranno verificati tutti i possibili curriculum originati dal programma. Infine, è possibile verificare il soddisfacimento di alcune proprietà espresse in logica LTL.

Questi strumenti possono essere utilizzati per verificare ognuna delle tre categorie in cui sono stati catalogati i vincoli:

1. Assenza di competency gaps;
2. Raggiungimento del learning goal;
3. Conformità del piano rispetto al curricula model;

7.5.1 Verifica dell'assenza di competency gaps

L'occorrenza di un competency gap si verifica nel momento in cui una o più precondizioni di un corso non sono rispettate. La Figura 7.11 schematizza questa situazione: la precondizione *f6* del corso *course3* non fa parte delle conoscenze dello studente e non rientra tra gli effetti di alcun corso precedente nella sequenza. Un problema di questo tipo può essere causato da un errore nella definizione dei corsi, oppure dal modo in cui questi vengono disposti all'interno del piano.

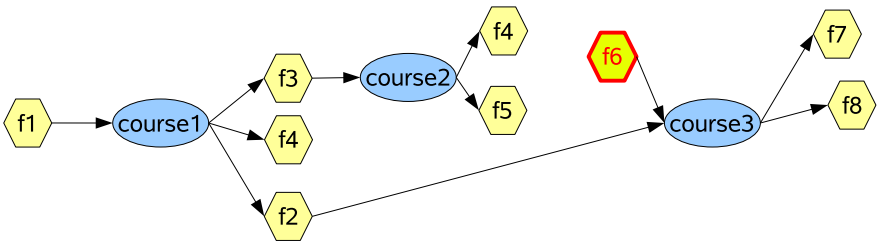


Figura 7.11: Esempio di piano che presenta un competency gap

In presenza di un tale errore, SPIN termina la verifica evidenziando che il piano non è valido. A questo punto, l'utente potrà richiedere la simulazione

passo passo dell'esecuzione che porta all'errore, per potersi, così, rendere conto di quale sia il problema. Verranno, quindi, visualizzati i vari concetti appresi, man mano che vengono aggiunti gli effetti dei vari corsi, fino ad arrivare alla situazione di errore.

In Figura 7.12 sono riportate le due finestre che segnalano l'errore: una permette di effettuare la *simulazione guidata*, cioè la simulazione del controesempio che falsifica l'asserzione, l'altra consente all'utente di capire quale sia il problema. Ad esempio, in figura, è riportato il seguente messaggio di errore:

```
pan:assertion violated
(event_driven_programming) >= 2) && (concurrent_programming >=
2) && (file_systems >= 1) && (standard_complexity_classes >= 2)
```

Questo significa che almeno una delle quattro condizioni che compaiono nell'asserzione non è verificata.

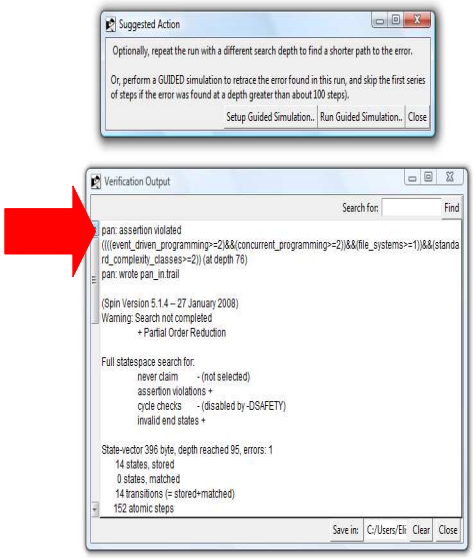


Figura 7.12: Segnalazione di errore in XSPIN: l'utente viene informato della violazione di una assert e può visualizzare l'esecuzione che porta all'errore.

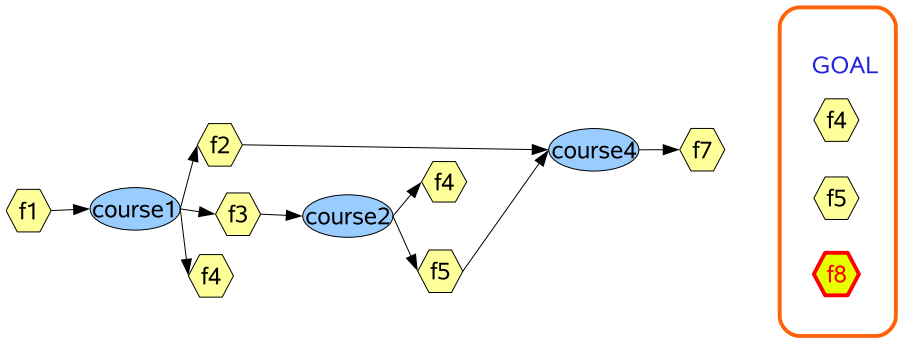


Figura 7.13: Esempio di piano che non soddisfa il learning goal dell'utente

7.5.2 Raggiungimento del learning goal

Come è già stato precisato più volte, un curricula di studi che non porta l'utente ad acquisire le conoscenze da lui richieste, è di scarsa utilità. Il learning goal, dunque, consiste in un insieme di condizioni che devono essere verificate nell'ultimo stato, il quale rappresenta l'insieme delle conoscenze in possesso dello studente al termine del piano di studi. Pertanto, come si può vedere dalla definizione del processo *CurriculumVerification*, riportato a pagina 123, la verifica del learning goal è l'ultima azione ad essere compiuta. Essa avviene dopo il controllo delle varie milestone.

Per controllare che ogni competenza sia stata acquisita al livello desiderato, il meccanismo è simile a quanto avviene per la verifica delle precondizioni e consiste nell'asserire una condizione data dalla congiunzione di tutte le richieste contenute nel learning goal.

```
1 inline LearningGoal() {
2     assert(
3           competenza_1 >= livello_1 &&
4           ... &&
5           competenza_n >= livello_n
6     );
7 }
```

Poiché non è possibile asserire una condizione falsa, nel caso una o più disuguaglianze non siano verificate, l'esecuzione del programma termina con un fallimento. Trattandosi della violazione di una *assert*, proprio come nel caso

di competency gaps, il comportamento del sistema è lo stesso: viene segnalato il fallimento all'utente, il quale può decidere di passare alla simulazione di tale esecuzione, per capire quali conoscenze non siano state acquisite.

In Figura 7.13 è riportato lo schema che rappresenta la violazione di un competency gap: la competenza *f8*, infatti, non è tra gli effetti di alcun corso del piano.

7.5.3 Conformità del piano rispetto al curricula model

Le relazioni tra le competenze e i vincoli temporali sulla loro acquisizione, vengono espressi nel linguaggio grafico DCML. Successivamente, da questo viene generato un insieme di formule in logica LTL che dovranno essere verificate sul piano di studi presentato.

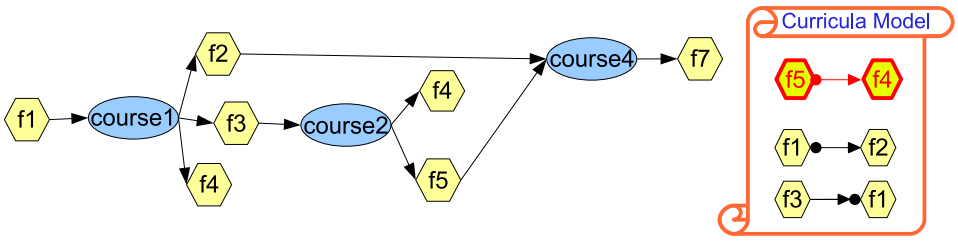


Figura 7.14: Esempio di piano che non soddisfa il curricula model

XSPIN permette di verificare formule LTL mediante un'apposita finestra (Figura 7.15), mediante la quale si possono esprimere tutte le relazioni contemplate dal linguaggio DCML. In particolare, la proprietà da verificare viene espressa mediante variabili legate da connettivi temporali e logici. Successivamente è possibile generare automaticamente la *never claim*, cioè la negazione della formula da verificare. La verifica, infatti, avviene controllando che non vi siano esecuzioni che soddisfano la negazione della proprietà.

Come si può vedere dalla Figura 7.15, la scritta *Verification Result: not valid* informa l'utente che la formula non è verificata. Anche in questo caso è possibile effettuare la simulazione guidata, controllando, così, il livello e la successione con cui vengono acquisite le varie competenze.

Eseguendo la simulazione guidata, è possibile controllare la sequenza con la quale vengono acquisite le varie competenze. In particolare, come si può

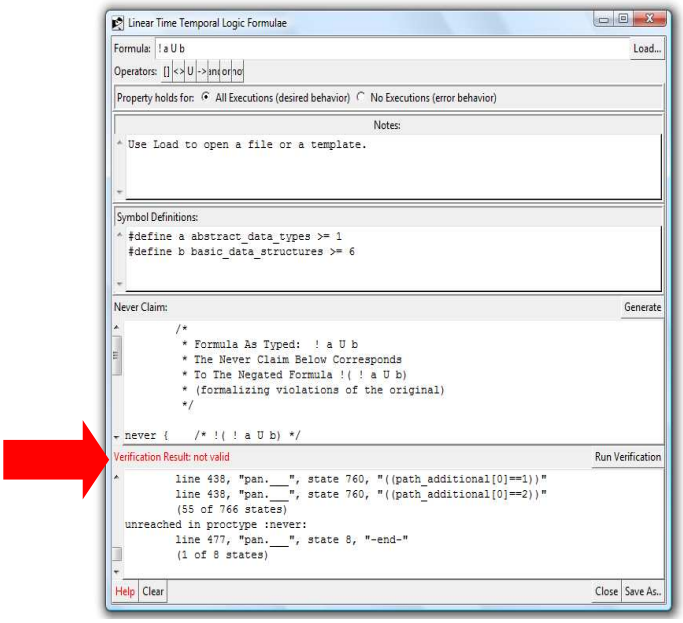


Figura 7.15: Verifica di formule LTL mediante l'interfaccia grafica XSPIN.

vedere in Figura 7.16, oltre alle varie condizioni che vengono via via verificate, viene riportata la sequenza delle operazioni eseguite. Sostanzialmente si tratta dell'ordine con cui vengono verificate le precondizioni e aggiunti gli effetti dei corsi. Nell'esempio, l'ultima azione ad essere eseguita è *effects_course_computer_science_III*. Da qui si può dedurre che l'aggiunta di qualche effetto causa la violazione del vincolo.

7.6 Caso di studio

Dalla descrizione fornita nella Sezione 7.4, si può intuire che la traduzione di un activity diagram, rappresentante un curricula di studi, in codice PROMELA non è un procedimento del tutto banale. Pertanto, in questa Sezione verrà presentato un esempio, allo scopo di chiarire meglio questa fase della verifica.

Il diagramma in esame è riportato in Figura 7.17. Esso si articola in quattro milestones e presenta due piani alternativi tra cui l'utente può scegliere.

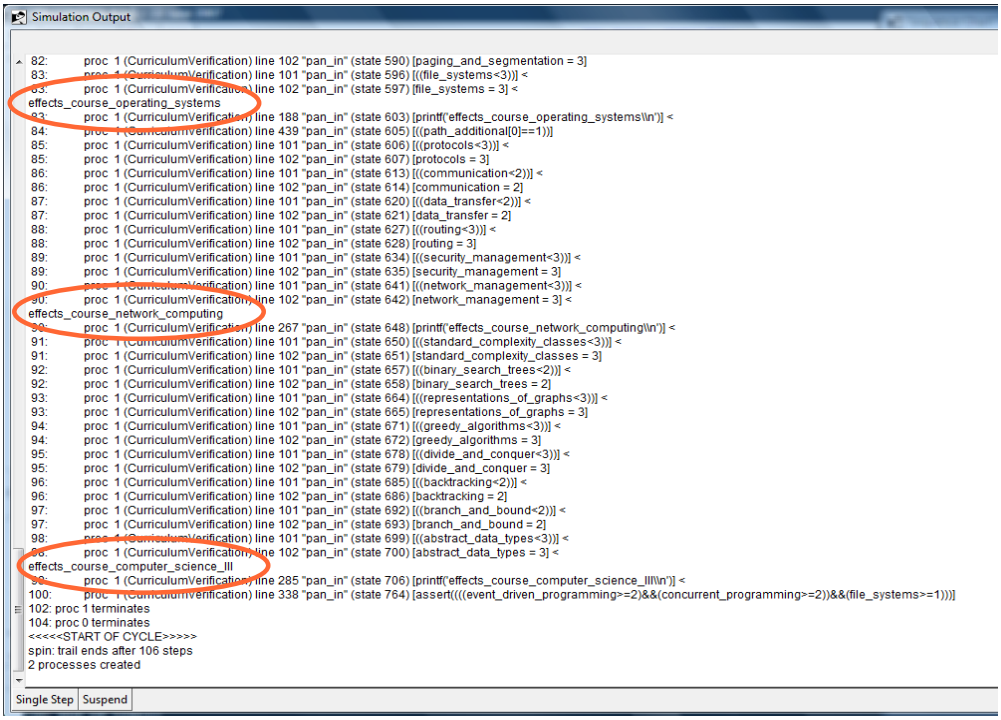


Figura 7.16: Controesempio generato da SPIN dopo il fallimento della verifica di una formula LTL

Per ogni milestone, verranno considerate prima le precondizioni dei corsi che in essa hanno inizio e poi gli effetti di quelli che terminano.

Pertanto, la traduzione del primo periodo risulta essere:

```
1  inline milestone_1(){
2    atomic{
3      preconditions_course_computer_science_I();
4      preconditions_course_logics();
5      if
6      :: true ->
7        if
8        ::(C_programming < 1)-> path_additional[1]=1;
9            preconditions_course_principle_of_programming();
10       ::else -> path_additional[1]=2;
11       fi;
12       path_additional[0] = 1;
```

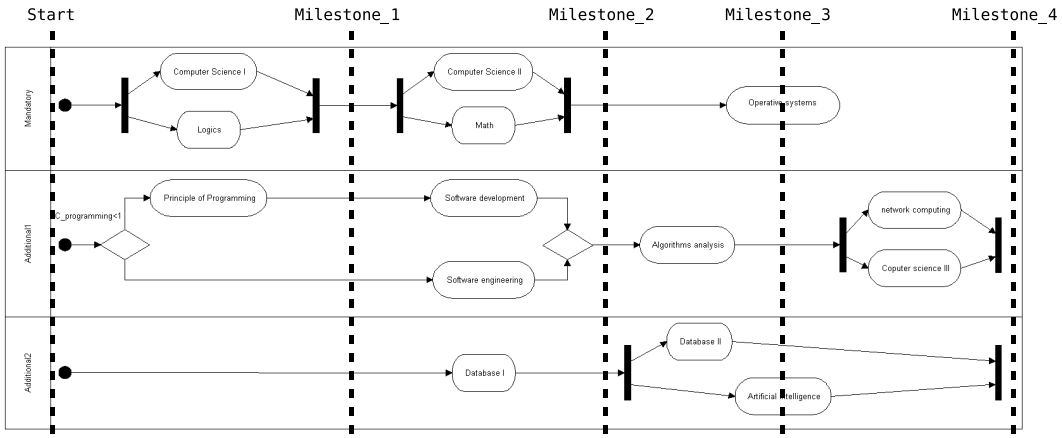


Figura 7.17: Esempio di curricula di studi

```
13      :: true -> path_additional[0] = 2;
14      fi;
15
16      effects_course_computer_science_I();
17      effects_course_logics();
18
19      if
20      :: (path_additional[0]==1) ->
21        if
22        ::(path_additional[1]==1) ->
              effects_course_principle_of_programming();
23        ::(path_additional[1]==2) -> skip;
24        fi;
25      :: (path_additional[0]==2) -> skip;
26      fi;
27    }
28  }
```

Prima di tutto, quindi, vengono verificate le precondizioni dei corsi *computer science I* e *logics*, che fanno parte della swimlane mandatory. Dopo di che, un primo *if*, le cui condizioni sono tutte a *true*, permette di distinguere tra le due swimlane *additional*. La prima si articola a sua volta in due strade precedute da una condizione che, se verificata, comporta il con-

trollo delle precondizioni per il corso *principle of programming*. In questo caso, occorre memorizzare la strada intrapresa al primo livello di annidamento. Questo avviene grazie all'istruzione *path_additional[1]=1*. Nel caso in cui la condizione non sia rispettata, invece, il valore memorizzato cambia: *path_additional[1]=2*. Tuttavia, in entrambi i casi, nella prima posizione dell'array, cioè *path_additional[0]*, viene memorizzato il valore uno. In questo modo si rappresenta il fatto che, tra le due alternative, sia stata scelta la prima.

Se il cammino addizionale scelto è il secondo, l'unica istruzione è *path_additional[0]=2*. In questo caso, infatti, l'utente non dovrebbe seguire altri corsi, oltre a quelli obbligatori, durante il primo periodo.

Per quanto riguarda l'applicazione degli effetti, il ragionamento da seguire è analogo. Dopo aver trattato i due corsi obbligatori (la cui durata è di un solo periodo), una condizione stabilisce quale piano additional sia stato scelto nella fase di verifica delle precondizioni. Sulla base di tale valore, memorizzato in *path_additional[0]*, vengono trattati i vari curricula in modo indipendente tra loro.

Nella seconda milestone, a differenza di quanto avveniva nella prima, le decisioni sui cammini addizionali sono già state prese e le condizioni sono già state verificate. Pertanto, per capire quale ramo considerare sarà sufficiente procedere per casi, sui valori assunti nelle diverse posizioni dalla variabile *path_additional*.

```
1  inline milestone_2(){
2    atomic{
3      preconditions_course_computer_science_II();
4      preconditions_course_math();
5
6      if
7      :: (path_additional[0]==1) ->
8        if
9        ::(path_additional[1]==1) ->
             preconditions_course_software_development();
10       ::(path_additional[1]==2) ->
             preconditions_course_software_engineering();
11       fi
```

```
12      :: (path_additional[0]==2) ->
            preconditions_course_database_I();
13      fi;
14
15      effects_course_computer_science_II();
16      effects_course_math();
17
18      if
19      :: (path_additional[0]==1) ->
20        if
21        ::( path_additional[1]==1) ->
              effects_course_software_development();
22        ::( path_additional[1]==2) ->
              effects_course_software_engineering();
23        fi;
24      :: (path_additional[0]==2) ->
25         effects_course_database_I();
26      fi;
27    }
28  }
```

Innanzi tutto, vengono verificate le precondizioni dei due corsi obbligatori: *computer science II* e *math*. Dopo di che, se l'alternativa scelta nella milestone precedente è la prima, occorre controllare quale ramo si debba considerare, sulla base del valore della condizione *(C_programming < 1)* verificata nello stato precedente. Per questo, quindi, si interroga la variabile *path_additional* prima sul valore contenuto in prima posizione (cioè quello che indica l'alternativa scelta) e poi, nel caso sia uno, sul valore contenuto nella seconda posizione (quello che indica quale ramo prendere in corrispondenza del primo livello di annidamento). A seconda del valore di quest'ultimo verranno verificate le precondizioni del corso *software development*, oppure di *software engineering*. L'alternativa due, invece, consiste di un solo corso. Pertanto, sarà sufficiente verificarne le precondizioni.

L'applicazione degli effetti, avviene in modo del tutto analogo a quanto descritto per la verifica delle precondizioni.

Diversa è la situazione che coinvolge la terza e la quarta milestone. In esse, infatti, compaiono due corsi della durata di due periodi. Il primo è

oprating sistem e rientra tra quelli obbligatori. Pertanto le precondizioni verranno controllate nella terza milestone, ma gli effetti saranno aggiunti solo in quella successiva. Simile è il caso che coinvolge il corso *artificial intelligence*. Questo, però, fa parte della seconda delle due alternative e compare in un parallelo con un altro corso. La netta divisione tra verifica delle precondizioni e applicazione degli effetti di ogni corso, non crea alcun problema alla traduzione di questa situazione. Semplicemente, le precondizioni del corso *artificial intelligence* saranno verificate, insieme a quelle di *database II*, nel terzo periodo e i suoi effetti saranno aggiunti solo in quello successivo.

```
1  inline milestone_3(){
2    atomic{
3      preconditions_course_operating_systems
4
5      if
6      ::(path_additional[0]==1) ->
7          preconditions_course_algorithm_analysis();
8      ::(path_additional[0]==2) ->
9          preconditions_course_database_II();
10         preconditions_course_artificial_intelligence();
11     fi;
12
13     if
14     ::(path_additional[0]==1) ->
15         effects_course_algorithm_analysis();
16     ::(path_additional[0]==2) ->
17         effects_course_database_II();
18     fi;
19   }
20 }
21
22 inline milestone_4(){
23   atomic{
24     if
25     ::(path_additional[0]==1) ->
26         preconditions_course_network_computing();
27         preconditions_course_computer_science_III();
28     ::(path_additional[0]==2) -> skip;
```

```
29      fi;
30
31      effects_course_operating_systems();
32
33      if
34      ::(path_additional[0]==1) ->
35          effects_course_network_computing();
36          effects_course_computer_science_III();
37      ::(path_additional[0]==2) ->
38          effects_course_artificial_intelligence();
39      fi;
40    }
41 }
```

7.7 Importanza della traduzione automatica di curricula di studi e di diagrammi DCML

Le rappresentazioni grafiche di curricula di studi, mediante activity diagram UML, e di curricula models, per mezzo del linguaggio DCML, costituiscono uno strumento di facile utilizzo per l'utente. Questi potrebbe, ad esempio, essere uno studente che sta per iscriversi ad un corso di laurea, oppure un docente che si occupa della verifica di piani di studio da proporre a vari utenti.

Tuttavia, non bisogna dimenticare che lo scopo finale è proprio quello della verifica.

Per poter conciliare le due fasi, però, è necessaria una traduzione del curricula, in linguaggio PROMELA, e dei vincoli tra le competenze, in logica temporale LTL. I vari esempi e programmi presentati nelle sezioni e nei capitoli precedenti, sono stati tradotti manualmente. Questo, potrebbe rappresentare un limite se si pensa che la verifica di piani di studio potrebbe essere utilizzata in contesti più ampi e complessi. È necessario, quindi, introdurre un meccanismo automatico di traduzione.

Per quanto riguarda i diagrammi UML, alcuni strumenti permettono di estrarre dalla rappresentazione grafica, una descrizione testuale in formato *XML based*. A partire da questa sarebbe sicuramente utile e interes-

sante sviluppare un meccanismo automatico per la produzione di programmi PROMELA, della forma descritta precedentemente.

Anche per la traduzione dei diagrammi DCML si potrebbe pensare ad un linguaggio intermedio, da cui estrarre le formule LTL. In questo modo, la rappresentazione sarebbe completamente indipendente dalla traduzione finale. Per cui, ogni aggiunta o modifica grafica riguarderebbe il solo linguaggio intermedio.

Al momento questa resta solo una proposta, in quanto non è stato implementato alcun meccanismo automatico di traduzione.

Capitolo 8

Considerazioni finali

Nei capitoli iniziali è stata sottolineata l'importanza del *riuso* e della *personalizzazione* di risorse nel Semantic Web. Utilizzando questi aspetti come linee guida, in questa tesi è stato presentato un approccio per la verifica di curricula di studi. In particolare, "personalizzazione" significa dare la possibilità all'utente di esprimere ciò che vuole ottenere. Un curricula di studi deve permettere di raggiungere questi obiettivi (*learning goal*).

Tuttavia, questo non rappresenta l'unico requisito di un corso di studi. Occorre verificare che lo studente sia sempre nella condizione di poter comprendere i contenuti dei vari corsi. Tale vincolo viene verificato controllando che sia in possesso di tutte le conoscenze che fanno parte dei prerequisiti di un corso, nel momento in cui si trova a doverlo affrontare.

Infine, per far sì che ad un certo percorso di studi venga riconosciuta una qualche certificazione, occorre che questa rispetti alcuni vincoli. Il curricula model definisce l'insieme di condizioni poste, ad esempio, dall'ente che eroga i corsi e che garantisce il riconoscimento di un qualche titolo al termine del percorso. Per facilitarne la definizione, è stato proposto DCML, un linguaggio grafico per la rappresentazione di vincoli e relazioni tra le competenze, successivamente tradotti in logica temporale LTL. In [42] viene descritta la realizzazione di un plugin grafico per la definizione di diagrammi DCML.

Il processo di verifica avviene mediante il model checker SPIN. Tale strumento è stato utilizzato per verifiche simili in altri contesti. Ad esempio, in [45], esso viene utilizzato per la verifica di "linee guida" nel settore del-

la medicina (clinical guidelines). Queste vengono rappresentate per mezzo del linguaggio grafico GLARE (GuideLine Acquisition, Representation and Execution) e successivamente tradotte in un programma PROMELA. In questo modo è possibile verificare su di esso un insieme di proprietà. La rappresentazione grafica di piani di studio, si ispira proprio a questo lavoro. Inizialmente, è stata presa in considerazione l'idea di utilizzare il lingaggio GLARE anche per la definizione di curricula di studio. Tuttavia, la maggiore flessibilità e potere espressivo degli activity diagram ha fatto propendere per quest'utlimi. Inoltre, DCML è stato definito con lo scopo di rappresentare l'insieme delle proprietà che un piano deve rispettare. Nel lavoro descritto in [45], invece, non vi sono meccanismi analoghi per la definizione dei vincoli.

Il meccanismo di verifica di curricula di studio verrà inserito come servizio all'interno del *Personal Reader Framework*, un sistema realizzato presso l'Università di Hannover, con la quale esiste una collaborazione con il gruppo *Personalized Information Systems group* della rete *Reasoning on the Web with Rules and Semantics European Network of Excellence (REWERSE).*

8.1 Il Personal Reader Framework

Il Perosnal Reader Framework[1] è uno strumento per la cerazione di applicazioni nel Semantic Web. L'utente può selezionare e combinare tra loro diversi servizi di cui vuole usufruire.

In Figura 8.1 sono riportate le principali componenti del sistema. Una applicazione consiste principalmente di tre tipi di servizi: il *Personalization Service (PService)*, il *Syndication Services (SynService)* ed il *Connector*. Il primo si occupa di realizzare alcune funzionalità che vengono messe a disposizione dell'utente. Il *SynService*, invece, rappresenta l'interfaccia grafica verso l'utente e permette di agevolare l'interoperabilità tra i vari servizi, fungendo da portale per le altre applicazioni. Tutto questo viene realizzato per mezzo di Servlets e pagine JSP. Infine, il Connector è responsabile del controllo della comunicazione tra interfaccia utente e PService.

Attualmente il sistema è in grado di ospitare due PService: il pianificatore

[1] http://www.personal-reader.de/

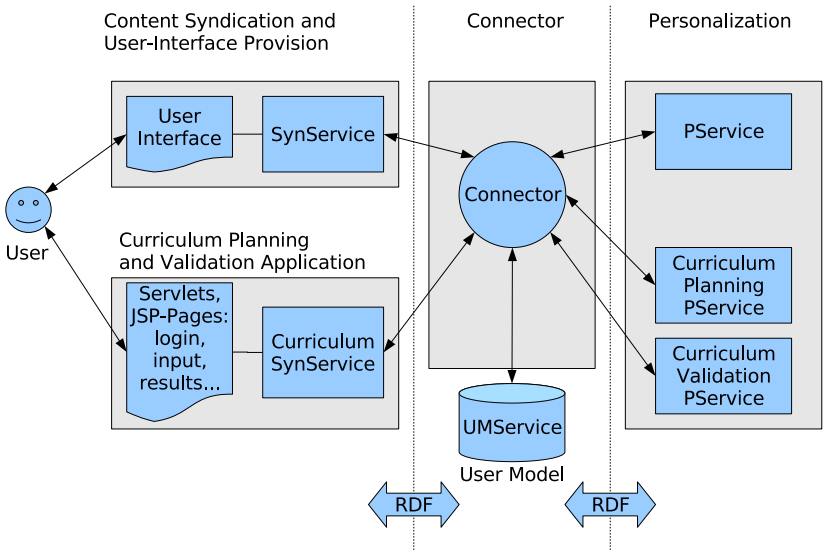


Figura 8.1: Architettura del Personal Reader Framework

[4, 36] (Curriculum Planning PService) e il sistema per la validazione di curricula di studio (Curriculum Validation PService).

Il compito del SynService è estremamente variabile e dipendente dal servizio richiesto. Esso si occupa di presentare l'interfaccia adeguata all'utente. Considerando il caso della validazione (funzionalità non ancora del tutto implementata), esistono diverse alternative per la sottomissione di un piano. In particolare, è possibile combinare il *Curriculum Planning PService* ed il *Curriculum Validation PService* facendo in modo che il piano prodotto dal primo, venga immediatamente sottoposto per la validazione. Un apposito pulsante "validate", permette di passare direttamente alla fase di verifica.

Un altro scenario possibile, è quello in cui l'utente decida di disegnare un proprio curriculum. Per la verifica di piani lineari potrebbe essere sufficiente interagire con l'utente mediante una pagina per l'inserimento della sequenza che si intende validare. Tuttavia, è possibile immaginare che utenti più esperti (quali, ad esempio, professori o enti che si occupano dell'erogazione di corsi) vogliano usufruire di maggiore potenzialità. Pertanto occorre prevedere la sottomissione di un piano rappresentato con un activity diagram UML.

La Figura 8.2 riassume i passaggi necessari per la validazione di un curricula, rispetto ad un curricula model.

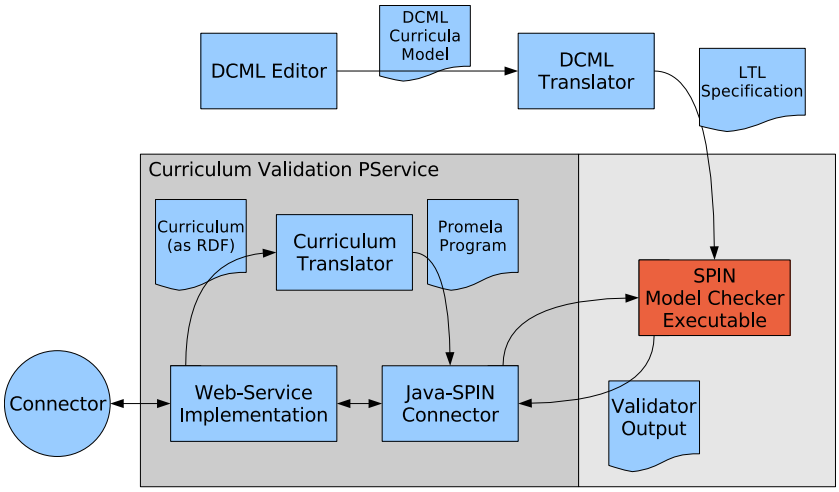


Figura 8.2: Processo per la validazione di un curricula rispetto ad un curricula model

8.2 Upper ontologies per la definizione di un "vocabolario" comune

Per poter realizzare i meccanismi di verifica presentati in questa tesi, occorre che i corsi vengano descritti in termini di precondizioni ed effetti. Allo stesso modo, l'utente deve poter esprimere i propri obiettivi. È necessario, quindi, che la comunicazione tra servizi (sia nel caso della validazione, sia in quello della pianificazione) ed utenti (studenti, professori, etc.) avvenga utilizzando lo stesso *vocabolario*. La pianificazione descritta in [4], è stata realizzata disponendo di descrizioni di corsi ricavate automaticamente dalle pagine dei docenti, mediante il sitema *Lixto*. Altre Università potrebbero usare sistemi analoghi per la descrizione delle proprie risorse.

Le *upper ontologies*[37] possono essere utilizzate come *lingua franca* per mappare in modo univoco competenze che rappresentano lo stesso concetto ma che vengono espresse in modi diversi. In questo modo, è possibile pensare all'integrazione di servizi erogati da enti differenti.

Ringraziamenti

Desidero ringraziare il prof. Matteo Baldoni che ha saputo trasformare una piccola curiosità in un grande entusiasmo. Grazie per la pazienza, i preziosi insegnamenti in ambito professionale e le incoraggianti parole nei momenti più difficili.

Un doveroso ringraziamento al prof. Alberto Martelli per la disponibilità dimostrata nell'accettare di essere controrlatore di questa tesi.

Grazie anche alla prof.ssa Cristina Baroglio, sempre pronta a regalare consigli e sorrisi, e alla prof.ssa Viviana Patti, con la quale è stato un piacere lavorare e condividere le varie esperienze all'estero.

Molto più di un pensiero va a Mamma, Papà e Lara, che mi sono sempre stati vicini e, soprattutto, che mi hanno sostenuta, incitata e... sopportata.

Grazie di tutto cuore a Chiara, Lorena e Valentina che sanno ascoltare e riescono a capirmi senza bisogno dire nulla. Grazie per non aver mai smesso di credere in me.

Non meno importanti sono tutte le persone che in questi anni mi hanno aiuta, gli Amici del gruppone "Quelli che alle nove un quarto da masu", i miei compagni e colleghi di avventura e tutte le persone che con me hanno sorriso, riso, scherzato, studiato e lavorato.

Grazie.

Bibliografia

[1] G. Antoniou and F. Van Harmelen. *A Semantic Web Primer*. MIT Press, 2004. pages 10

[2] M. Baldoni, C. Baroglio, G. Berio, and E. Marengo. Declarative representation of curricula models: an LTL- and UML-based approach. In *Proc. of WOA 2007: Dagli oggetti agli agenti, Agenti e Industrie: Applicazioni tecnologiche degli agenti software*, pages 34–41, Genova, Italy, September 2007. pages 106

[3] M. Baldoni, C. Baroglio, I. Brunkhorst, N. Henze, E. Marengo, and V. Patti. Constraint Modeling for Curriculum Planning and Validation. Sottoposto per la pubblicazione nell'*Interactive Learning Environments journal, Special Issue on "Semantic Technologies for Multimedia-anhanced Learning Environments"*, 2008. pages 57

[4] M. Baldoni, C. Baroglio, I. Brunkhorst, N. Henze, E. Marengo, and V. Patti. A Personalization Service for Curriculum Planning. In *14th Workshop on Adaptivity and User Modeling in Interactive Systems, ABIS 2006*, pages 17–20, 2006, October Hildesheim, Germany. pages 148, 149

[5] M. Baldoni, C. Baroglio, I. Brunkhorst, E. Marengo, and V. Patti. A Service-Oriented Approach for Curriculum Planning and Validation. In *Proceedings of the Multi-Agent Logics, Languages, and Organisations, Federated Workshops, MALLOW'007, Agent, Web Services and Ontologies, Integrated Methodologies, MALLOW-AWESOME'007*, Durham, GB, September 2007. pages 67

[6] M. Baldoni, C. Baroglio, and N. Henze. Personalization for the Semantic Web. In *Reasoning Web, First International REWERSE Summer School 2005*, pages 173–212, Verlag, Malta, July 2005. pages 13, 16

[7] M. Baldoni, C. Baroglio, N. Henze, and V. Patti. Setting up a Framework for Comparing Adaptive Educational Hypermedia: First Steps and Application on Curriculum Sequencing. In *ABIS-Workshop 2002: Personalization for the mobile World, Workshop on Adaptivity and User Modeling in Interative Software Systems*, pages 43–50, Hannover, Germany, October 2002. pages 17

[8] M Baldoni, C Baroglio, and E. Marengo. Curricula Modeling and Checking. In *Proc. of AI*IA 2007: Advances in Artificial Intelligence, 10th Congress of the Italian Association for Artificial Intelligence*, volume 4733 of LNAI, pages 471–482, Rome, Italy, September 2007. pages 57, 100

[9] M. Baldoni, C. Baroglio, and V. Patti. Web-Based Adaptive Tutoring: An Approach Based on Logic Agents and Reasoning about Actions. In *Artificial Intelligence Review*, pages 3–39, 2004. pages 7, 46, 47, 48, 52, 97, 99

[10] M. Baldoni, L. Giordano, A. Martelli, and V. Patti. Reasoning about Complex Actions with Incomplete Knowledge: A Modal Approach. In A. Restivo, S. Ronchi Della Rocca, and L. Roversi, editors, *Proc. of Theoretical Computer Science, 7th Italian Conference, ICTCS'2001*, volume LNCS 2202, pages 405–425, 2001. pages 48

[11] M. Baldoni, L. Giordano, A. Martelli, and V. Patti. Programming Rational Agents in a Modal Action Logic. *Annals of Mathematics and Artificial Intelligence, Special issue on Logic-Based Agent Implementation*, pages 207–257, 2004. pages 54

[12] M. Baldoni and E. Marengo. Curriculum Model Checking: Declarative Representation and Verification of Properties. In *EC-TEL 2007 - Second European Conference on Technology Enhanced Learning*, volume 4753 of LNCS, pages 432–437, 2007. pages 67

[13] I. Barland, J. Greiner, and M. Vardi. Model Checking Concurrent Programs, 2005. pages 112, 118

[14] P. Brusilovsky, E. Schwarz, and G. Weber. A tool for developing adaptive electronic textbook on WWW. In *Proceeding of WebNet'96 - World Conference if the Web Society*, Boston, Ma, USA, 1996. pages 19

[15] P. Brusilovsky, E. Schwarz, and G. Weber. ELM-ART: An intelligent tutoring system on world wide web. In *Proceedings of the Sixth International Conference on User Modeling*, pages 261–269, Berlin, 1996. pages 18

[16] P. Brusilovsky and J. Vassileva. Course sequencing techniques for large-scale web-based education. In *International Journal of Cont. Engineering Education and Lifelong Learning*, volume 13(1-2), pages 75–94, 2003. pages 21, 22, 28, 29, 45

[17] Peter Brusilovsky. Course sequencing for static courses? applying ITS techniques in large-scale web-based education. In *Intelligent Tutoring Systems*, pages 625–634, 2000. pages 21

[18] C. Castro, B. Crawford, and E. Monfroy. A Quantitative Approach for the Design of Academic Curricula. In *Human Interface and the Management of Information. Interacting in Information Environments*, volume LNCS 4558, pages 279–288, 2007. pages 43

[19] Carlos Castro and Sebastián Manzano. Variable and Value Ordering When Solving Balanced Academic Curriculum Problems. In *Proceedings of 6th Workshop of ERCIM WG on Constraints*, Prague, June 2001. pages 40, 43

[20] O. E. M. Clarke and D. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 2001. pages 114

[21] Department of Electronics CM316 Multimedia Systems Coursework and Computer Science. Learning objects - does size matter? pages 11, 13

[22] M. del Mar Gallardo, P. Merino, and E. Pimentel. Debugging UML Designs with Model Checking. Journal of Object Technology, July-August 2002. pages 112

[23] E. Della Valle, I. Celino, and D. Cerizza. *Semantic Web. Modellare e condividere per innovare*. Addison-Wesley, 2008. pages 5

[24] R. Farrell, S. D. Liburd, and J. C. Thomas. Dynamic assebly of learning objects. In *Proc. of WWW 2004*, New York, USA, May 2004. pages 21, 29, 31

[25] N. Guelfi and M. Mammar. A Formal Semantics of Timed Activity Diagrams and its PROMELA Translation. In *Proc. of the 12th Asia-Pacific Software Engineering Conference (APSEC'05)*, volume IEEE Computer Society, pages 283–290, 2005. pages 112

[26] Nicola Henze. Adaptive Hyperbooks: Adaptation for Project-Based Learning Resources, 2000. pages 17, 19

[27] Gerard J. Holzmann. The Model Checker Spin. In *IEEE Transaction on Software Engineering*, volume 23 NO. 5, May 1997. pages 119

[28] Gerard J. Holzmann. *The SPIN Model Checker, Primer and Reference Manual*. Addison-Wesley, 2003. pages 60, 71, 74, 81, 117, 118

[29] M. Huth and M. Ryan. *Logic in Computer Science*. Cambridge Univ. Press., 2004. Second edition. pages 59, 116

[30] P. Inverardi and H. Muccini. CHARMY: A Plugin-based Tool for Architectural Analysis. In *ESEC-FSE05, The fifth joint meeting of the European Software Engineering Conference*, September 2005. pages 112

[31] Joost-Pieter Katoen. *Concepts, Algorithms and Tools for Model Checking*. Lecture Notes of the Course "Mechanised Validation of Parallel Systems", 1998/1999. pages 114

[32] T. Lambert, C. Castro, E. Monfroy, M. C. Riff, and F. Saubion. Hybridization of Genetic Algorithms and Constraint Propagation for the

BACP. In *Logic Programming*, volume LNCS 3668, pages 421–423, 2005. pages 43

[33] T. Lambert, C. Castro, E. Monfroy, M. C. Riff, and F. Saubion. Hybridization of Genetic Algorithms and Constraint Propagation: application to the Balanced Academic Curriculum Problem. In *Artificial Intelligence and Soft Computing – ICAISC 2006*, volume LNCS 4029, pages 410–419, 2006. pages 43

[34] T. Lambert, C. Castro, E. Monfroy, and F. Saubion. Solving the Balanced Academic Curriculum Problem with an Hybridization of Genetic Algorithm and Constraint Propagation. In L. Rutkowski et al., editor, *ICAISC 2006*, volume LNAI 4029, pages 410–419, 2006. pages 39, 40, 43

[35] J. Lilius and I. P. Paltor. vUML: a Tool for Verifying UML Models, May 1999. pages 112

[36] E. Marengo. Realizzazione di un web service prolog per la pianificazione di curricula di studi: un approccio ad azioni nel Semantic Web., 2006. Relazione di Tirocinio in informatica, Corso di studi in Informatica, Università degli Studi di Torino. pages 148

[37] V. Mascardi, P. Rosso, and V. Cordì. Enhancing Communication inside Multi-Agent Systems. An Approach based on Allignment via Upper Ontologies. In M. Baldoni, C. Baroglio, and V. Mascardi, editors, *Proceedings of MALLOW-AWESOME'007. First Workshop on Agents, Web-Services, and Ontologies.*, pages 92–107, 2007. pages 149

[38] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, Marzo 1997. pages 40

[39] Permanand Mohan and Christopher Brooks. Learning Object on the Semantic Web. In *Proc. of the 3rd IEEE International Conference onAdvanced Learning Technologies*, Athens, Greece, 2003. pages 12, 13

[40] OMG. Unified Modeling Language: Superstructure, version 2.1.1. OMG, Object Management Group, February 2007. pages 99, 105, 108

[41] V. Patti. *Programming Rational Agents: a Modal Approach in a Logic Programming Setting*. PhD thesis, Dipartimento di Informatica, Università degli studi di Torino, Italy, 2002. pages 48

[42] O. Pistamiglio. Sviluppo di un plugin grafico per eclipse: DCML Designer., 2007. Laurea specialistica in informatica, Corso di studi in Sistemi per il Trattamento dell'Informazione, Università degli Studi di Torino. pages 146

[43] Stuart Russell and Peter Norvig. *Artificial Intelligence, A Modern Approach*. Prentice Hall Series in Artificial Intelligence, 2003 second edition. pages 47, 54

[44] M. P. Singh. A Social Semantics for Agent Communication Languges. In *Issues in Agent Communication*, volume 1926 of LNCS, pages 31–45, 2000. pages 65

[45] Paolo Terenziani, Laura Giordano, Alessio Bottrighi, Stefania Montani, and Loredana Donzella. SPIN Model Checking for Verification of Clinical Guidelines. pages 146, 147

[46] W. M. P. van der Aalst and M. Pesic. DecSerFlow: Towards a Truly Declarative Service Flow Language. In Mario Bravetti and Gialuigi Zavattaro, editors, *Proc. of WS-FM*, LNCS, Vienna, September 2006. Springer. pages 65

[47] Jarmo Viteli. Finnish Future: From eLearning to mLearning? In *Proceedings for the ASCILITE Conference*, Perth, Australia, 2000. pages 10

[48] G. Weber and M. Specht. User modeling and adaptive navigation support in www-based tutoring systems. In *Proceedings of the Sixth International Conference in User Modeling, UM97*, Sardegna, Italia, 1997. pages 18

[49] Kun Wu and William S. Havens. Modelling an Academic Curriculum Plan as a Mixed-Initiative Constraint Satisfaction Problem. In B. Kégl

and G. Lapalme, editors, *Advances in Artificial Intelligence*, volume LNAI 3501, pages 79–90, 2005. pages 33, 34

[50] P Yolum and M. P. Singh. Reasoning about Commitments in the Event Calculus: An Approach for Specifying and Executing Protocols. *Annals of Mathematics and Artificial Intelligence*, 42(1-3):227–253, 2004. pages 55, 98

[51] P. Yolum and M.P. Singh. Flexible Protocol Specification and Execution: Applying Calculus Planning using Commitments. In *AAMAS'02*, pages 527–534, Bologna,Italy, 2002. pages 55

Appendice A

Articoli correlati

Di seguito vengono allegati i seguenti articoli, relativi alle proposte descritte in questa tesi:

[1] M. Baldoni, C. Baroglio, I. Brunkhorst, N. Henze, E. Marengo, and V. Patti. Constraint Modeling for Curriculum Planning and Validation. Sottoposto per la pubblicazione nell'*Interactive Learning Environments journal, Special Issue on "Semantic Technologies for Multimedia-anhanced Learning Environments"*, 2008. pages

[2] M. Baldoni, C. Baroglio, G. Berio, and E. Marengo. Declarative representation of curricula models: an LTL- and UML-based approach. In *Proc. of WOA 2007: Dagli oggetti agli agenti, Agenti e Industrie: Applicazioni tecnologiche degli agenti software*, pages 34–41, Genova, Italy, September 2007. pages

[3] M Baldoni, C Baroglio, and E. Marengo. Curricula Modeling and Checking. In *Proc. of AI*IA 2007: Advances in Artificial Intelligence, 10th Congress of the Italian Association for Artificial Intelligence*, volume 4733 of LNAI, pages 471–482, Rome, Italy, September 2007. pages

[4] M. Baldoni, C. Baroglio, I. Brunkhorst, E. Marengo, and V. Patti. A Service-Oriented Approach for Curriculum Planning and Validation. In *Proceedings of the Multi-Agent Logics, Languages, and Organisations, Federated Workshops, MALLOW'007, Agent, Web Services and*

Ontologies, Integrated Methodologies, MALLOW-AWESOME'007, Durham, GB, September 2007. pages

[5] M. Baldoni and E. Marengo. Curriculum Model Checking: Declarative Representation and Verification of Properties. In *EC-TEL 2007 - Second European Conference on Technology Enhanced Learning*, volume 4753 of LNCS, pages 432–437, 2007. pages

[6] M. Baldoni, C. Baroglio, I. Brunkhosrt, E. Marengo, and V. Patti. Reasoning-based Curriculum Sequencing and Validation: Integration in a Service-Oriented Architecture. In *EC-TEL 2007 - Second European Conference on Technology Enhanced Learning*, volume 4753 of LNCS, pages 426–431, 2007. pages

M. Baldoni, C. Baroglio, I. Brunkhorst, N. Henze, E. Marengo, and V. Patti. Constraint Modeling for Curriculum Planning and Validation. Sottoposto per la pubblicazione nell'Interactive Learning Environments journal, Special Issue on "Semantic Technologies for Multimediaanhanced Learning Environments", 2008. pages 57

Constraint Modeling for Curriculum Planning and Validation

Matteo Baldoni[1], Cristina Baroglio[1], Ingo Brunkhorst[2], Nicola Henze[2], Elisa Marengo[1], Viviana Patti[1]

[1] Dipartimento di Informatica — Università degli Studi di Torino
C.so Svizzera, 185 — I-10149 Torino (Italy)
{baldoni,baroglio,patti}@di.unito.it
[2] L3S Research Center, University of Hannover, Germany
{brunkhorst,henze}@l3s.de

Abstract. Flexibility in studying, mobility of students, and exchange and collaboration of universities are fostered by initiatives like the Bologna process. It becomes more and more important that students can perform their studies at different universities, following individually tailored and optimized study plans (so-called *curricula*) which conform to the regulations of the university / universities. Curricula, that match the students interest in certain competencies, as well as respect the already acquired ones. This paper introduces a novel approach for the construction of individually tailored curricula. It translates the construction process into a planning problem and provides with the Declarative Curricula Model Language (DCML) an effective tool for visually modeling constrained-based curricula. Curricula specified in DCML are automatically verified and feasible study plans are presented to the students which take the student's individual preferences for topics into account and simultaneously guarantee the compliance of the study plan with the regulations imposed by the universities. The curriculum planner has been realized as a Web services, and has been integrated into the Personal Reader framework, a Plug & Play - environment for realizing personalized information systems in the Semantic Web.

1 Introduction

The birth of the Semantic Web brought along standard models, languages and tools for representing and dealing with machine-interpretable semantic descriptions of Web resources, giving a strong new impulse to research on personalization. The introduction of machine-processable semantics, in fact, widens the range of the the applicable reasoning techniques, thus fostering the implementation of new personalization functionalities. It is more and more often the case when repositories of *learning resources* [23] are available on the web. By the term "learning resource" we here mean any object that provides content with educational purposes. One of the greatest problems that arise in this context is how to identify learning resources to be recommended or, more generally, how

to *re-use* learning resources. This is a serious problem because on one hand, the production of learning resources is often expensive, on the other hand, making the wrong choice for recommendation will confuse learners. For instance, recommending a resource just because it provides some content without checking if that content requires some basic knowledge to the user and without checking if the user has this knowledge makes the recommendation useless. In order to solve this kind of problems there is the need of describing learning resources by means of *meta-data*, which, in a Semantic Web setting, consist of machine-interpretable semantic annotations [45, 39]. The use of semantic annotations is even more important considering that learning resources are *heterogeneous* in different ways, for instance they can be supplied as *different media* (textual documents as well as animations), they can have *different granularities* (single lessons as well as courses), and they can be gathered from *different repositories*.

So far, reasoning in the Semantic Web [18, 17, 6] is mostly reasoning about knowledge, expressed in some *ontology* [40, 5]. However, personalization may involve also other kinds of reasoning and other ways for representing knowledge, that conceptually lie at the *logic* and *proof* layers of the Semantic Web tower [4]. Moreover, the next Web generation promises to deliver Semantic Web Services, that can be retrieved and *combined* in a way that satisfies the user, opening the way to forms of *service-oriented* personalization. Web services provide, in fact, an ideal infrastructure for constructing Plug&Play-like environments, where the user can select and combine the adaptive functionalities he or she prefers. The object of our studies is the problem of supporting the construction of *personalized compositions of semantically annotated learning resources*. In the remainder of the article we will refer to such compositions as to *curricula*. We do this by both exploiting reasoning techniques, that lay at the proof and logic layers of the semantic web tower (namely, planning, model checking, temporal reasoning), and adopting a service-oriented architecture in the implementation. In particular, when the problem of supporting the personalized (automatic) compositions of semantically annotated learning resources, *independently* from the kind of resources which are combined, is faced, three main issues arise [20]: (a) be sure that the curriculum allows the acquisition of the desired knowledge; (b) be sure that the curriculum is sound, i.e. that at every point the knowledge that is necessary for understanding the contents of the subsequent learning resource has been supplied; (c) be sure that the curriculum respects the design goals, which capture the intended educational purposes of the curriculum.

The first task that it is useful to make automatic is the verification that a curriculum, designed by a user by composing a set of learning resources, allows the achievement of the declared user's *learning goal*, i.e. of the knowledge the user wishes to acquire. A simple model for allowing this verification consists in representing each learning resource by means of the set of its educational content, i.e. the *competencies* it supplies. By doing so, we separate the description of the resource from the resource itself [10, 13, 9]. In this way, it is easy to check that the composition supplies all the desired knowledge. Normally, however, learning resources might have also *pre-requisite* constraints, i.e. in order to

learn what the resource teaches, it is necessary to have some background knowledge. In this case, it becomes possible to perform a second reasoning task: the verification that the curriculum shows no *competence gaps* [24], i.e. the knowledge that is necessary to fully understand a learning resource is introduced or available before the learning resources having it as a pre-requisite are accessed. These verifications can be easily performed by applying *temporal reasoning* [28] or *model checking* techniques [22]. Given a representation of learning resources in terms of pre-requisites and supplied competences, it becomes possible to apply *planning* techniques [42] for building curricula automatically. Such curricula can be tailored to a specific user by starting from an initial state that contains *that* user's knowledge.

This perspective is taken in several works that face the so called *curriculum sequencing* problem [35, 20, 48], that is the problem of generating a personalized learning path for each student by dynamically selecting at any moment a resource that, in the context of other available resources, brings the student closest to the learning goal. In most of the cases, the exploited models represent knowledge as a *set (or a network) of concepts*, where each concept represents an atom of knowledge about the learning domain. Then, each available learning resource is annotated by the concepts it deals with (for instance by knowledge prerequisites and supplied knowledge), leading to the creation of a knowledge-based representation for all the single resources in the repository.

We also adopt a similar approach. In particular, our representation relies on a semantic annotation of resources based on an RDF vocabulary. The semantic annotation of the resources can be in principle provided by different authors. This is a natural assumption in an e-learning framework, where every resource would be annotated by the person who produced it, and it is also a natural assumption for semantic web-based systems in general, where the repository of available resources are typically distributed. More specifically, we rely on the interpretation of learning resources as *actions* discussed in [13, 14], where the meta-data capture the *learning objectives* of the learning resource and its *prerequisites*.

The knowledge model of the learning resources depicted above can be used by a reasoning engine for accomplishing a curriculum sequencing task as well as for accomplishing simple verification tasks like learning goal achievement and presence of competence gaps. However, there are other tasks that this knowledge model alone does not support. For instance, not every learning resource composition makes sense from an educational perspective. Indeed, many educational institutes define *guidelines*, the *design goals*, which, rather than specifying curricula, specify those rules which a good curriculum should respect. Often such constraints express *temporal dependencies* at the knowledge level, i.e. among the competences that are acquired by attending the curriculum. Such constraints establish the model of curricula educational goals and strategies –*curricula model*, in short– which is naturally separated by the model describing the semantic annotation of the single resources. The curricula model can be used for making automatic the task of verifying the compliance of a given curriculum against the

guidelines given by the institution [20], a tedious task that is currently performed manually. In Section 4 we present a constraint-based representation of curricula models. Constraints are expressed as formulas in Linear Temporal Logic (LTL),[25], represented by means of a graphical language that we have defined and which is named Declarative Curricula Model Language (DCML). As we show, it is possible to exploit *model checking* techniques [22] for validating a given curriculum w.r.t a curricula model.

Notice that there are further kind of knowledge models that one should consider in order to build a complete system for combining resources. Such models (some example can be found in [49, 21, 37]) basically bind the fruition of the learning resources according to time, location, and so on, and therefore handle constraints such as maximum load, mandatory requirements (e.g. some courses must be attended in order to attend a bachelor's degree in Philosophy) and prerequisite constraints at the course level (e.g. a given course cannot be planned for a semester unless another course, which is stated as prerequisite, have been planned in previous semesters).

Besides the definition of the reasoning tasks and of the necessary knowledge models, and after the definition of DCML, we have also integrated within the framework supplied by the *Personal Reader for e-learning* (PR) [34] the personalization functionalities depicted above, in order to support users –students, teachers and institutions– in the task of combining resources into academic curricula. The PR relies on a *service-oriented architecture* enabling personalization, via the use of semantic *Personalization Services*. Each service offers a different personalization functionality, e.g. recommendations tailored to the needs of specific users, pointers to related (or interesting or more detailed/general) information, and so on. Such web services communicate solely based on RDF documents. The data that we have used in this work has been extracted automatically by applying Lixto [16] to the web-pages supplied by HIS-LSF (http://www.his.de). In the following, after the motivations and related works (Section 2), we present our proposal for representing learning resources and curricula (Section 3), DCML (Section 4), and the developed reasoning techniques (Section 5). Afterwards, we present the implementation of an architecture that includes personalization services that are based on the proposed formalization (Section 6).

2 Scenario and related works

The need of personalizing the sequencing of semantic learning resources, w.r.t. the student's learning goal and context, has often to be *combined* with the ability to check that the resulting curriculum *complies* to some abstract *specification*, which encodes the *curricula-design goals*, expressed by the teachers or by the institution offering the courses. Let us consider the living matter concerning the implementation of processes like cooperation among institutes in curricula design and integration, which are actually the focus of the so called *Bologna Process* [26], promoted by the EU. The Bologna process objectives are designed to foster the mobility of students, the design of *easily comparable academic programs*

and the deployment of *comparable educational criteria and methodologies*. The final aim is drawing up *integrated curricula* for European students, where the various European institutions offer specific segments which may complement and altogether allow for reaching the desired educational goals.

In order to allow the integration of the available courses (i.e. the learning resources in the given scenario) as steps of an integrated curriculum, each university should provide a *semantic annotation* of the courses in terms of concepts belonging to a shared ontology. This would lead to the creation of an enormous semantic distributed repository of resources annotated by the different partners. Second, different academic institutions should design their guidelines as constraints for guaranteeing the acquisition of certain educational goals according to given teaching strategies.

In this setting, let us consider an Italian student who decides to move to Germany for a period, s/he should be guided by an Italian mentor in the generation of a curriculum that combines her/his interests, the educational goals of the Italian academic institution and the description of the courses offered by and of the local constraints posed by the German hosting institution.

The student's *interests* will be represented as a set of competencies specifying the *learning goal*. The initial knowledge of the student can be represented as a set of competences. Instead, the educational goals of the Italian and of the German academic institutions will be specified as *curricula models*, while the description of the courses offered by the German partner are encoded by a *semantic annotation*. Given such knowledge representation framework, the student, the mentor, and the institutions can use a set of services for accomplishing various tasks. The *student* can, for instance, assemble a curriculum manually, and then use a validation service for checking whether by it he/she will achieve the *learning goal*, whether there a are *competence gaps*, and the *compliance* w.r.t. the German curricula model. Notice that, once the German institution makes available also information about the time periods (e.g. semesters) of the offered courses, it would be possible to take into account such constraints during the validation process of the curriculum manually composed by the student.

Moreover, the *German institution* could use a curriculum planner for offering automatically to its students personalized plans that fit student specific learning goals and the requirements of the institution. The *mentor* could use a validation service for checking if the curriculum automatically generated by the German institution is also compliant against the Italian educational guidelines.

The possibility to check specific curricula against constraints stored in curricula models, makes it easier for academic institutions to deal with updates in the guidelines provided by higher-order institutions. It is possible, for instance, to automatically check whether the curricula currently offered by a given University respect also the latest specification published by the Ministry of Education or by the European Community.

2.1 Related works

In the literature it is possible to find different works that are related to ours in different ways. A recent proposal for automatizing the *competence gap verification* is done in [38] where an analysis of pre- and post-requisite annotations of the Learning Objects, representing the learning resources, is proposed. A logic based validation engine can use these annotations in order to validate the curriculum/learning object composition. Melia and Pahl's proposal is inspired by the *CocoA* system, by Brusilovski and Vassileva [20], that allows for 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". Along this line, also Garro et al. [30] propose an approach to the automatic construction of *learning paths* (analogous to curricula), that has been implemented as part of the MASEL system [31]. A learning path is a sequence of learning objects, which allows the acquisition of a *skill* (a competence, in our terminology) that is desired by the user. Learning object have a rich description; in particular, each learning object has some effects (skills that are supplied). Learning objects do not have preconditions, which are substituted by a "pre-requisite" relation between skills (this skill is to be acquired before this other skill). The approach also allows for the representation of the duration of learning objects and of the complementarity of certain sets of learning objects, moreover, skills include a proficiency level. In this work, however, the only reasoning task that is handled is planning, they do not have an explicit representation of curricula models and, therefore, they do not face the verification issues.

Brusilovsky and Vassileva [19, 20] adopt a method that is close, in principle, to curriculum sequencing: *course sequencing*. The aim of course sequencing, a technique originally proposed in the field of Intelligent Tutoring Systems, is to supply users with personalized courses, which select, at every step of the learning process, the best teaching method, i.e. the teaching method that will help the user to get the closest to his/her learning goal. The term 'teaching method' refers, for instance, to the possibility of facing an exercise rather than reading more detailed documentation about a topic. In particular, two models are proposed: DCG and CoCoA. Both systems help the construction of personalized courses on the basis of a semantic network, which composes a set of Domain Knowledge Elements, roughly corresponding to our competences. DCG organizes such elements in an AND-OR graph, while CoCoA adopts an heterarchic structure relying on the relations *part-of* and *attribute-of*. Both organizations represent the domain model. DCG applies "dynamic planning" techniques: at every step the student's advancements are verified by a test. If the test is passed, a new topic is presented, otherwise some replanning is performed in order to allow the student to fill the gaps. Each node in the domain model can be presented in different ways (e.g. test, exercise, motivation, example). A presentation plan component has the task of identifying the best presentation of the concept for that particular student in that particular context. This selection is performed based on a set of "teaching rules" encoded in the system. Dynamic planning fits

very well the task of building student-oriented personalizations but it does not scale equally well to class-based personalization unless all students in class have similar learning skills. CoCoA supports the construction of courses by performing a set of operations among which consistency checks and quality checks. Also in this case different kinds of presentation are identified. CoCoA can perform *prerequisite checks* by simulating the execution of a course and verifying at every step that the preconditions to the current step are satisfied, i.e. that at that point the student will have the necessary knowledge. Each kind of presentation of each concept has its own prerequisites (so there are question prerequisites, presentation prerequisites, etc.). A limit of this approach is that prerequisites allow the expression of a single kind of constraint: what is to be learnt before a certain learning step can be accomplished. We have shown, by presenting DCML, that there are many other kinds of relation that it is interesting to capture, leading to the introduction of a further level of abstraction: the one given by our curricula model.

In [20] Brusilovsky and Vassileva also define various verification tasks, besides competence gaps, among which two tasks that we accomplish in the present proposal: (a) verifying that the curriculum allows the achievement of 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. In [20] curricula models are said to specify general rules for designing sequences of learning resources (courses). We interpreted 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.

Another proposal is the one by Farrell, Liburd, and Thomas [27], who propose an approach called *Dynamic Assembly*. This approach allows the automatic generation of a course by composing pre-existing learning objects on the base of a set of parameters (e.g. level of detail, time, keywords) which are specified by the user. Parameters are expressed in a *course assembly page*. The system provides two methods, namely *indepth* and *overview*, which respectively narrow the search to the topics listed by the user and return also learning objects concerning related topics. The result of the search is a numbered sequence of links to learning objects. The approach does not supply methods for verifying that the produced sequences, besides reaching the goal, also "make sense" from an educational point of view, for instance by exploiting some abstract model.

The works in [49, 21, 37] focus on the curriculum planning problem intended as the problem of planning the different courses of an academic curriculum on a given set of time periods (e.g. semesters), satisfying some academic (e.g. course availability, prerequisites, eligibility rules) or student constraints. The problem of planning an academic schedule can be naturally tackled by adapting to the application context a *Constraint Satisfaction Problem (CSP) modeling frame-*

work, as proposed in [49, 21, 37]. In general, CSP models can be considered a good starting point for dealing with bindings to the fruition of the learning resources that may depend on the kind of infrastructure supplied by the academic institution (e.g. number and location of rooms, support services for computing, number of teachers and so on). In particular, Wu and Havens [49] have proposed an extended model that exploits *mixed-initiative* constraint reasoning algorithms and provides flexibility in satisfying not only the constraints given by the institution offering the courses, but also the student's preferences and needs. In a first phase (*curriculum planning*), a set of courses available in a certain semester is identified by interpreting the task as a constraint satisfaction problem with preferences. Courses are, then, presented to the user by means of an interface that organizes them in a table, whose columns correspond to teaching periods. The user can adjust the presented solution by modifying part of it. The first plan is obtained by applying the constraint given by the institution, while the modification actions taken by the user are interpreted as user constraints. A modified plan is validated in order to see if it still respects the overall set of constraints and then the interaction with the user can be repeated. The kinds of constraint that the system can handle are of different types. Among them: a course can appear only once per plan, courses can require other courses (not competences) as prerequisites, it is not possible to attend "equivalent" courses, some courses are mandatory. As one can observe, this kind of constraints is not defined on the competences required/thought by courses nor they express the constraints on the fruition of the teaching materials supplied by the teacher.

Last but not least, in [21, 37] the authors tackle the problem of building *balanced academic curricula*. This term identifies curricula in which courses are well-distributed, according to a set of *load constraints*, along the trimesters/years. Besides load constraints also constraints concerning course prerequisites are considered during the planning phase. As in the previous approach, a course prerequisites correspond to other courses which are to be attended before the one at issue. The problem is interpreted as an optimization problem, which can be tackled in different ways (e.g. branch and bound, constraint programming, genetic algorithms).

3 Representing Learning Resources and Curricula

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 this work, learning resources are represented by a set of *preconditions* and a set of *effects*. Both, preconditions and effects, are sets of *competencies*, i.e. ontological terms denoting a knowledge element [14]. Preconditions gather all those competences that are to be owned by the user in order to learn by using the resource, while effects gather the knowledge elements that are supplied by the resource. Each competence includes the *proficiency level* at

which the corresponding competency is owned or supplied. Proficiency levels range over a set of possible values, for instance *beginner*, *advanced*, *expert* or any other enumeration. In the following we will use a set of *integer* values.

This representation is general because it does not make any assumption on some specific kind of resource that is handled, because it expresses an information that lays at the knowledge level. As an example, let us consider a learning resource with name `db_for_biotech`, which requires beginner's knowledge about relational databases and supplies advanced knowledge about scientific databases. It will be semantically annotated as follows:

resource_name: db_for_biotech,
preconditions: (relational_db, beginner)
effects: (scientific_db, advanced)

So, along the line of [13, 9, 12], we can intuitively interpret a learning resource as an action, that can be executed given that its preconditions hold (a learning resource can intuitively be used profitably if the user has certain knowledge). By executing it, a set of post-conditions, the effects, will become true (the user will acquire new competences).

Let us now consider specific curricula. In the simplest case, a curriculum can be 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 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 courses 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* [1]. The diagram represents essentially the "student personal process" to achieve the final degree. Apart its standard meaning and visualization, 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 organization. Activity diagrams are rich enough to represent alternative, intermediate statuses and conditional paths.

However, we found two principles very useful 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.

Activity diagrams are well suited for representing curricula under the two principles reported above. Fig. 1 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

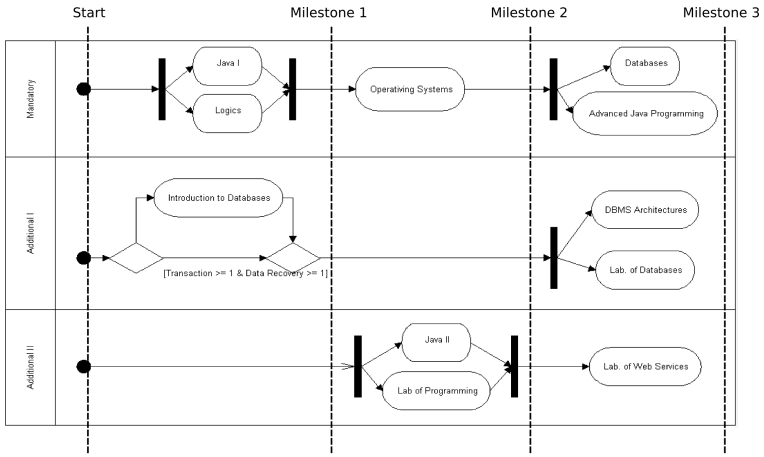


Fig. 1. Activity diagram representing a curriculum with mandatory and additional, student chosen, courses. Swimlanes represent the sequencings of courses. Vertical divisions capture the different milestones (trimesters).

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 one-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 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 some 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 of the courses to supply competences that are pre-requisites for the other. Again, with reference to our example, in Fig. 1, 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 visualized 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 visualized 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).

4 Curricula models and DCML

In this work we propose a constraint-based representation of curricula models. Constraints are expressed as formulas in a temporal logic (LTL, linear temporal logic [25]) represented by means of a simple graphical language that we call DCML (*Declarative Curricula Model Language*, see next section). 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. More specifically, the idea is to use a model checker in order to verify if it satisfies the constraints captured by the curricula model.

This approach differs from previous work [13], 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 a *schema*, which is implemented by means of prolog-like logic clauses. The use of procedure clauses is, however, limiting because they have a *prescriptive* nature and 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". If, on the one hand, it makes sense that some topics have a specific ordering, it is also intuitive that this does not happen for all the involved competences. For example, it makes sense that the

competence $(database, beginner)$ (knowledge about databases with proficiency level "beginner") is to be acquired before $(database, advanced)$ but what about a competence $(English, advanced)$? Should it be acquired before or after the other two mentioned competences? Intuitively, it is not necessary to impose that $(English, advanced)$ is, for instance, the first competence to acquire because it is not related to the other topics. The problem is that an ordering that is not explicitly mentioned in the model is not legal, so the designer must foresee many variants within the schema, one for each legal ordering of every pair of competences for which no temporal priority is defined. In the example, we should have at least four different sequences, in which English appears also at the second, third, and last position.

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 [43, 44]. 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, avoiding overspecification. 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. So in the previous case, we only need to express that $(database, beginner)$ must be acquired before $(database, advanced)$ and that $(English, beginner)$ is to be acquired sooner or later. Generally, the constraints-based approach is more flexible and more suitable in an open environment.

In order to allow the construction of curricula models we have defined a graphical language, named *Declarative Curricula Model Language* (DCML, for short), by means of which it is possible to express various kinds of constraints. DCML is inspired by DecSerFlow, the Declarative Service Flow Language to specify, enact, and monitor web service flows by van der Aalst and Pesic [47]. DCML, as well as DecSerFlow, is grounded in *Linear Temporal Logic* [25] 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 [36, Chapter 6]. The set of LTL formulas obtained for a curricula model are, then, used to verify whether a curriculum will respect it. 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. The adoption of a graphical language with a logical grounding

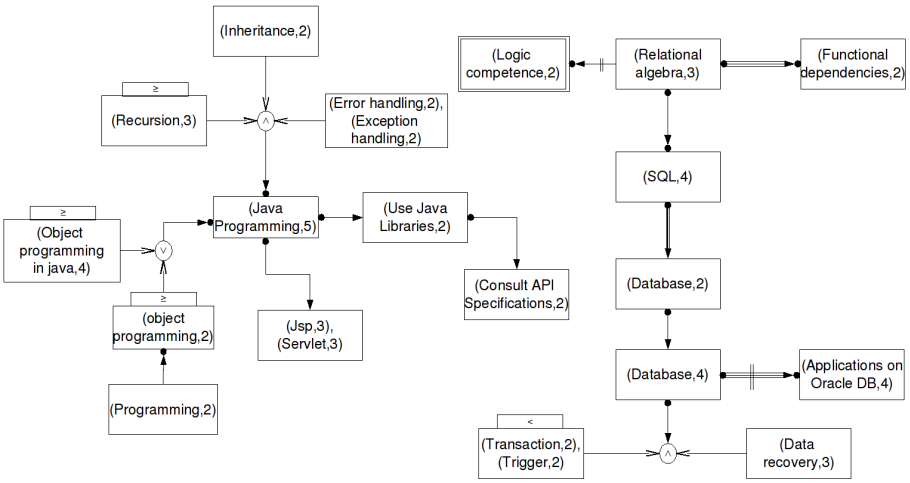


Fig. 2. An example of curricula model in DCML.

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, Fig. 2[1] shows a curricula model expressed in DCML. Every box contains at least one competence. Boxes/competences are connected by arrows, which represent (mainly) temporal constraints among the times at which they are to be acquired. Altogether the constraints describe a curricula model.

4.1 Expressing competences and basic constraints

DCML, as mentioned, includes 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. 3.

On top of competences, in DCML it is possible to define three basic *constraints*. The "*level of competence*" constraint, Fig. 3, 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

[1] Notice that triple arrows are defined in Section 7.

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. 3. 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. 3, 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.

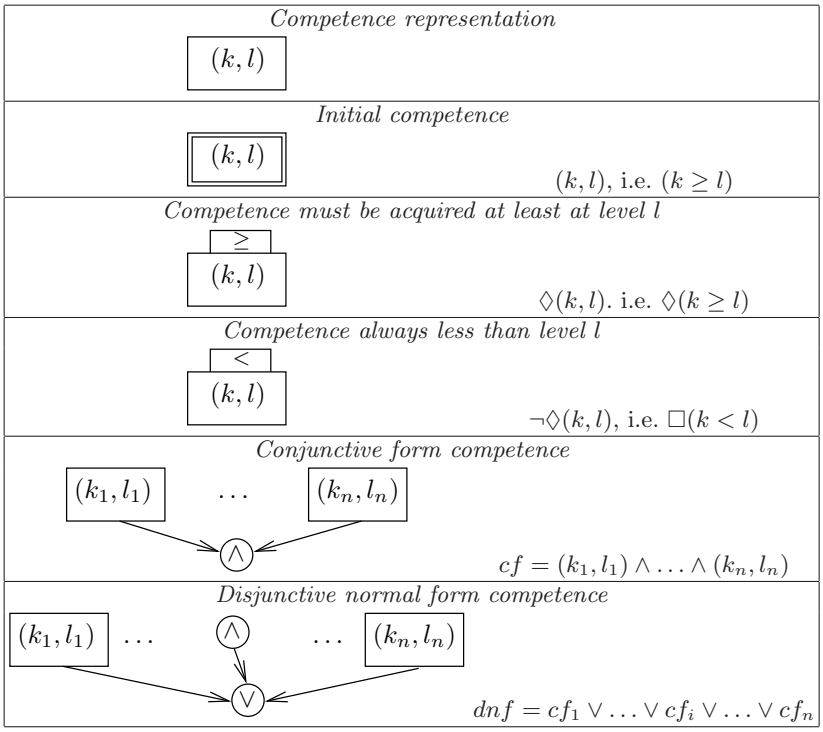


Fig. 3. DCML representation of competences, conjunctions and disjunctions of competences.

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[2]. Graphically, a conjunction of basic constraints is represented by a circle with a "∧" symbol inside, having as many incoming arrows as conjuncts. A disjunction, instead, can compose both basic

[2] A DNF formula is an expression having the form $(a_{1,1} \wedge a_{1,2} \wedge \ldots) \vee (a_{2,1} \wedge a_{2,2} \wedge \ldots) \vee \ldots$.

constraints and conjuncts of constraints. Graphically, it is represented by a circle with a "$\vee$" symbol inside, the disjuncts are connected to it by means of arrows.

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 *conjunctive competence formula* cf is a conjunction 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* ($\mathsf{negation}(cf)$), *level of competence* ($\mathsf{existence}(cf)$), and *always less than level* ($\mathsf{absence}(cf)$), to a conjunctive competence cf formula as follow:

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

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 conjunctive competence formulas $dnf = \{cf_1, \ldots, cf_n\}$. Therefore, a disjunctive normal competence formula is a set of sets of atomic competences.

4.2 Constraints among competences

Besides the representation of competences and of constraints on competences, DCML allows the representation of *relations* among competences. These relations are represented as different kinds of arrows. It is also possible to represent "negative relations" by using two vertical lines to break the arrow that represents the constraint.

There are three main kinds of relation: *before*, *implication*, and *succession*. 'Before' expresses the fact that in order to acquire a competence, another competence is to be owned in advance; 'implication' means that if a competence is acquired then also another competence must be present (either because it will be learnt or it has already been learnt); 'succession' means that when a competence is acquired, then another will be acquired in the future. In a way, succession combines the previous two relations, in fact, it expresses a temporal implication.

Before Arrows ending with a little-ball, Fig. 4, express the *before* temporal constraint between two competences. "(k_1, l_1) *before* (k_2, l_2)" requires that (k_1, l_1) holds *before* (k_2, l_2). This constraint can be used to express that to understand some topic (e.g., k_2 at least at the level l_2), some proficiency of another is required as precondition (in the example, k_1 at least at the level l_1). 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 $\neg(k_2, l_2)\ \mathsf{U}\ (k_1, l_1)$. More generally, in presence of DNF formulas as antecedent and consequence of a "before" relation, we have the following definition for "dnf_1 *before* dnf_2":

$$\bigvee_{cf_i \in dnf_1, cf_j \in dnf_2} \mathsf{negation}(cf_j)\ \mathsf{U}\ cf_i$$

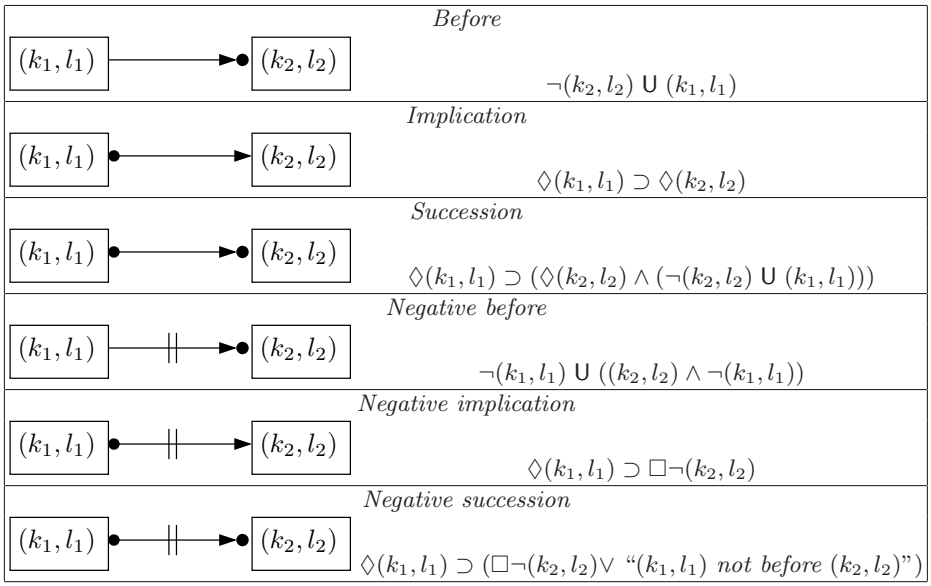


Fig. 4. DCML representation of basic constraints among competences. Arrows can connect not only competences but also conjunctions and disjunctions of competences.

"(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. More generally, in presence of DNF formulas, "dnf_1 *not before* dnf_2" is:

$$\bigvee_{cf_i \in dnf_1, cf_j \in dnf_2} \mathsf{negation}(cf_i)\ \mathsf{U}\ (cf_j \wedge \mathsf{negation}(cf_i))$$

Implication Before relation represent temporal constraint between competences. The *implication* relation, for example "(k_1, l_1) *implies* (k_2, l_2)", denoted by arrows starting with a little-ball (see Fig. 4), 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 $\Diamond(k_1, l_1) \supset \Diamond(k_2, l_2)$. More generally, in presence of DNF formulas as antecedent and consequence of a

"implication" relation, we have the following definition for "dnf_1 *implies* dnf_2":

$$\bigvee_{cf_i \in dnf_1, cf_j \in dnf_2} \mathsf{existence}(cf_i) \supset \mathsf{existence}(cf_j)$$

"(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, $\Diamond(k_1, l_1) \wedge \Box\neg(k_2, l_2)$, would impose the presence of certain concepts (k_1 at least at level l_1, in the example). More generally, in presence of DNF formulas, "dnf_1 *not implies* dnf_2" is:

$$\bigvee_{cf_i \in dnf_1, cf_j \in dnf_2} \mathsf{existence}(cf_i) \supset \mathsf{absence}(cf_j)$$

Succession The last constraint is *succession* (arrows starting and ending with a little-ball, Fig. 4). "(k_2, l_2) *succeeds* (k_1, l_1)" 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 $\Diamond(k_1, l_1) \supset (\Diamond(k_2, l_2) \wedge (\neg(k_2, l_2)\ \mathsf{U}\ (k_1, l_1)))$. More generally, in presence of DNF formulas as antecedent and consequence of a "succession" relation, we have the following definition for "dnf_1 *succeeds* dnf_2":

$$\bigvee_{cf_i \in dnf_1, cf_j \in dnf_2} \mathsf{existence}(cf_i) \supset (\mathsf{existence}(cf_j) \wedge \text{ "}cf_i \textit{ before } cf_j\text{" })$$

"(k_2, l_2) *not succeeds* (k_1, l_1)" imposes that a certain competence, (k_2, l_2), cannot be acquired after another, (k_1, l_1), 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)"). Similarly to the previous negative constraints, we choose to use a weaker formula than the natural negation of the succession relation because the simple negation of formulas, $\Diamond(k_1, l_1) \wedge (\Box\neg(k_2, l_2) \vee \neg$ "(k_1, l_1) *before* (k_2, l_2)"), would impose the presence of certain concepts (k_1 at least at level l_1, in the example). More generally, in presence of DNF formulas, "dnf_1 *not succeeds* dnf_2" is:

$$\bigvee_{cf_i \in dnf_1, cf_j \in dnf_2} \mathsf{existence}(cf_i) \supset (\mathsf{absence}(cf_j) \vee \text{ "}cf_i \textit{ not before } cf_j\text{" })$$

5 Automatic construction and verification of curricula

In the framework that we have defined, it is possible to execute different forms of automated reasoning. For what concerns the automatic construction of curricula, it is possible to exploit standard *planning techniques*, from the research

area of Artificial Intelligence [42]. In fact, given our interpretation of learning resources, a curriculum can, be interpreted as a *plan*, whose execution causes transitions from a state to another, until some final state is reached. A transition between two states is due to the application of the action corresponding to a learning resource. Of course, for an action to be applicable, its preconditions must hold in the state to which it is applied. The application of the action consists in an *update* of the state. We assume that facts can only be added to states. The intuition behind this assumption is that the act of using a new learning resource (e.g. attending a course) will not erase from the student's memory the concepts acquired so far. The *planning problem*, therefore, amounts to identify a *sequence* of learning resources, which does not show any competence gap, and such that it supplies the desired knowledge. The service that we have developed, that is described in details in Section 6.3, is specifically aimed at building *linear curricula*, i.e. curricula which consist of a sequence of courses only. The planning service works on a representation of German courses which has been automatically extracted from the web pages supplied by HIS-LSF by using Lixto [16].

For what concerns verification, instead, the tasks on which we have focussed and that are particularly meaningful, from an educational point of view, are:

1. the verification that a curriculum does not show *competence gaps*,
2. the verification that a curriculum supplies the user's *learning goal*, and
3. the verification that a curriculum satisfies the *course design goals*, i.e. the constraints imposed by the curricula model.

To perform these tasks, we use *model checking techniques* [22]. 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. Even though it is possible to argue that model checking techniques are not efficient because they analyze the whole space of possible executions, the fact of having counterexamples is extremely important because it supplies an *explanation* of the failure. This explanation is a very important feedback for the user. It is well-known that the user will not be confident in the answer unless he/she can trust the *reasons* why the answer was produced. Following [3], since in a Semantic Web system the answer is the result of a reasoning process, the justification can be given as a derivation of the conclusion. Moreover, all the constraints defined by means of DCML are in conjunction with one another, therefore, it is possible to check them one at a time, reducing the complexity of the problem drastically because the automaton resulting for a single constraint is small. As soon as a constraint verification fails, the procedure is stopped.

SPIN, by G. J. Holzmann [36], is the most representative tool of this kind. The approach that we propose consists in translating the activity diagram, that represents a (set of) curricula, in a Promela (the language used by SPIN) program, and, then, to verify whether this program satisfies the LTL formulas which

correspond to the curricula model. In this process it is possible to take into account specific knowledge about the user just by asserting in the initial state all the facts that are taken from the *user model*. The user model is a description of the user, typically used in personalization tasks, which is given in terms of all those characteristics that are considered as relevant in the learning task. Specifically, in our case it contains all the competences of the user (jointly with their proficiency levels).

In the literature, we can find some proposals to translate UML activity diagrams into Promela programs, such as [29, 32]. 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.

The verification process is, in a way, a simulation of the possible executions of the program. Every milestone corresponds to a *state*, containing specific competences (all those acquired up to that point, also due to the attended courses). The *initial state* contains all the facts concerning the user at issue, among which we have identified the competences that the user has from the start. This set can also be empty. This happens when we do not have any information about the user's knowledge, i.e. when we need to verify a curriculum generally offered by an institution against a curricula model. The assumption in this case is that the user has no knowledge. The *final state* contains the competences that are gained by using the various learning resources which lay upon a specific path.

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 correspond to the various milestones into which the curriculum is organized. For instance, in Fig. 1 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 "Lab. of Web Services" is as follows:

```
inline preconditions_course_databases() {
  assert(N_tier_architectures >= 4 && sql >= 2);
}
inline effects_course_databases() {
  SetCompetenceState(jsp, 4);
  SetCompetenceState(markup_language, 5);
}
```

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 alternatives.

```
inline milestone_1() {
  atomic {
    preconditions_course_java_I();
    preconditions_course_Logics();
    if
    :: true ->
        if
        :: (Transaction >= 1 && Data recovery >= 0) -> skip;
        :: else -> preconditions_course_Introduction_to_databases;
        fi
        path = 1;
    :: true ->
        path = 2;
    effects_course_Java_I();
    effects_course_Logics();
    if
    :: (path == 1) ->
        if
        :: (Transaction >= 1 && Data recovery >= 0) -> skip;
```

```
        :: else -> effects_course_Introduction_to_databases;
        fi
    :: (path == 2) -> 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.

6 Implementation in the Personal Reader Framework

We designed the Personal Reader (PR) Framework[3] as a tool and test-bed for creating applications in the Semantic Web, with a strong focus on Semantic Web Services. It offers an environment for designing, implementing and realizing Web content readers using a service-oriented approach, for a more detailed description, see [34]. In applications based on the Personal Reader Framework, users can select and combine —plug together— the personalization support they want to receive. The framework has already been used for developing Web Content Readers that present online materials in an embedded context [16, 2, 33].

Figure 5 gives a brief overview of the main components of the PR architecture. A typical PR application consists of three types of services. *Personalization services* (PService) provide personalization functionalities: they deliver personalized recommendations for content, as requested by the user and obtained or extracted from the Semantic Web. The *Syndication Services* (SynService) realize the user interface and facilitate interoperability with the other services in the framework, e.g. it allows for the discovery of the applications' interfaces by a portal. The *Connector* is a single central instance responsible for controlling

[3] `http://www.personal-reader.de/`

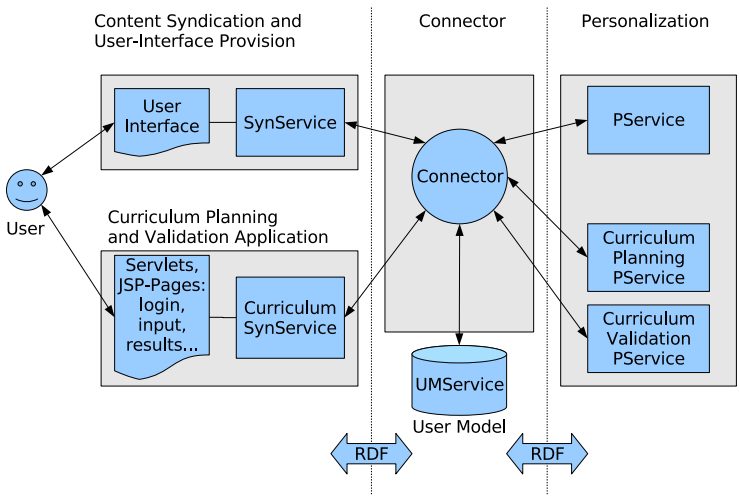


Fig. 5. Personal Reader Framework Overview.

the communication between user interface and personalization services. It selects services based on their semantic description and the requirements by the SynService. The Connector protects —by means of a public-key-infrastructure (PKI)— the communication among the involved parties. It also supports the customization and invocation of services and interacts with a user modelling service, called the *UMService*, which maintains a central user model.

For the implementation of the Curriculum Planning and Validation Application, the User Interface (SynService) was realized using Servlets and JSP Pages. Other applications, like the MyEar system [34] also use active scripting approaches (Java Script and Ajax) for creating adaptive user interfaces. For the actual data processing and personalization, two PServices were implemented, one containing the planner, the other hosting the validation engine.

6.1 Metadata Description of Courses

To apply our ideas to a real world scenario, we created the corpus of courses by extracting information from an existing database of courses. We used the the Lixto [15] tool to extract the needed data from the web-pages provided by the HIS-LSF[4] system of the University of Hannover. This approach was chosen based on our experience with Lixto in the *Personal Publication Reader* [16] project, where we used Lixto for creating the bibliographic database by crawling the publication pages of all the project partners. The effort to adapt our existing tool for the new data source was only small. From the extracted metadata we created the RDF knowledge base. For each of the courses, the course name, catalog

[4] `http://www.his.de/`

identifier, semester, the number of credit points, effects and preconditions, and the type, e.g. laboratory, seminar, or regular course with examinations in the end, was recorded. Figure 6 shows the metadata properties of the course *Digital Image Processing*.

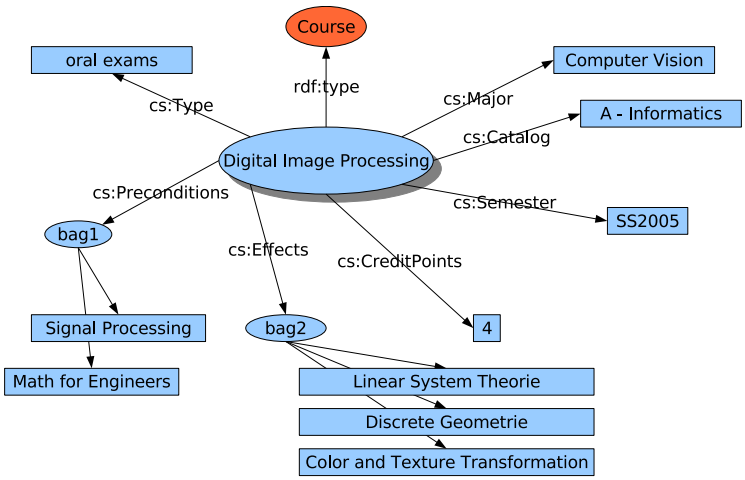


Fig. 6. Metadata for the course *Digital Image Processing* from the Hannover course database.

As it turned out, the biggest problem was that the quality of most of the information in the database is insufficient, largely because of inconsistencies in the description of prerequisites and effects of the courses. Additionally the corpus was not annotated using a common set of terms, but course authors and department secretaries each used a slightly varying vocabulary for the description of their learning objects, instead of relying on a common classification system, like e.g. the ACM Computing Classification System (ACM CCS[5]).

As a consequence, we focussed only on a subset of the courses (computer science and engineering courses), and manually post-processed the harvested data. In our system, courses are annotated with prerequisites and effects, which correspond to knowledge concepts or competences, i.e. ontology terms. After automatic extraction of effects and preconditions, the collected terms were spell-checked and harmonized, synonyms were removed and annotations were corrected where necessary. The resulting corpus had a total of 65 courses left, with 390 effects and 146 preconditions.

[5] `http://www.acm.org/class/`

6.2 The User Interface and Syndication Service

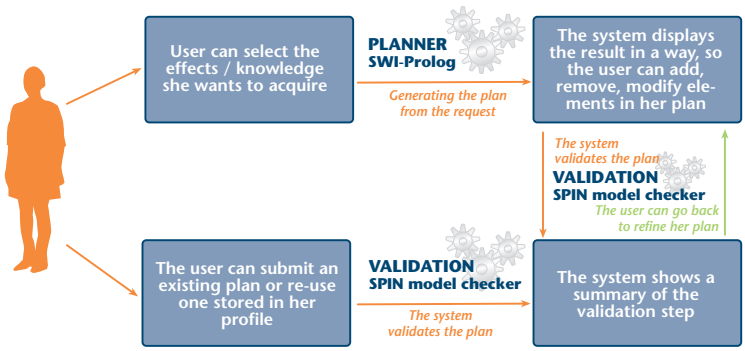


Fig. 7. The Actions supported by the User Interface.

The Syndication service (SynService) is responsible for creating the user interface to present to the user, as well as requesting the required personalization functions from the connector, as well as all communication between the different Web services. This interface has to fulfill multiple roles. It is responsible for identifying the user presenting the user an interface to select the knowledge to be acquired. Furthermore, it has to display the results of the planning and validation step (see figure 7), allowing the further refinement of created plans.

The creation of curriculum sequences and the validation is delegated to the two independent Personalization Services, the "Curriculum Planning PService", and the "Curriculum Validation PService".

Because of the Plug & Play nature of the infrastructure, the two PServices can be used by other applications (SynServices) as well (Fig. 7). It is also possible to use additional PServices by extending the SynService, to provide additional planning and validation capabilities to our application. The current implementation of the Curriculum Planning and Validation Prototype can be reached via the Projects page of the Personal Reader Homepage `http://personal-reader.de`.

Figure 8 is a screenshot of the initial learning goal selection page, as generated by the syndication service and presented to the user. It contains fields for up to three required goals, as well as the number of credit points to achieve and the number of plans to generate.

6.3 Automatic construction of curricula: the Curriculum Planning PService

We have developed a simple service for building personalized curricula, which has been integrated as a Plug & Play personalization service in the Personal

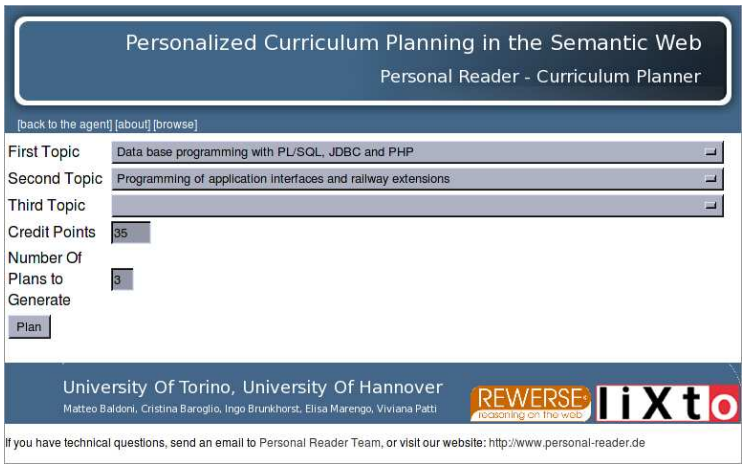


Fig. 8. Selection of Learning Goals.

Reader architecture. The curriculum is *personalized* in the sense that it allows a user to reach his/her learning goals, starting from the current competences the user has, which are included in the user model.

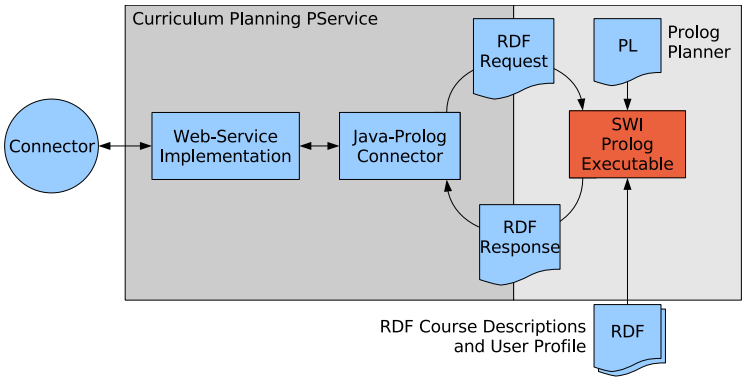


Fig. 9. Curriculum Planning Web Service.

The planner was implemented as a Prolog program, so we had to embed a Prolog reasoner into the Web service. The planner executes a simple *depth-first forward planning* (an early prototype was presented in [7]), where actions cannot be applied more than once. The algorithm is simple:

1. Starting from the initial state, the set of *applicable* actions (those whose preconditions are contained in the current state) is identified.
2. One of such actions is selected and its application is simulated leading to a new state.
3. The new state is obtained by adding to the previous one the competencies supplied as effects of the selected action.
4. The procedure is repeated until either the goal is reached or a state is reached, in which no action can be applied and the learning goal is not satisfied.
5. In the latter situation, backtracking is applied to look for another solution.

The procedure will eventually end because the set of possible actions is finite and each is applied at most once. If the goal is achieved, the sequence of actions that label the transitions leading from the initial to the final state is returned as the resulting *curriculum*. If desired, the backtracking mechanism allows to collect a set of alternative solutions to present to the user.

Figure 9 gives an overview over the components in the current implementation. The Web service implements the Personalization Service (*PService* [34]) interface, defined by the Personal Reader framework, which allows for the processing of RDF documents and for inquiring about the services capabilities. The *Java-to-Prolog Connector* runs the SWI-Prolog executable in a sub-process; essentially it passes the RDF document containing the request *as-is* to the Prolog system, and collects the results, already represented as RDF.

The curriculum planning task itself is accomplished by a reasoning engine, which has been implemented in SWI Prolog[6]. The interesting thing of using SWI Prolog is that it contains a semantic web library allowing to deal with RDF statements. Since all the inputs are sent to the reasoner in a *RDF request document*, it actually simplifies the process of interfacing the planner with the Personal Reader. In particular the request document contains: a) links to the RDF document containing the database of courses, annotated with metadata, b) a reference to the user's context c) the user's actual learning goal, i.e. a set of knowledge concepts that the user would like to acquire, and that are part of the *domain ontology* used for the semantic annotation of the actual courses. The reasoner can also deal with information about credits provided by the courses, when the user sets a credit constraint together with the learning goal.

Given a request, the reasoner runs the Prolog planning engine on the database of courses annotated with prerequisites and effects. The initial state is set by using information about the user's context, which is maintained by the User Modelling component of the PR. In fact such user's context includes information about what is considered as already learnt by the student (attended courses, learnt concepts) and such information is included in the request document. The Prolog planning engine has been implemented by using a classical depth-first search algorithm. This algorithm is extremely simple to implement in declarative languages as Prolog.

[6] `http://www.swi-prolog.org/`

At the end of the process, a *RDF response document* is returned as an output. It contains a list of plans (sequences of courses) that fulfill the user's learning goals and profile. The maximum number of possible solutions can be set by the user in the request document. Notice that further information stored in the user profile is used at this stage for adapting the presentation of the solutions, here simple hints are used to *rank higher* those plans that include topics that the user has an expressed special interest in. Figure 10 shows the output generated by the syndication service for the generated plan, decoded from the RDF response document. The interface also provides the means for the user to modify the plan and submit it for further validation.

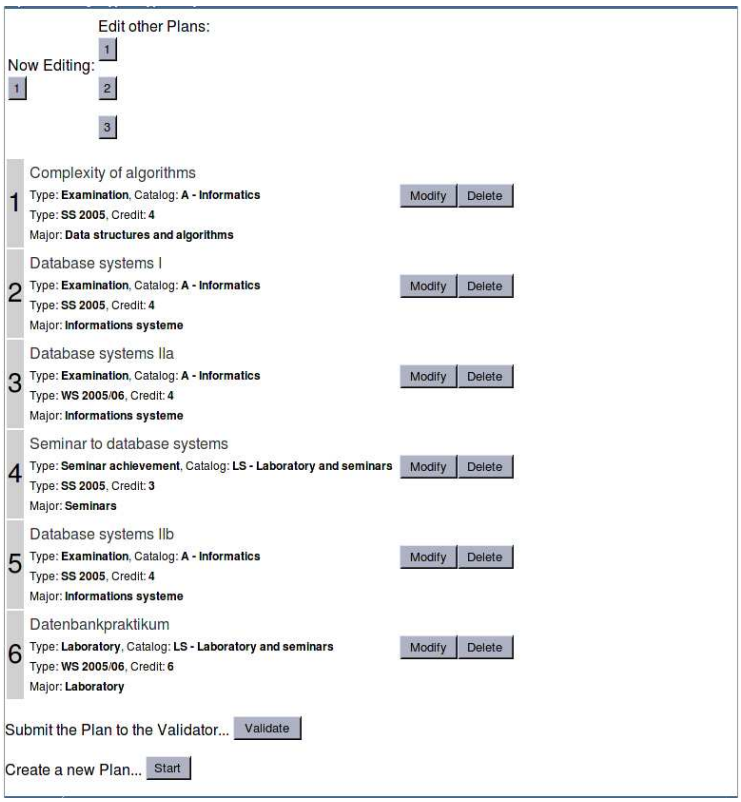


Fig. 10. Output of the Planning Step.

6.4 The Curricula Validation PService: current state of the implementation

The verification that a curriculum satisfies some course design goals requires a user, e.g. a student, to supply a curriculum, which is to be checked against a set of constraints expressed by a curricula model by following the process described in Figure 11.

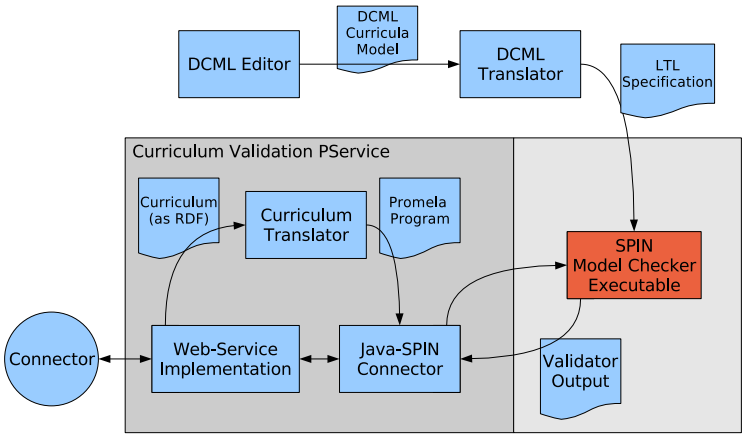


Fig. 11. The validation w.r.t. a curricula model: process workflow.

In order to build curricula models, we have developed an Eclipse plugin (the *DCML Designer*, see Figure 12 [41]). This tool is thought for being used by the *instructional designers* of an academic institute. The task of an instructional designer is to define the educational offer of the institute and the curricula models that must be respected by the curricula defined by the students. Once a curricula model is defined, it can be translated into a LTL formula by following the process described in Section 5.

For what concerns the interaction of the user with the validation system, there is the need of allowing the user to insert the curriculum to be validated into the system. The easiest way for allowing an a *naïve user*, like the average student, to perform this task is to ask him/her to enter a sequence of courses, corresponding to the desired curriculum, by using a web interface (as it was done, for instance, in the WLog system [13]). A linear plan is the simplest kind of curriculum that can be captured by means of an activity diagram, however, this choice makes sense because it is unlikely that the student learns to exploit the full potential of an activity diagram-based representation in order to express curricula. A good handling of activity diagrams requires, in fact, some expertise that the naïve user does not have. Moreover, since the planning PService produces linear plans, it is also possible to compose the effect of the planning process

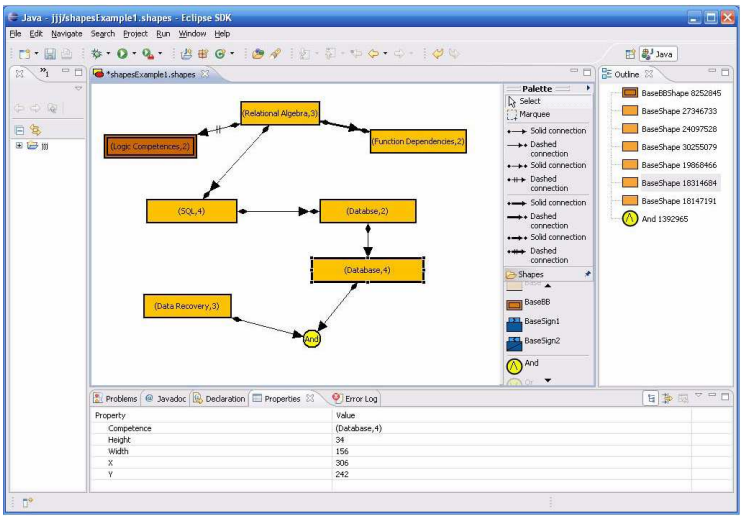


Fig. 12. A screenshot of the DCML designer.

with the validation service, in order to check the compliance of an automatically generated curriculum with a given curricula model.

The full potential of activity diagrams can, instead, be exploited by the instructional designer, when he/she faces the task of building and proposing new educational paths, which may, for instance, include a mandatory part and one or more (alternative) options. In this case, the interaction will not necessarily be performed via the browser. The designer can, in fact, write activity diagrams by means of standard UML design system.

Moreover, when we want to check that a curriculum does not show competence gaps and supplies the user's learning goal, it is required to interface the Validation PService with the RDF course descriptions and with the user profile. This can be done along the line of what we have described in the previous section by translating the information stored in the RDF files into Promela code.

7 Extending DCML to deal with time proximity

The constraints presented in Section 4 express temporal relations which do not capture the time proximity between the acquisitions of two competences. It is often the case, however, when by saying that a competence is to be acquired, for instance, before another competence, the designer actually means "immediately before", e.g. in the previous term. In order to express this stronger kind of relations we have extended the core of DCML by adding the notions of *immediateness*. All the relations that we have introduced (see Figure 4) have a correspondent stronger version: *immediate before*, *immediate implication*, *immediate succession* and their negations. Graphically, the notion of immediateness

is represented by triple arrows (see Figure 2). In the following we briefly introduce these new relations, whose definitions exploit the temporal logic operator *next-time*: $\bigcirc\varphi$ means that the formula φ holds in the next state of the run.

Immediate before *Immediate before* is represented by means of a triple line arrow that ends with a little-ball. 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)" $\wedge\Box((k_1, l_1) \supset (\bigcirc(k_2, l_2) \vee \Box\neg(k_2, l_2)))$. More generally, in presence of DNF formulas as antecedent and consequence of a "immediate before" relation, we have the following definition for "dnf_1 *immediate before* dnf_2":

$$\bigvee_{cf_i \in dnf_1, cf_j \in dnf_2} \text{“}cf_i \textit{ before } cf_j\text{”} \ \wedge \Box(cf_i \supset (\mathsf{next}(cf_j) \vee \mathsf{absence}(cf_j)))$$

where $\mathsf{next}(cf) = \bigwedge_{(k_i,l_i)\in cf} \bigcirc(k_i, l_i)$.

The *not immediate before* is translated exactly in the same way as the *not before*. Indeed, it is a special case because we assume that a competence cannot be forgotten. More generally, in presence of DNF formulas, "dnf_1 *not immediate before* dnf_2" is:

$$\bigvee_{cf_i \in dnf_1, cf_j \in dnf_2} \mathsf{negation}(cf_i) \ \mathsf{U} \ (cf_j \wedge \mathsf{negation}(cf_i))$$

Immediate implication The *immediate implication*, 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 afterwards. 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: $\Diamond(k_1, l_1) \supset \Diamond(k_2, l_2) \wedge \Box((k_1, l_1) \supset \bigcirc(k_2, l_2))$. More generally, in presence of DNF formulas as antecedent and consequence of a "immediate implication" relation, we have the following definition for "dnf_1 *immediate implication* dnf_2":

$$\bigvee_{cf_i \in dnf_1, cf_j \in dnf_2} \text{“}cf_i \textit{ implies } cf_j\text{”} \ \wedge \Box(cf_i \supset \mathsf{next}(cf_j))$$

"(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 implication. More generally, in presence of

DNF formulas, "dnf_1 *not immediate implies* dnf_2" is:

$$\bigvee_{cf_i \in dnf_1, cf_j \in dnf_2} \mathsf{existence}(cf_i) \supset \mathsf{absence}(cf_j)$$

Immediate succession 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)" $\wedge\Box((k_1, l_1) \supset \bigcirc(k_2, l_2))$. More generally, in presence of DNF formulas as antecedent and consequence of a "immediate succession" relation, we have the following definition for "dnf_1 *immediate succeeds* dnf_2":

$$\bigvee_{cf_i \in dnf_1, cf_j \in dnf_2} \text{“}cf_i \textit{ succession } cf_j\text{”} \ \wedge \Box(cf_i \supset \mathsf{next}(cf_j))$$

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)" $\vee\Diamond((k_1, l_1) \wedge \bigcirc\neg(k_2, l_2)))$. More generally, in presence of DNF formulas, "dnf_1 *not immediate succeeds* dnf_2" is:

$$\bigvee_{cf_i \in dnf_1, cf_j \in dnf_2} \mathsf{existence}(cf_i) \supset (\mathsf{absence}(cf_j) \vee \ \text{“}cf_i \textit{ not succeeds } cf_j\text{”} \vee \Diamond(cf_i \wedge \mathsf{next}(\mathsf{negation}(cf_j))))$$

8 Conclusions

This article integrates and extends the results of the collaboration, carried on in the last years between the Department of Computer Science at the University of Torino and the University of Hannover in the area of e-learning, within the Personalized Information Systems group of the Reasoning on the Web with Rules and Semantics European Network of Excellence (REWERSE) [8, 9, 11]. In this work, great attention has been posed on the issue of how to represent learning resources and curricula, and how to define curricula models. The chosen approach relies on the notion of "competence", thus allowing an abstract perspective, in which learning resources do not directly depend on one another but rather have knowledge prerequisites. Based on competences it is also possible to represent constraints that a curriculum must respect. To this aim, we have identified a set of useful constraints and defined DCML, a graphical language for designing curricula. As described in the previous sections, DCML allows the representation of temporal constraints posed on the acquisition of competences (supplied by courses), taking into account both the concepts supplied/required and the proficiency level. The language has a grounding in Linear Temporal Logic and, hence, allows the application of various forms of reasoning in order to execute tasks like: the verification that a curriculum does not have competence gaps, the verification that a curriculum allows the acquisition of some knowledge of interest for

the user, the verification that a curriculum satisfies the constraints contained in a curricula model. We have shown how model checking techniques can be used to execute all these verification tasks. This use of model checking is inspired by [47], 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 [46], 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 [46], 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 [14, 9, 13], 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, did not allow to tackle with curricula models and to the related verification tasks.

We have also reported about the integration of the above approach into the Personal Reader Framework [34]. In this latter context, the goal of personalization is to give support in the process of defining sequences of courses that fit the specific context and the learning goal of individual students. Despite some manual post-processing for fixing inconsistencies, we used real information from the Hannover University database of courses for extracting the meta-data. Currently the courses are annotated also by meta-data concerning the *schedule*, given in terms of *semesters*, and location of courses, like for instance room-numbers, addresses and teaching hours. This information is not used by the implemented services yet but it would be interesting to develop, as future work, also other services, which, along the line of [49], complete the curricula planning process taking into account also these parameters.

This basic representation, given in terms of competences, can also easily be enriched by adding in the precondition of learning resources *media-related* information and *accessibility* information, expressing constraints on how the resource is to be played/used. This information differs, in quality, from the other kind of precondition mentioned before, in fact, while the former only concerns the user's knowledge and captures a constraint on the possibility of using the resource *with profit* given the current (supposed) knowledge of the user, the latter concerns some either environmental or physical aspect concerning the ability of *using* the resource. So, for instance, a visually-impaired user cannot watch an explanatory animation. The information about the need of the ability of watching could be associated, as a precondition, to the learning resource (actually, to all learning resources that are animations). On the other hand, the information about accessibility constraints for a user could be obtained directly from the user profile.

Thus, all this information could be used for filtering out resources that does not fit the user's abilities, before to proceed in the composition of a curriculum by using the automatic support of our reasoning services.

The Personal Reader Platform provides a natural framework for implementing a service-oriented approach to personalization in the Semantic Web, allowing to investigate how (semantic) web service technologies can provide a suitable infrastructure for building personalization applications, that consist of re-usable and interoperable personalization functionalities. The idea of taking a service oriented approach to personalization is quite new and was born within the personalization working group of the Network of Excellence REWERSE. The adoption of a service-oriented architecture per the Personal Reader makes the introduction of new personalization functionalities very easy: it is, in fact possible to adopt a separation of concerns approach and develop services which handle a given kind of knowledge/constraints and, depending on the repository of learning resources and on the user's desires, it is possible to combine different services in order to combine their functionalities and perform more complex personalization tasks.

As future work, it will also be interesting to complement the architecture by integrating Web 2.0 features and develop personalization functionalities for handling information extracted from the activity of the community of users. This seems a very promising direction for recommendation purposes. For instance, when a user is uncertain on which curriculum to choose between two offers that are equivalent from the point of view of the supplied knowledge, he/she could be recommended to select one of the two on the basis of the behavior of other members in the community that we know as being "friends" of the current user.

Acknowledgements

The authors would like to thank Giuseppe Berio 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 by DAAD and Ateneo Italo-Tedesco though the Vigoni German and Italian researchers exchange Program 2007-2008.

References

1. Unified Modeling Language: Superstructure, version 2.1.1. OMG, Object Management Group, February 2007.
2. F. Abel, I. Brunkhorst, N. Henze, D. Krause, K. Mushtaq, P. Nasirifar, and K. Tomaschweski. Personal reader agent: Personalized access to configurable web services. Technical report, Distributed Systems Institute, Semantic Web Group, University of Hannover, 2006.
3. G. Antoniou, A. Bikakis, N. Dimaresis, M. Genetzakis, G. Georgalis, G. Governatori, E. Karouzaki, N. Kazepis, D. Kosmadakis, M. Kritsotakis, G. Lilis, A. Papadogiannakis, P. Pediaditis, C. Terzakis, R. Theodosaki, and D. Zeginis. Proof explanation for the semantic web using defeasible logic. In Z.Zhang and J. H.

Siekmann, editors, *Proc. of 2nd Int. Conf. on Knowledge Science, Engineering and Management, KSEM 2007*, volume 4798 of *LNCS*, pages 186–197. Springer, 2007.
4. Grigoris Antoniou, Uwe Aßmann, Cristina Baroglio, Stefan Decker, Nicola Henze, Paula-Lavinia Patranjan, and Robert Tolksdorf, editors. *Reasoning Web, Third International Summer School 2007, Dresden, Germany, September 3-7, 2007, Tutorial Lectures*, volume 4636 of *Lecture Notes in Computer Science*. Springer, 2007.
5. Grigoris Antoniou, Enrico Franconi, and Frank van Harmelen. Introduction to semantic web ontology languages. In Norbert Eisinger and Jan Maluszynski, editors, *Reasoning Web*, volume 3564 of *Lecture Notes in Computer Science*, pages 1–21. Springer, 2005.
6. Grigoris Antoniou and Frank van Harmelen. *A Semantic Web Primer*. the MIT Press, 2004.
7. M. Baldoni, C. Baroglio, I. Brunkhorst, N. Henze, E. Marengo, and V. Patti. A Personalization Service for Curriculum Planning. In E. Herder and D. Heckmann, editors, *Proc. of the 14th Workshop on Adaptivity and User Modeling in Interactive Systems, ABIS 2006*, pages 17–20, Hildesheim, Germany, October 2006.
8. M. Baldoni, C. Baroglio, I. Brunkhorst, E. Marengo, and V. Patti. Reasoning-based Curriculum Sequencing and Validation: Integration in a Service-Oriented Architecture. In E. Duval and R. Klamma, editors, *Proc. of EC-TEL 2007 - 2nd Europ. Conf. on Technology Enhanced Learning*, volume 4756 of *LNCS*, pages 426–431. Springer, 2007.
9. M. Baldoni, C. Baroglio, and N. Henze. Personalization for the Semantic Web. In N. Eisinger and J. Maluszynski, editors, *Reasoning Web, First International REWERSE Summer School 2005*, volume 3564 of *LNCS Tutorials*, pages 173–212. Springer-Verlag, Malta, July 2005.
10. M. Baldoni, C. Baroglio, N. Henze, and V. Patti. Setting up a framework for comparing adaptive educational hypermedia: First steps and application on curriculum sequencing. In N. Henze, editor, *Proc. of ABIS-Workshop 2002: Personalization for the mobile World, Workshop on Adaptivity and User Modeling in Interative Software Systems*, pages 43–50, Hannover, Germany, October 2002.
11. M. Baldoni, C. Baroglio, and E. Marengo. Curricula modeling and checking. In *Proc. of the 10th Congress of the Italian Association for Artificial Intelligence, AI*IA 2007*, volume 4733 of *LNCS*, pages 471–482, 2007.
12. 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 M. Bouzid and N. Henze, editors, *Proc. of the Semantic Web Personalization Workshop, held in conjuction with the 3rd European Semantic Web Conference*, pages 53–62, Budva, Montenegro, 2006.
13. 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.
14. M. Baldoni, C. Baroglio, V. Patti, and L. Torasso. Reasoning about learning object metadata for adapting SCORM courseware. In L. Aroyo and C. Tasso, editors, *AH 2004: Workshop Proceedings, Part I, International Workshop on Engineering the Adaptive Web, EAW'04: Methods and Technologies for personalization and Adaptation in the Semantic Web*, pages 4–13, Eindhoven, The Netherlands, August 2004. Technische Universiteit Eindhoven.
15. R. Baumgartner, S. Flesca, and G. Gottlob. Visual web information extraction with lixto. In Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi,

Kotagiri Ramamohanarao, and Richard T. Snodgrass, editors, *VLDB*, pages 119–128. Morgan Kaufmann, 2001.
16. R. Baumgartner, N. Henze, and M. Herzog. The personal publication reader: Illustrating web data extraction, personalization and reasoning for the semantic web. In *ESWC*, pages 515–530, 2005.
17. Tim Berners-Lee. Semantic web - keynote at xml 2000 conference, 2000. http://www.w3.org/2000/Talks/1206-xml2k-tbl/slide10-0.html.
18. Tim Berners-Lee, Jim Hendler, and Ora Lassila. The semantic web. *Scientific American*, May 2001.
19. P. Brusilovsky. Course sequencing for static courses? applying ITS techniques in large-scale web-based education. *Intelligent tutoring systems*, pages 625–634, 2000.
20. 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.
21. C. Castro and S. Manzano. Variable and value ordering when solving balanced academic curriculum problems. In *Proc. of the 6th Workshop of ERCIM WG on Constraints*, 2001.
22. O. E. M. Clarke and D. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 2001.
23. Learning Technology Standards Committee. Ieee standard for learning object metadata, 2002. IEEE Standard 1484.12.1, Institute of Electrical and Electronics Engineers.
24. J. L. De Coi, E. Herder, A. Koesling, C. Lofi, D. Olmedilla, O. Papapetrou, and W. Sibershi. A model for competence gap analysis. In *WEBIST 2007, Proceedings of the Third International Conference on Web Information Systems and Technologies: Internet Technology / Web Interface and Applications*, Barcelona, Spain, Mar 2007. INSTICC Press.
25. E. A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science*, volume B, pages 997–1072. Elsevier, 1990.
26. European Commission, Education and Training. The Bologna process. http://europa.eu.int/comm/education/policies/educ/bologna/bologna_en.html.
27. R. Farrell, S. D. Liburd, and J. C. Thomas. Dynamic assembly of learning objects. In *Proc. of WWW 2004*, 2004.
28. M. Fisher, D. Gabbay, and L. Vila. *Handbook of Temporal Reasoning in Artificial Intelligence (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, USA, 2005.
29. 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.
30. A. Garro, N. Leone, and F. Ricca. Logic Based Agents for E-learning. In *Proc. of the IJCAI'03 Workshop on Knowledge Representation and Automated Reasoning for E-Learning Systems*, pages 36–45, 2003.
31. A. Garro, L. Palopoli, and F. Ricca. Exploiting agents in e-learning and skills management context. *AI Commun.*, 19(2):137–154, 2006.
32. N. Guelfi and A. Mammar. A Formal Semantics of Timed Activity Diagrams and its PROMELA Translation. In *Proc. of the 12th Asia-Pacific Software Engineering Conference (APSEC'05)*, pages 283–290. IEEE Computer Society, 2005.
33. N. Henze. Personal readers: Personalized learning object readers for the semantic web. In *12th International Conference on Artificial Intelligence in Education, AIED05*, Amsterdam, The Netherlands, 2005.

34. N. Henze and D. Krause. Personalized access to web services in the semantic web. In *The 3rd International Semantic Web User Interaction Workshop (SWUI, collocated with ISWC 2006*, November 2006.
35. N. Henze and W. Nejdl. Adaptation in open corpus hypermedia. *IJAIED Special Issue on Adaptive and Intelligent Web-Based Systems*, 12:325–350, 2001.
36. G. J. Holzmann. *The SPIN Model Checker, Primer and Reference Manual*. Addison-Wesley, 2003.
37. T. Lambert, C. Castro, E. Monfroy, and F. Saubion. Solving the balanced academic curriculum problem with an hybridization of genetic algorithm and constraint propagation. In L. Rutkowski, R. Tadeusiewicz, L. A. Zadeh, and J. Zurada, editors, *Artificial Intelligence and Soft Computing, ICAISC 2006*, volume 4029 of *LNCS*, pages 410–419. Springer, 2006.
38. M. Melia and C. Pahl. Automatic Validation of Learning Object Compositions. In *Information Technology and Telecommunications Conference IT&T'2005: Doctoral Symposium*, Carlow, Ireland, 2006.
39. P. Mohan and C. Brooks. Learning Object on the Semantic Web. In *Proc. of the 3rd IEEE International Conference onAdvanced Learning Technologies*, Athens, Greece, 2003.
40. L. Pazzi. Three points of view in the characterization of complex entities. In N. Guarino, editor, *Formal Ontology in Information Systems*. IOS Press, 1998.
41. O. Pistamiglio. Sviluppo di un plugin grafico per eclipse: Dcml designer. Laurea specialistica in informatica, Corso di Studi in Informatica, Università degli Studi di Torino, 2007. Relatore: Prof. M. Baldoni.
42. Stuart Russell and Peter Norvig. *Artificial Intelligence, A Modern Approach, Second Edition*. Prentice Hall Series in Artificial Intelligence, 2003.
43. M. P. Singh. Agent communication languages: Rethinking the principles. *IEEE Computer*, 31(12):40–47, 1998.
44. 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.
45. L. Stojanovic, S. Staab, and R. Studer. eLearning based on the Semantic Web. In *WebNet2001 - World Conference on the WWW and Internet*, Orlando, Florida, USA, 2001.
46. 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: evidence-based guidelines and protocols*, Riva del Garda, August 2006.
47. W. M. P. van der Aalst and M. Pesic. DecSerFlow: Towards a Truly Declarative Service Flow Language. In Mario Bravetti and Gialuigi Zavattaro, editors, *Proc. of WS-FM*, LNCS, Vienna, September 2006. Springer.
48. J. Vassileva. Dynamic CAL-Courseware Generation Within an ITS-Shell Architecture. In I. Tomek, editor, *Computer Assisted Learning*, volume 601 of *LNCS*, pages 581–591. Springer, 1992.
49. K. Wu and W. S. Havens. Modelling an academic curriculum plan as a mixed-initiative constraint satisfaction problem. In B. Kégl and G. Lapalme, editors, *Canadian Conference on AI*, volume 3501 of *LNCS*, pages 79–90, 2005.

M. Baldoni, C. Baroglio, G. Berio, and E. Marengo. Declarative representation of curricula models: an LTL- and UML-based approach. In M. Baldoni, A. Boccalatte, F. De Paoli, M. Martelli, and V. Mascardi, editors, Proc. of WOA 2007: Dagli oggetti agli agenti, Agenti e Industrie: Applicazioni tecnologiche degli agenti software, pages 34-41, Genova, Italy, September 2007. Seneca Edizioni.

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 (α U β, 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

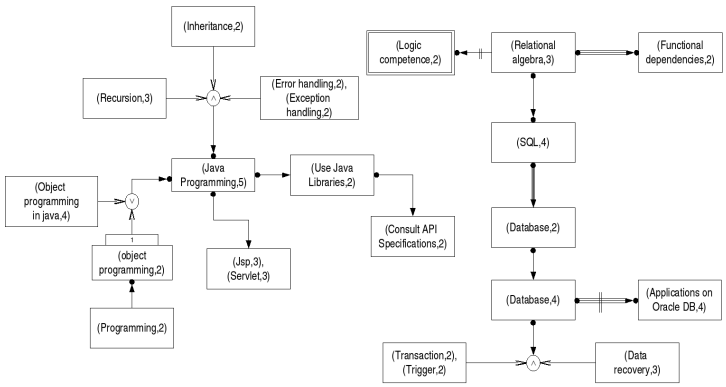


Fig. 1. An example of curricula model in DCML.

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

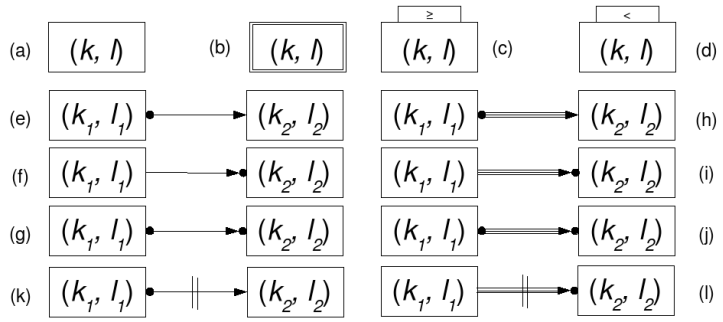


Fig. 2. Competences (a) and basic constraints (b), (c), and (d). Relations among competences: (e) implication, (f) before, (g) succession, (h) immediate implication, (i) immediate before, (j) immediate succession, (k) not implication, (l) not immediate before.

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

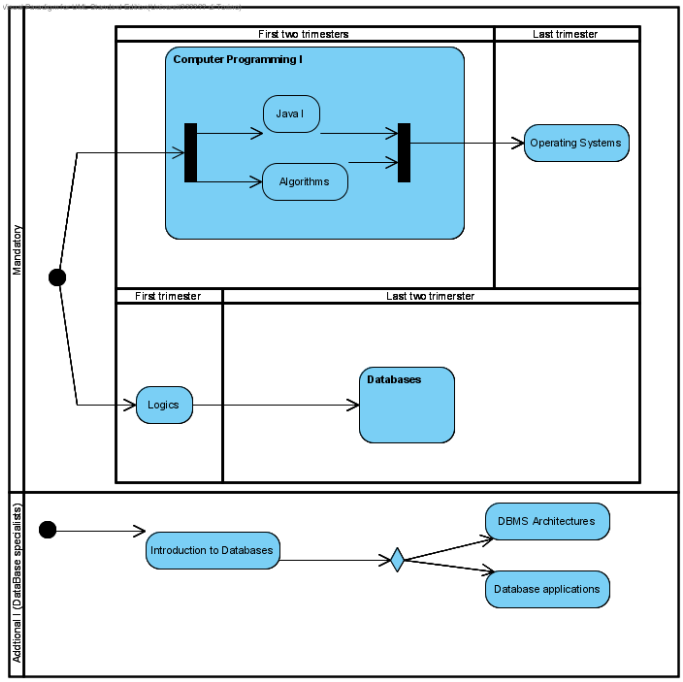


Fig. 3. Activity diagram representing a curriculum with mandatory and additional, student chosen, courses. Swimlanes represent the sequencings of courses. Vertical divisions capture the different milestones (trimesters).

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* $cf_j \vee \Diamond(cf_i \wedge \mathsf{next}(\mathsf{negation}(cf_j))))$.

M. Baldoni, C. Baroglio, and E. Marengo. Curricula Modeling and Checking. In R. Basili and M. T. Pazienza, editors, Proc. of AI*IA 2007: Advances in Artificial Intelligence, 10th Congress of the Italian Association for Artificial Intelligence, volume 4733 of LNAI, pages 471-482, Rome, Italy, September 2007. Springer.

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

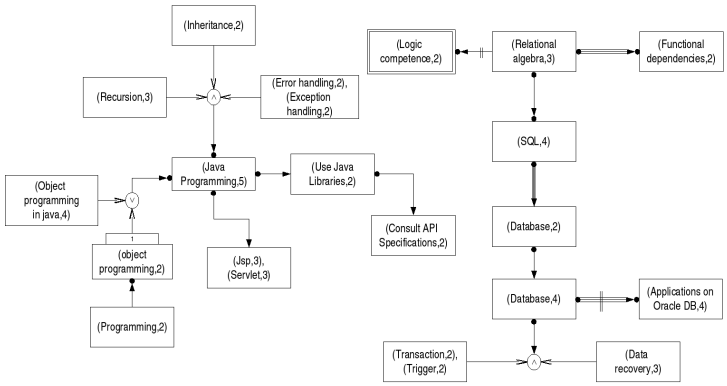


Fig. 1. An example of curricula model in DCML

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*.

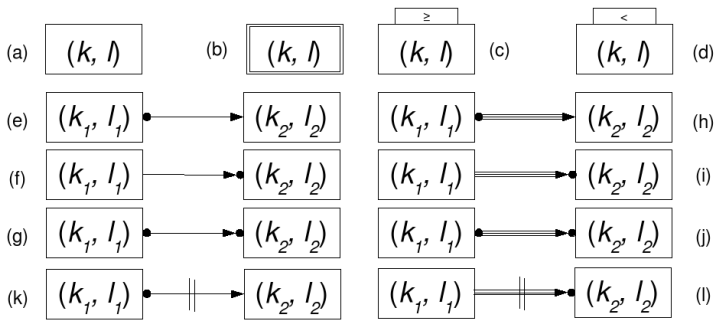


Fig. 2. Competences (a) and basic constraints (b), (c), and (d). Relations among competences: (a) implication, (b) before, (c) succession, (d) immediate implication, (e) immediate before, (f) immediate succession, (g) not implication, (h) not immediate before.

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

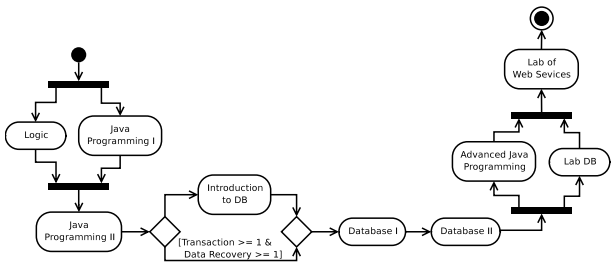


Fig. 3. Activity diagram representing a set of eight different curricula. Notice that *Logic* and *Java Programming I* can be attended in any order (even in parallel), as well as *Advanced Java Programming* and *Lab DB*, while *Introduction to DB* will be considered only if the guard *Transaction* and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

M. Baldoni, C. Baroglio, I. Brunkhorst, E. Marengo, and Patti. A Service- Oriented Approach for Curriculum Planning and Validation. In Proceedings of the Multi- Agent Logics, Languages, and Organisations, Federated Workshops, MALLOW'007, Agent, Web Services and Ontologies, Integrated Methodologies, MALLOW- AWESOME'007, Durham, GB, September 2007.

A Service-Oriented Approach for Curriculum Planning and Validation

Matteo Baldoni[1], Cristina Baroglio[1], Ingo Brunkhorst[2], Elisa Marengo[1], Viviana Patti[1]

[1] Dipartimento di Informatica — Università degli Studi di Torino
c.so Svizzera, 185, I-10149 Torino (Italy)
{baldoni,baroglio,patti}@di.unito.it, elisa.mrng@gmail.com
[2] L3S Research Center, University of Hannover
D-30539 Hannover, Germany
brunkhorst@l3s.de

Abstract. We present a service-oriented personalization system, set in an educational framework, based on a semantic annotation of courses, given at a knowledge level (what the course teaches, what is requested to know for attending it in a profitable way). The system supports users in building personalized curricula, formalized by means of an action theory. It is also possible to verify the compliance of curricula w.r.t. a model, expressing constraints at a knowledge level. For what concerns the first task, classical planning techniques are adopted, which take into account both the student's initial knowledge and her learning goal. Instead, curricula validation is done against a model, formalized as a set of temporal constraints. We have developed a prototype of the planning and validation services, by using -as reasoning engines- SWI-Prolog and the SPIN model checker. Such services will be supplied and combined as plug-and-play personalization services in the Personal Reader framework.

1 Introduction and Motivation

The birth of the Semantic Web brought along standard models, languages, and tools for representing and dealing with machine-interpretable semantic descriptions of Web resources, by giving a strong new impulse to research on personalization. The introduction of machine-processable semantics makes the use of a variety of reasoning techniques for implementing personalization functionalities possible, widening the range of the forms that personalization can assume. So far, reasoning in the Semantic Web is mostly reasoning about knowledge expressed in some ontology. However personalization may involve also other kinds of reasoning and knowledge representation, that conceptually lie at the logic and proof layers of the Semantic Web tower.

Moreover, the next Web generation promises to deliver Semantic Web Services, that can be retrieved and *combined* in a way that satisfies the user. It opens the way to many forms of *service-oriented personalization*. Web services provide an ideal infrastructure for enabling *interoperability* among personalization applications and for constructing Plug&Play-like environments, where the user can

select and combine the kinds of services he or she prefers. Personalization can be obtained by taking different approaches, e.g. by developing services that offer personalization functionalities as well as by personalizing the way in which services are selected, and *composed* in order to meet specific user's requirements.

In the last years we carried on a research in the educational domain, by focussing on *semantic web* representations of learning resources and on *automated reasoning* techniques for enabling different and complementary personalization functionalities, e.g. curriculum sequencing [6, 7] and verification of the compliance of a curriculum against some course design goals [5]. Our current aim is to implement such results in an organic system, where different personalization services, that exploit semantic web reasoning, can be combined to support the user in the task of building a curriculum, based on *learning resources* that represent courses.

While in early times learning resources were simply considered as "contents", strictly tied to the platform used for accessing them, recently, greater and greater attention has been posed on the issue of *re-use* and of a *cross-platform* use of educational contents. The proposed solution is to adopt a *semantic annotation* of contents based on standard languages, e.g. RDF and LOM. Hereafter, we will consider a *learning resource* as formed by *educational contents* plus *semantic meta-data*, which supply information on the resources at a *knowledge level*, i.e. on the basis of concepts taken from an ontology that describes the educational domain. In particular we rely on the interpretation of learning resources as *actions* discussed in [6, 7]: the meta-data captures the *learning objectives* of the learning resource and its *pre-requisites*. By doing so,one can rely on a classical theory of actions and apply different reasoning methods -like *planning*- for building personalized curricula [6, 7]. The modeling of learning resources as actions also enables the use of model checking techniques for developing a validation service that detects if a user-given curriculum is compliant w.r.t an abstract model, given as a set of constraints. In the following we present our achievements in the implementation of a Planning service and a Validation service that can interoperate within the Personal Reader Framework [18].

Curriculum planning and validation offer a useful support in many practical contexts and can be fruitfully combined for helping students or teaching institutions. Often a student knows what competency he/she would like to acquire but has no knowledge of which courses will help him/her acquiring it. Moreover, taking courses at different Universities is becoming more and more common in Europe. As a consequence, building a curriculum might become a complicated task for students, who must deal with an enormous set of courses across the European countries, each described in different languages and on the basis of different keywords.

The need of personalizing the sequencing of learning resource, w.r.t. the student's interests and context, has often to be *combined* with the ability to check that the resulting curriculum *complies* against some abstract *curricula specification*, which encodes the *curricula-design goals* expressed by the teachers or by the institution offering the courses. Consider a student, who wants to build

a valid curriculum with the support of our automatic system. The student can either use as a basis the suggestion returned by the system or he/she can design the curriculum by hand, based on own criteria. In both cases a personalized curriculum is obtained and can be given in input to the validation service for checking the compliance against a curricula model. Curricula models specify general rules for building learning paths and can be interpreted as constraints designed by the University for guaranteeing the achievement of certain learning goals. These constraints are to be expressed in terms of knowledge elements, and maybe also on features that characterize the resources.

Consider now a university which needs to certify that the specific curricula, that it offers for achieving a certain educational goal, and that are built upon the courses offered locally by the university itself, respect some European guidelines. In this case, we could, in fact, define the guidelines as a set of constraints at an abstract level, i.e. as relations among a set of competencies which should be offered in a way that meets some given scheme. At this point the verification could be performed automatically, by means of a proper reasoner. Finally, the automatic checking of compliance combined with curriculum planning could be used for implementing processes like cooperation among institutes in curricula design and integration, which are actually the focus of the so called *Bologna Process* [15], promoted by the EU.

While SCORM [2] and Learning Design [19, 20] represent the most important steps in the direction of managing and using e-learning based courses and workflows among a group of actors participating in learning activities, most of the available tools lack the machine-interpretable information about the learning resources, and as a result they are not yet open for reasoning-based personalization and automatic composition and verification. Given our requirements, it is a natural choice to settle our implementation in the Personal Reader (PR) framework. The PR relies on a service-oriented architecture enabling personalization, via the use of semantic *Personalization Services*. Each service offers a different personalization functionality, e.g. recommendations tailored to the needs of specific users, pointers to related (or interesting or more detailed/general) information, and so on. These semantic web services communicate solely based on RDF documents.

The paper is organized as follows. Section 2 describes our approach to the representation and reasoning about learning resources, curricula, and curricula models. The implementation of the two services and their integration into the PR Framework is discussed in section 3. We finish with conclusions and hints on future work in Section 4.

2 Curricula representation and reasoning

Let us begin with the introduction of our approach to the representation of learning resources, curricula, and curricula models. The basic idea is to describe all the different kinds of objects, that we need to tackle and that we will introduce hereafter, on the basis of a set of predefined *competencies*, i.e. terms identifying

specific *knowledge elements*. We will use the two terms as synonyms. Competencies can be thought of, and implemented, as concepts in a shared ontology. In particular, for what concerns the application system described here, competencies were extracted by means of a semi-automatic process and stored as an RDF file (see Section 3.1 for details).

Given a predefined set of competencies, the initial knowledge of a student can be represented as a set of such concepts. This set changes, typically it grows, as the student studies and learns. In the same way, a user, who accesses a repository of learning resources, does it with the aim of finding materials that will allow him/her to acquire some knowledge of interest. Also this knowledge, that we identify by the term *learning goal*, can be represented as a set of knowledge elements. The learning goal is to be taken into account in a variety of tasks. For instance, the construction of a personalized curriculum is, actually, the construction of a curriculum which allows the achievement of a learning goal expressed by the user. In Section 3 we will describe a *curricula planning service* for accomplishing this task.

2.1 Learning resources and curricula

A *curriculum* is a sequence of *learning resources* that are homogeneous in their representation. Based on work in [6, 7], we rely on an *action theory*, and take the abstraction of resources as *simple actions*. More specifically, a learning resource is modelled as an action for acquiring some competencies (called *effects*). In order to understand the contents supplied by a learning resource, the user is sometimes required to own other competencies, that we call *preconditions*. Both preconditions and effects can be expressed by means of a *semantic annotation* of the learning resource [7]. In the following we will often refer to learning resources as "courses" due to the particular application domain that we have considered (university curricula).

As a simple example of "learning resource as action", let us, then, report the possible representation (in a classical STRIPS-like notation) of the course "databases for biotechnologies" (*db_for_biotech* for short):

ACTION: db_for_biothec(),
PREREQ: relational_db, EFFECTS: scientific_db

The prequisites to this action is to have knowledge about *relational databases*. Its effect is to supply knowledge about *scientific databases*.

Given the above interpretation of learning resources, a *curriculum* can be interpreted as a *plan*, i.e. as a sequence of actions, whose execution causes transitions from a state to another, until some final state is reached. The *initial state* contains all the competences that we suppose available before the curriculum is taken, e.g. the knowledge that the student already has. This set can also be empty. The *final state* is sometimes required to contain specific knowledge elements, for instance, all those that compose the user's learning goal. Indeed, often curricula are designed so to allow the achievement of a well-defined *learning goal*.

A transition between two states is due to the application of the action corresponding to a learning resource. Of course, for an action to be applicable, its preconditions must hold in the state to which it should be applied. The application of the action consists in an *update* of the state. We assume that competences can only be added to states. Formally, we assume that the domain is monotonic. The intuition behind this assumption is that the act of using a new resource will never erase from the students' memory the concepts acquired insofar. Knowledge grows incrementally.

2.2 Curricula models

Curricula models consist in sets of constraints that specify desired properties of curricula. Curricula models are to be defined on the basis of knowledge elements as well as of learning resources (courses). In particular, we would like to restrict the set of possible sequences of resources corresponding to curricula. This will be done by imposing constraints on the *order* by which knowledge elements are added to the states (e.g. "a knowledge element α is to be acquired before a knowledge element β"), or by specifying some *educational objectives* to be achieved, in terms of knowledge that must be contained in the final state (e.g. "a knowledge element α must be acquired sooner or later"). Therefore, we represent a curricula model as a set of *temporal constraints*. Being defined on knowledge elements, a curricula model is *independent* from the specific resources that are taken into account, for this reason, it can be *reused* in different contexts and it is suitable to open and dynamic environments like the web.

The possibility of *verifying the compliance of curricula to models* is extremely important in many applicative contexts, as explained by examples in the introduction. In some cases these checks could be integrated into the curriculum construction process; nevertheless, it is important to be able to perform the verification independently from the construction process. Let us consider again our simple scenario concerning a university, which offers a set of curricula that are proved to satisfy the guidelines given by the EU for a certain year. After a few years, the EU guidelines change: our University has the need to check if the curricula that it offers, still satisfy the guidelines, without rebuilding them.

A natural choice for representing temporal constraints on action paths is linear-time temporal logic (LTL) [14]. This kind of logic allows to verify if a property of interest is true for all the possible executions of a model (in our case the specific curriculum). This is often done by means of model checking techniques [12].

The curricula as we represent them are, actually, Kripke structures. Briefly, a Kripke structure identifies a set of states with a transition relation that allows passing from a state to another. In our case, the states contain the knowledge items that are owned at a certain moment. Since the domain is monotonic (as explained above we can assume that knowledge only grows), states will always contain *all* the competencies acquired up to that moment. The transition relation is given by the actions that are contained in the curriculum that is being checked.

It is possible to use the LTL logic to verify if a given formula holds starting from a state or if it holds for a set of states.

For example, in order to specify in the curricula model constraints on *what* to achieve, we can use the formula $\Diamond\alpha$, where $\Diamond$ is the eventually operator. Intuitively, such a formula expresses the fact that a set of knowledge elements will be acquired sooner or later. Moreover, constraints concerning *how* to achieve the educational objectives, such as "a knowledge element β cannot be acquired before the knowledge element α is acquired", can, for instance, be expressed by the LTL temporal formula $\neg\beta\ U\ \alpha$, where U is the *weak until* operator. Given a set of knowledge elements to be acquired, such constraints specify a partial ordering of the same elements.

2.3 Planning and Validation

Given a semantic annotation with preconditions and effects of the courses, classical planning techniques are exploited for creating *personalized curricula*, in the spirit of the work in [6, 7]. Intuitively the idea is that, given a repository of learning resources, which have been semantically annotated as described, the user expresses a *learning goal* as a set of *knowledge elements* he/she would like to acquire, and possibly also a set of already owned competencies. Then, the system applies planning to build a sequence of learning resources that, read in sequence, will allow him/her to achieve the goal.

The particular planning methodology that we implemented (see Section 3.3 for details) is a simple *depth-first forward planning* (an early prototype was presented in [3]), where actions cannot be applied more than once. The algorithm is simple:

1. Starting from the initial state, the set of *applicable* actions (those whose preconditions are contained in the current state) is identified.
2. One of such actions is selected and its application is simulated leading to a new state.
3. The new state is obtained by adding to the previous one the competencies supplied as effects of the selected action.
4. The procedure is repeated until either the goal is reached or a state is reached, in which no action can be applied and the learning goal is not satisfied.
5. In the latter situation, backtracking is applied to look for another solution.

The procedure will eventually end because the set of possible actions is finite and each is applied at most once. If the goal is achieved, the sequence of actions that label the transitions leading from the initial to the final state is returned as the resulting *curriculum*. If desired, the backtracking mechanism allows to collect a set of alternative solutions to present to the user.

Besides the capability of automatically building personalized curricula, it is also interesting to perform a set of verification tasks on curricula and curricula models. The simplest form of verification consists in *checking the soundness* of

curricula which are built by hand by users themselves, reflecting their own personal interests and needs. Of course, not all sequences which can be built starting from a set of learning resources are lawful. Learning dependencies, imposed by courses themselves in terms of preconditions and effects, must be respected. In other words, a course can appear at a certain point in a sequence only if it is *applicable* at that point, therefore, there are no *competency gaps*. These implicit "applicability constraints" capture precedences and dependencies that are innate to the nature of the taught concepts. In particular, it is important to verify that all the *competencies*, that are necessary to fully understand the contents, offered by a learning resource, are introduced or available before that learning resource is accessed. Usually, this verification, as stated in [13], is performed manually by the learning designer, with hardly any guidelines or support.

Given the interpretation of resources as actions, the verification of the *soundness of a curriculum*, w.r.t. the learning dependencies and the learning goal, can be interpreted as an *executability check* of the curriculum. Also in this case, the algorithm is simple:

1. Given an initial state, representing the knowledge available before the curriculum is attended, a simulation is executed, in which all the actions in the curriculum are (virtually) executed one after the other.
2. An action (representing a course) can be executed only if the current state contains all the concepts that are in the course precondition. Intuitively, it will be applied only if the student owns the notions that are required for understanding the topics of the course.
3. If, at a certain point, an action that should be applied is *not applicable* because some precondition does not hold, the verification fails and the reasons of such failure can be reported to the user.
4. Given that all the courses in the sequence can be applied, one after the other, the final state that is reached must be compared with the learning goal of the student: all the desired goal concepts must be achieved, so the corresponding knowledge elements must be contained in the final state.

This latter task actually corresponds to another basic form of verification, i.e. to check whether a (possibly hand-made) curriculum allows the *achievement of the desired learning goal*. These forms of basic verifications can be accomplished by the service described in Section 3.4.

Another interesting verification task consists in checking if a *personalized curriculum is valid w.r.t. a particular curricula model* or, following Brusilovski's terminology, checking if the curriculum is *compliant against the course design goals* [11]. Indeed, a personalized curriculum that is proved to be executable, cannot automatically be considered as being *valid* w.r.t. a particular *curricula model*. A curricula model, in fact, imposes further constraints on *what* to achieve and *how* achieving it. We will return to this kind of verification in Section 3.4.

3 Implementation in the Personal Reader Framework

The Personal Reader Framework has been developed with the aim of offering a uniform entry point for accessing the Semantic Web, and in particular Semantic Web Services. Indeed it offers an environment for designing, implementing and realizing Web content readers in a service-oriented approach, for a more detailed description, see [18] (`http://www.personal-reader.de/`).

In applications based on the Personal Reader Framework, a user can select and combine —plug together— which personalized support he or she wants to receive. The framework has already been used for developing Web Content Readers that present online material in an embedded context [10, 1, 17]. Besides

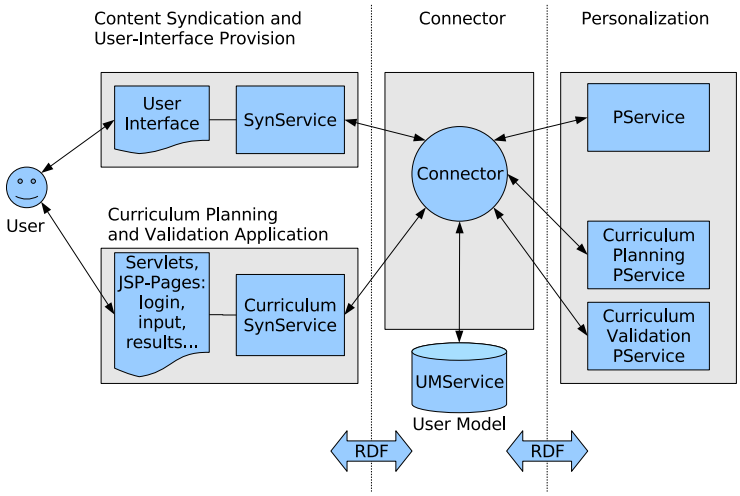


Fig. 1. Personal Reader Framework Overview

a user-interface, as shown in figure 1, a Personal Reader application consists of three types of *services*. *Personalization services* (PService) provide personalization functionalities: they deliver personalized recommendations for content, as requested by the user and obtained or extracted from the Semantic Web. *Syndication Services* (SynService) allow for some interoperability with the other services in the framework, e.g. for the discovery of the applications interfaces by a portal. The *Connector* is a single central instance responsible for all the communication between user interface and personalization services. It selects services based on their semantic description and on the requirements by the SynService. The Connector protects –by means of a public-key-infrastructure (PKI)– the communication among the involved parties. It also supports the customization and invocation of services and interacts with a user modelling service, called the *UMService*, which maintains a central user model.

3.1 Metadata Description of Courses

In order to create the corpus of courses, we started with information collected from an existing database of courses. We used the Lixto [9] tool to extract the needed data from the web-pages provided by the HIS-LSF (http://www.his.de/) system of the University of Hannover. This approach was chosen based on our experience with Lixto in the *Personal Publication Reader* [10] project, where we used Lixto for creating the publications database by crawling the publication pages of the project partners. The effort to adapt our existing tool for the new data source was only small. From the extracted metadata we created an RDF document, containing course names, course catalog identifier, semester, number of credit points, effects and preconditions, and the type of course, e.g. laboratory, seminar or regular course with examinations in the end, as illustrated in Figure 2.

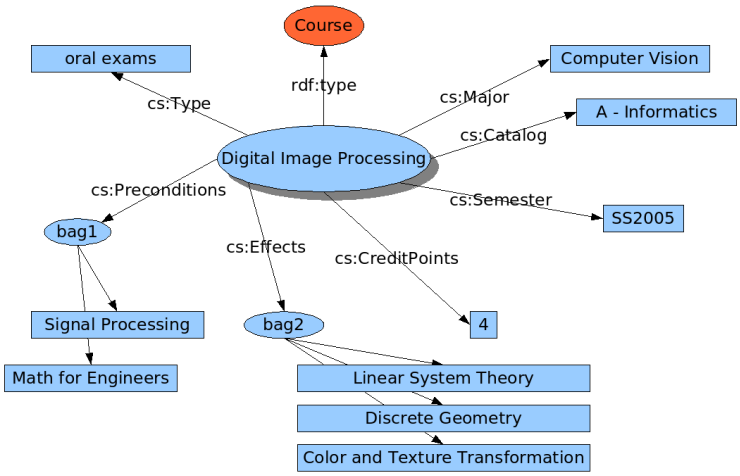


Fig. 2. An annotated course from the Hannover course database

The larger problem was that the quality of most of the information in the database turned out to be insufficient, mostly due to inconsistencies in the description of prerequisites and effects of the courses. Additionally the corpus was not annotated using a common set of terms, but authors and department secretaries used a slightly varying vocabulary for each of their course descriptions, instead of relying on a common classification system, like e.g. the ACM CCS for computer science.

As a consequence, we focussed only on a subset of the courses (computer science and engineering courses), and manually post-processed the data. Courses are annotated with prerequisites and effects, that can be seen as knowledge concepts or competences, i.e. ontology terms. After automatic extraction of effects

and preconditions, the collected terms were translated into proper English language, synonyms were removed and annotations were corrected where necessary. The resulting corpus had a total of 65 courses left, with 390 effects and 146 preconditions.

3.2 The User Interface and Syndication Service

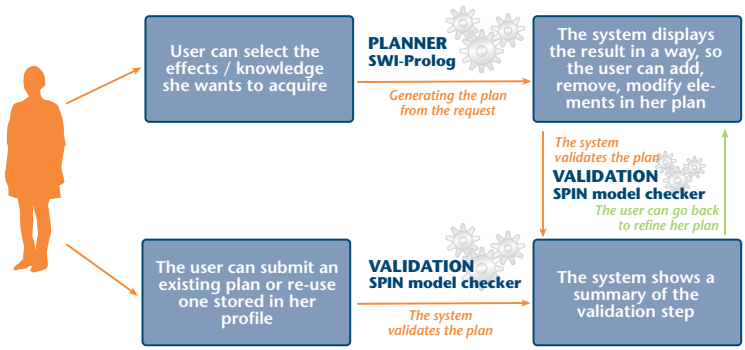


Fig. 3. The Actions supported by the User Interface

In our implementation, the user interface (see figure 3) is responsible for identifying the user, presenting the user an interface to select the knowledge she wants to acquire, and to display the results of the planning and validation step, allowing further refinement of created plans. The creation of curriculum sequences and the validation are implemented as two independent Personalization Services, the "Curriculum Planning PService", and the "Curriculum Validation PService". Because of the plug-and-play nature of the infrastructure, the two PServices can be used by other applications (SynServices) as well (Fig. 3). Also possible is that PServices, which provide additional planning and validation capabilities can be used in our application. The current and upcoming future implementations of the Curriculum Planning and Validation Prototype are available at `http://semweb2.kbs.uni-hannover.de:8080/plannersvc`.

3.3 The Curriculum Planning PService

In order to integrate the Planning Service as a plug-and-play personalization service in the Personal Reader architecture we worked at embedding the Prolog reasoner into a web service. Figure 4 gives an overview over the components in the current implementation. The web service implements the Personalization

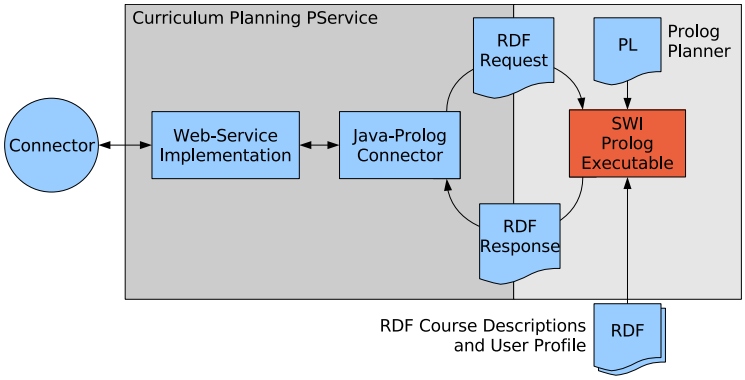


Fig. 4. Curriculum Planning Web Service

Service (*PService* [18]) interface, defined by the Personal Reader framework, which allows for the processing of RDF documents and for inquiring about the services capabilities. The *Java-to-Prolog Connector* runs the SWI-Prolog executable in a sub-process; essentially it passes the RDF document containing the request *as-is* to the Prolog system, and collects the results, already represented as RDF.

The curriculum planning task itself is accomplished by a reasoning engine, which has been implemented in SWI Prolog[3]. The interesting thing of using SWI Prolog is that it contains a semantic web library allowing to deal with RDF statements. Since all the inputs are sent to the reasoner in a *RDF request document*, it actually simplifies the process of interfacing the planner with the Personal Reader. In particular the request document contains: a) links to the RDF document containing the database of courses, annotated with metadata, b) a reference to the user's context c) the user's actual learning goal, i.e. a set of knowledge concepts that the user would like to acquire, and that are part of the *domain ontology* used for the semantic annotation of the actual courses. The reasoner can also deal with information about credits provided by the courses, when the user sets a credit constraint together with the learning goal.

Given a request, the reasoner runs the Prolog planning engine on the database of courses annotated with prerequisites and effects. The initial state is set by using information about the user's context, which is maintained by the User Modelling component of the PR. In fact such user's context includes information about what is considered as already learnt by the student (attended courses, learnt concepts) and such information is included in the request document. The Prolog planning engine has been implemented by using a classical depth-first search algorithm [22]. This algorithm is extremely simple to implement in declarative languages as Prolog.

[3] `http://www.swi-prolog.org/`

At the end of the process, a *RDF response document* is returned as an output. It contains a list of plans (sequences of courses) that fulfill the user's learning goals and profile. The maximum number of possible solutions can be set by the user in the request document. Notice that further information stored in the user profile is used at this stage for adapting the presentation of the solutions, here simple hints are used to *rank higher* those plans that include topics that the user has an expressed special interest in.

3.4 The Curriculum Validation PService

In order to verify if a curriculum is valid w.r.t. a curricula model, we adopt *model checking* techniques, by using SPIN. To check a curriculum with SPIN, this must be translated in the Promela language. *Competencies* are represented as *boolean variables*. In the beginning, only those variables that represent the initial knowledge of the student are true. *Courses* are implemented as actions that can modify the value of the variables. Since our application domain is monotonic, only those variables, whose value is false in the initial state, can be modified.

The Promela program consists of two processes: one is named *CurriculumVerification* and the other *UpdateState*. While the former contains a representation of the curriculum itself, and simulates its execution, the latter contains the code for updating the state (i.e. the set of competencies achieved so far) step by step along the simulation of the execution of the curriculum. The two processes communicate by means of two channels, *attend* and *feedback*. The notation *attend!courseName* represents the fact that the course with name *courseName* is to be attended. In this case the sender process is *CurriculumVerification* and the receiver is *UpdateState*. *UpdateState* will check the preconditions of the course in the current state and will send a feedback to *CurriculumVerification* after updating the state. On the other hand, the notation *feedback?feedbackMsg* represents the possibility for the process *Curriculum* of receiving a feedback of kind *feedbackMsg* from the process *UpdateState*.

Given these two processes, it is possible to perform a test, aimed at verifying the possible presence of competency gaps. This test is implemented as a *deadlock* verification: if the sequence is correct w.r.t. the action theory, no deadlock arises, otherwise a deadlock will be detected. The *curricula model* is to be supplied apart, as a set of temporal logic formulas, possibly obtained by an automatic translation process from a DCML representation. Notice that curricula can contain branching points. The branching points are encoded by either conditioned or non-deterministic *if*; each such *if* statement refers to a set of alternative courses (e.g. *languagesEnvironmentProg* and *programmingLanguages*). Depending on the course communicated by the channel *attend*, it updates the state. The process continues until the message *stop* is communicated. Then the learning goal is checked.

Let us see how to use the model checker to verify the *temporal constraints* that make a curricula model. Model checking is the algorithmic verification of the fact that a finite state system complies to its specification. In our case the

specification is given by the curricula model and consists of a set of temporal constraints, while the finite state system is the curriculum to be verified.

SPIN allows to specify and verify every kind of LTL formulas and it also allows to deal with curricula that at some points contain alternatives. This makes the system suitable to more realistic application scenarios. In fact, for what concerns curricula written by hand, users often do not have a clear mind and, thus, it is difficult for them to write a single sequence. In the case of curricula built by an automatic system, there are planners that are able to produce sets of alternative solutions gathered in a tree structure.

The following are examples of constraints, expressed as LTL formulas, that could be part of a curricula model:

(1) $\neg jdbc \,\mathsf{U}\, (sql \wedge relational\_algebra)$,
(2) $\neg op\_systems \,\mathsf{U}\, basis\_of\_prog$,
(3) $\neg basis\_of\_oo \,\mathsf{U}\, basis\_of\_prog$,
(4) $\Diamond basis\_of\_prog \supset \Diamond basis\_of\_java\_prog$,
(5) $\Diamond database$,
(6) $\Diamond web\_services$.

The first constraint means that before learning *jdbc* the student must own Knowledge about *sql* and about *relational algebra*. The following two constraints are of the same kind but involve different competencies. Constraint (4) means that if the student acquires knowledge about "basis of programming", he/she will also have knowledge about "basis of java programming" but the two events are not temporally related. Constraints (5) and (6) mean that soon or later knowledge about databases and web services must be acquired.

4 Conclusion, Further and Related Works

In this work we have described the current state of the integration of semantic personalization web services for Curriculum Planning and Validation within the Personal Reader Framework. The goal of personalization is to create sequences of courses that fit the specific context and the learning goal of individual students. Despite some manual post-processing for fixing inconsistencies, we used real information from the Hannover University database of courses for extracting the meta-data. Currently the courses are annotated also by meta-data concerning the schedule and location of courses, like for instance room-numbers, addresses and teaching hours. As a further development, it would be interesting to let our Curriculum Planning Service to make use also of such metadata in order to find a solution that fits the desires and the needs of the user in a more complete way.

The Curriculum Planning Service has been integrated as a new plug-and-play personalization service in the Personal Reader framework. In the current implementation, the learning goal corresponds to a set of hard constraints; that is to say that the planner returns only plans that satisfy them *all*. A different choice would be to consider the constraints given by the goal as *soft* constraints, and allow the return of plans which do satisfy the goal only partially. This

would be approapriate, for instance, in the case in which a student would like to acquire a range of competencies of interest but it is not possible to build, on top of a given repository of course descriptions, a curriculum for achieving them all. Nevertheless, it would be possible to build a curriculum for achieving *part* of them. In some circumstances, it would anyway be helpful for the student to receive this information as a feedback. Of course, in this case many questions arise, e.g. the issue of ranking the goals based on the actual interest of the requestor, so to know what can possibly be discarded and what is mandatory. From an implementation perspective, the spirit of the SOA infrustructure given to the Personal Reader is, indeed, meant to easily allow extensions by adding new Personalization Services. We can, therefore, think to develop and add a soft-goal planning service, to be used in these circumstances. The new planner would inherit the wrapping and interaction part from the current planning service but implement an algorithm like for instance [16].

The Curriculum Validation Service has been designed. An early prototype of the validation system based on the model checker SPIN has been developed [5] and is currently being embedded in the same framework. The choice of relying on SPIN, rather than developing a simpler and ad hoc checking system, is due to the need of rapidly developing a prototype. For this reason we have decided to rely on already exisiting and well-established technology. The engineering of the developed services should be tailored to the specific kinds of constraint that can be used to design the model. Analogous considerations can be done for the planning algorithm. The one that has been used is the simplest that can be thought of. Of course, there are many possible optimizations and extensions (e.g. the adoption of soft goals mentioned above) that could be done, and many algorithms are already available in the literature. Our choice has been motivated by the desire of quickly testing our ideas rather than developing a system thought for real use.

The Personal Reader Platform provides a natural framework for implementing a service-oriented approach to personalization in the Semantic Web, allowing to investigate how (semantic) web service technologies can provide a suitable infrastructure for building personalization applications, that consist of re-usable and interoperable personalization functionalities. The idea of taking a service oriented approach to personalization is quite new and was born within the personalization working group of the Network of Excellence REWERSE (Reasoning on the Web with Rules and Semantics, http://rewerse.net).

Writing curricula models directly in LTL is not an easy task for the user. For this reason, we have recently developed a graphical language, called DCML (Declarative Curricula Model Language) [8, 4], inspired by DecSerFlow, the Declarative Service Flow Language by van der Aalst and Pesic [23]. DCML allows to express the temporal relations between the times of acquisition of the concepts. 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. At the same time, a rigorous and precise meaning is also given, due to the logic grounding of

the language. Moreover, in [4] we represent curricula as UML activity diagrams and include the possibility of handling the concurrent attending of courses. Also in this case curricula can be translated in Promela programs so that it becomes possible to perform all the kinds of verification that we have described.

DCML, besides being a graphical language, has also a textual representation. We are currently working at an integration of this new more sophisticated solution into the Personal Reader Framework by implementing an automatic system for translating DCML textual representations into LTL, for translating curricula (activity diagrams) in Promela, and then run the checks.

Another recent proposal for automatizing the competency gap verification is done in [21] where an analysis of pre- and post-requisite annotations of the Learning Objects (LO), representing the learning resources, is proposed. In this approach, whenever an error will be detected by the validation phase, a correction engine will be activated. This engine will use a "Correction Model" to produce suggestions for correcting the wrong curriculum, by means of a reasoning-by-cases approach. The suggestions will, then, be presented to the course developer, who is in charge to decide which ones to adopt (if any). Once a curriculum will have been corrected, it will have to be validated again, because the corrections might introduce new errors. Melia and Pahl's proposal is inspired by the CocoA system [11], that allows to perform the analysis and the consistency check of static web-based courses. Competency 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 by knowledge elements.

Acknowledgement 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`).

References

1. F. Abel, I. Brunkhorst, N. Henze, D. Krause, K. Mushtaq, P. Nasirifar, and K. Tomaschweski. Personal reader agent: Personalized access to configurable web services. Technical report, Distributed Systems Institute, Semantic Web Group, University of Hannover, 2006.
2. Advanced Distributed Learning Network. SCORM: The sharable content object reference model, 2001. http://www.adlnet.org/Scorm/scorm.cfm.
3. M. Baldoni, C. Baroglio, I. Brunkhorst, N. Henze, E. Marengo, and V. Patti. A Personalization Service for Curriculum Planning. In E. Herder and D. Heckmann, editors, *Proc. of the 14th Workshop ABIS*, pages 17–20, Hildesheim, Germany, October 2006.
4. M. Baldoni, C. Baroglio, and E. Marengo. Curricula Modeling and Checking. In *Proc. of AI*IA 2007: Advances in Artificial Intelligence*, volume 4733 of *LNAI*, pages 471–482. Springer, 2007.
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 the Semantic Web Personalization Workshop, pages 53–62, Budva, Montenegro, 2006.
6. M. Baldoni, C. Baroglio, and V. Patti. Web-based adaptive tutoring: An approach based on logic agents and reasoning about actions. *Artificial Intelligence Review*, 1(22):3–39, 2004.
7. M. Baldoni, C. Baroglio, V. Patti, and L. Torasso. Reasoning about learning object metadata for adapting SCORM courseware. In L. Aroyo and C. Tasso, editors, *Int. Workshop on Engineering the Adaptive Web, EAW'04*, pages 4–13, 2004.
8. M. Baldoni and E. Marengo. Curriculum Model Checking: Declarative Representation and Verification of Properties. In *Proc. of 2nd Eur. Conf. EC-TEL*, volume 4753 of *LNCS*, pages 432–437. Springer, 2007.
9. R. Baumgartner, S. Flesca, and G. Gottlob. Visual web information extraction with lixto. In Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass, editors, *VLDB*, pages 119–128. Morgan Kaufmann, 2001.
10. R. Baumgartner, N. Henze, and M. Herzog. The personal publication reader: Illustrating web data extraction, personalization and reasoning for the semantic web. In *ESWC*, pages 515–530, 2005.
11. 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.
12. O. E. M. Clarke and D. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 2001.
13. Juri L. De Coi, Eelco Herder, Arne Koesling, Christoph Lofi, Daniel Olmedilla, Odysseas Papapetrou, and Wolf Sibershi. A model for competence gap analysis. In *Proc. of WEBIST 2007*, 2007.
14. E. A. Emerson. Temporal and model logic. In *Handbook of Theoretical Computer Science*, volume B, pages 997–1072. Elsevier, 1990.
15. European Commission, Education and Training. The Bologna process. `http://ec.europa.eu/education/policies/educ/bologna/bologna_en.html`.
16. E. Giunchiglia and M. Maratea. SAT-based planning with minimal-♯actions plans and "soft" goals. In *Proc. of AI*IA 2007: Advances in Artificial Intelligence*, volume 4733 of *LNAI*. Springer, 2007.
17. N. Henze. Personal readers: Personalized learning object readers for the semantic web. In *12th International Conference on Artificial Intelligence in Education, AIED05*, Amsterdam, The Netherlands, 2005.
18. N. Henze and D. Krause. Personalized access to web services in the semantic web. In *The 3rd International Semantic Web User Interaction Workshop (SWUI, collocated with ISWC 2006*, November 2006.
19. IMSGlobal. Learning design specifications. Available at `http://www.imsglobal.org/learningdesign/`.
20. R. Koper and C. Tattersall. *Learning Design: A Handbook on Modelling and Delivering Networked Education and Training*. Springer Verlag, 2005.
21. M. Melia and C. Pahl. Automatic Validation of Learning Object Compositions. In *Information Technology and Telecommunications Conference IT&T'2005: Doctoral Symposium*, Carlow, Ireland, 2006.
22. S. Russel and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.
23. W. M. P. van der Aalst and M. Pesic. DecSerFlow: Towards a Truly Declarative Service Flow Language. In Mario Bravetti and Gialuigi Zavattaro, editors, *Proc. of WS-FM*, LNCS, Vienna, September 2006. Springer.

M. Baldoni and E. Marengo. Curriculum Model Checking: Declarative Representation and Verification of Properties. In E. Duval, R. Klamma, and M. Wolpers, editors, Proc. of EC-TEL 2007 - Second European Conference on Technology Enhanced Learning, volume 4753 of LNCS, pages 432-437. Springer, 2007.

Curriculum Model Checking: Declarative Representation and Verification of Properties

Matteo Baldoni and Elisa Marengo

Dipartimento di Informatica — Università degli Studi di Torino
C.so Svizzera, 185 — I-10149 Torino (Italy)
baldoni@di.unito.it,elisa.mrng@gmail.com

Abstract. When a curriculum is proposed, it is important to verify at least three aspects: that the curriculum allows the achievement of the user's learning goals, that the curriculum is compliant w.r.t. the course design goals, specified by the institution that offers it, and that the sequence of courses that defines the curriculum does not have competency gaps. In this work, we present a constrained-based representation for specifying the goals of "course design" and introduce a design graphical language, grounded into Linear Time Logic.

Keywords: formal model for curricula description, *model checking*, *verification of properties*, *competence gaps*.

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 [11,8]. The more learning resources are freely available through the Web, the more 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 services are retrieved and composed [8]. As in a service composition it is necessary to verify that, at every point, all the information necessary to the subsequent invocations is 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, i.e. a *curriculum*, does not have to show any *competency gap*. Unfortunately, this verification, is usually performed *manually* by the designer, with hardly any guidelines or support [6].

In [11] an analysis of pre- and post-requisite annotations of the Learning Objects (LO), representing the learning resources, is proposed for automatizing the competency gap verification. A logic based validation engine can use these annotations to validate the LO composition. This proposal is inspired by the CocoA system [5], that allows to perform the analysis and the consistency check of static web-based courses. Competency 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 by concepts, elementary pieces of domain of knowledge.

Brusilovsky and Vassileva [5] sketch many other kinds of verification. In our opinion, two of them are particularly important: (a) verifying that the curriculum allows

E. Duval, R. Klamma, and M. Wolpers (Eds.): EC-TEL 2007, LNCS 4753, pp. 432–437, 2007.

to achieve the users' *learning goals*, and (b) verifying that the curriculum is compliant against the *course design goals*. Verifying (a) is fundamental to guarantee that users will acquire the desired knowledge. At the same time, manually or automatically supplied curricula, developed to reach that learning goal, should match the design document, a curricula model, specified by the institution. Curricula models specify general rules for designing sequences of learning resources (courses) and can be interpreted as *constraints*. These constraints are to be expressed in terms of *concepts* and, in general, it is not possible to associate them directly to a learning resource, as instead is done for pre-requisites, because they express constraints on the acquisition of concepts, independently from the resources that supply them.

This work differs from previous work [4], where the authors presented an adaptive tutoring system, that exploits *reasoning about actions and changes* to plan and verify curricula. That approach was based on abstract representations, capturing the *structure* of a curriculum, and implemented as prolog-like clauses. A procedure-driven planning was applied to build personalized curricula. The advantage of such planning techniques is that the only curricula that are tried are the possible executions of the procedure itself, and this restricts considerably the search space of the planning process. In this context, we proposed also forms of verification: of competency 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. 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, this approach becomes unfeasible.

For this reason it is 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 describing communication protocols for multi-agent systems and service oriented architecture [12]. 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.

In this paper we present a constraint-based representation of curricula models. Constraints are expressed as formulas in a temporal logic (LTL, linear-time logic [7]) represented by means of a simple graphical language that we call DCML (*Declarative Curricula Model Language*). This kind of logic allows the verification of some properties of interest for all the possible executions of a model, which in our case corresponds to the specific curriculum.

2 DCML: A Declarative Curricula Model Language

In this section we describe our Declarative Curricula Model Language (DCML), a graphical language to represent the relations that can occur among concepts supplied

by attending courses. DCML is inspired by DecSerFlow, the Declarative Service Flow Language to specify, enact, and monitor web service flows [13]. As such, DCML is grounded in Linear Temporal Logic (LTL) [7] and it allows a curricula model to be described in an easy way, with 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* (α U β, the formula α remains true until β). The set of LTL formulas obtained for a curricula model are, then, used to verify whether a curriculum will respect it [3]. As an example, Fig. 1 shows a curricula model expressed in DCML. Every box

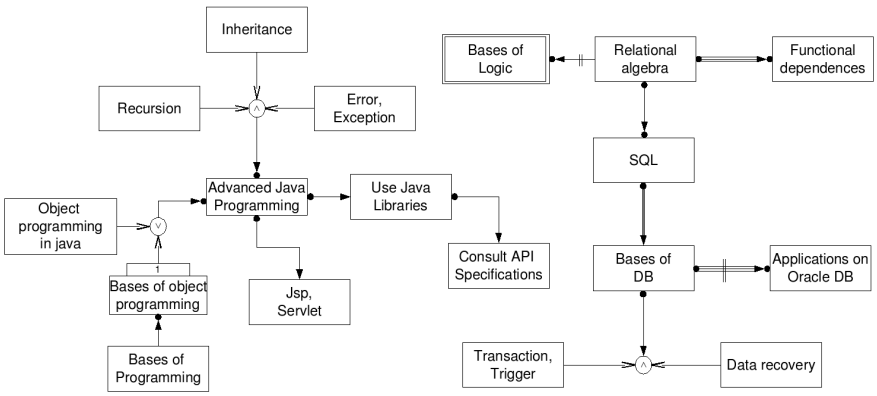


Fig. 1. An example of curricula model in DCML

contains at least one competency. Boxes/competencies 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. Hereafter, we describe most of such elements.

The simplest kinds of constraint concern the *existence*, *absence*, or *possibility* of acquisition for a certain competency. The *existence constraint* imposes that a certain concept k must be acquired sooner or later. It captures the fact that a concept characterizes a curriculum, so a student cannot present a plan in which it does not appear. It is represented by the LTL formula $\Diamond k$, that is k must eventually become true. Similarly, a course designer can impose that a concept k must never appear in a curriculum. This is possible by means of the *absence constraint*. The LTL formula $\Box\neg k$ expresses this fact: it means that k cannot appear. On the diagram these two constraints are given by marking boxes with the "cardinality" of the concepts (1 for existence and 0 for absence). When both 0 and 1 appear on the same box, we have a *possibility constraint*. The corresponding LTL formula is $\Diamond k \vee \Box\neg k$. When no cardinality is expressed explicitly, possibility is assumed. The last constraint on concepts is represented by a double box, which means that a concept k must belong to the initial knowledge of the student. In other words, the simple logic formula k must hold in the initial state.

In DCML it is also 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, although an example can be seen in Fig. 1. The interested reader can find an extended version of this paper that is available in the home page of the authors.

Besides the representation of competencies and of constraints on competencies, DCML allows to represent *relations* among competencies. For simplicity, in the following presentation we will always relate simple competencies, although it is, of course, possible to connect DNF formulas.

Arrows ending with a little-ball, express the *before* temporal constraint between two concepts: a concept must be acquired *before* another one. This constraint can be used to express that, to understand a topic, some other knowledge is required. Notice that if the antecedent never becomes true, also the consequent must be invariably false. k_1 *before* k_2 corresponds to the LTL formula $\neg k_2 \ \mathsf{U} \ k_1$.

One can express that a concept must be acquired *immediately before* some other by means of a triple line arrow that ends with a little-ball. The constraint "k_1 *immediate before* k_2" imposes that k_1 is acquired before k_2 and that either k_2 is true in the next state (w.r.t. when k_1 is acquired) or it is never acquired. Immediate before is stronger than before because it imposes that two concepts have to be acquired in strict sequence. The LTL formula for *immediate before* is $\neg k_2 \ \mathsf{U} \ k_1 \wedge \Box(k_1 \supset (\bigcirc k_2 \vee \Box \neg k_2))$, that is k_1 *before* k_2 and whenever k_1 holds, either in the next state k_2 holds or k_2 never holds.

The *implication* relation specifies, instead, that if a certain concept holds, some other concept must be acquired sooner or later. The acquisition of the consequent is imposed by the truth value of the antecedent, but, in case this one is true, the implication does not specify when the consequent is to be achieved (it could be before, after or in the same state as the antecedent). k_1 *implies* k_2 is expressed by the LTL formula $\Diamond k_1 \supset \Diamond k_2$.

The *immediate implication* instead, specifies that the consequent must hold in the state right after the one in which the antecedent is acquired. 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 holds, k_2 holds in the next state: $\Diamond k_1 \supset \Diamond k_2 \wedge \Box(k_1 \supset \bigcirc k_2)$. *Implication* and *immediate implication* are graphically represented with an arrow that starts with a little-ball and with a triple line arrow that starts with a little-ball.

The last two kinds of temporal constraints are *succession* and *immediate succession* The *succession* relation specifies that if k_1 is acquired, afterwards k_2 is also achieved. Succession is expressed by the LTL formula $\Diamond k_1 \supset (\Diamond k_2 \wedge (\neg k_2 \ \mathsf{U} \ k_1))$. While in the *before* relation, when the antecedent is never acquired also the consequent must be false, in the *succession* relation this is not relevant. This behaviour is due to the fact that the *succession* specifies a condition of the kind: *if* k_1 *then* k_2. The *before*, instead, represents a constraint without any conditional premise. The fact that the consequent must be acquired *after* the antecedent differentiates *implication* from *succession*.

The *immediate succession* imposes that the acquisition of the consequent must happen either in the same state the antecedent is acquired or in the state immediately after (not before nor later). The immediate succession is expressed by the LTL formula: $\Diamond k_1 \supset (\Diamond k_2 \wedge (\neg k_2 \ \mathsf{U} \ k_1)) \wedge \Box(k_1 \supset \bigcirc k_2)$. The representation of (*immediate*) *succession*, see Fig. 1, is an (triple) arrow that starts and ends with a little-ball.

The graphical notations for "negative relations" is very intuitive: two vertical lines break the arrow that represents the constraint. Some examples are shown in Fig. 1.

k_1 *not before* k_2 specifies that the concept k_1 cannot be acquired before or in the same state of the concept k_2. The LTL formula is $\neg k_1 \ \mathsf{U} \ (k_2 \wedge \neg k_1)$. Notice that this is not obtained by simply negating the before relation but it is weaker because the negation would impose the acquisition of the concepts specified as consequents, the *not before* does not. The *not immediate before* is translated exactly in the same way of the *not before*. Indeed, it is a special case of *not before*. This happens because the acquired knowledge cannot be forgotten.

By means of k_1 *not implies* k_2 we express that the acquisition of the concept k_1 implies that k_2 will never be acquired. We express this by the LTL formula $\Diamond k_1 \supset \Box \neg k_2$. Again, we choose to use a weaker formula than the natural negation of the implication relation, that is $\Diamond k_1 \wedge \Box \neg k_2$. k_1 *not immediate implies* k_2 constraint imposes that when the concept k_1 is acquired, in the immediately subsequent state, the concept k_2 must be false. Afterwards, the truth value of k_2 does not matter (it is weaker than $\neg(k_1$ *immediate implies* $k_2)$). The corresponding LTL formula is $\Diamond k_1 \supset (\Box \neg k_2 \vee \Diamond(k_1 \wedge \bigcirc \neg k_2))$.

The *not succession*, and the *not immediate succession* are weaker versions of, respectively, negation of succession and of immediate succession. The first one imposes that a concept cannot be acquired after another. This means that it could be acquired before, or it will always be false. The LTL formula is $\Diamond k_1 \supset (\Box \neg k_2 \vee k_1$ *not before* $k_2)$. The second imposes that if a concept is acquired in a certain state, in the state that follows another concept must be false: $\Diamond k_1 \supset (\Box \neg k_2 \vee k_1$ *not before* $k_2 \vee \Diamond(k_1 \wedge \bigcirc \neg k_2))$.

3 Conclusions

The presented work is an evolution of earlier works [2,4]. In those works, we semantically annotated learning objects with the aim of building compositions of new learning objects, based on the user's learning goals and by 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. Of course, the new proposal, presented in this paper, can be applied also to learning objects, given a semantic annotation of theirs, as introduced in the cited works. In [1] we discuss the integration, into the Personal Reader Framework [9], of a verification web service that implements the explained techniques.

In particular, in this work we have introduced a graphical language to describe temporal constraints posed on the acquisition of competencies (supplied by courses). In the extended version, that is available on-line, we show how to use UML activity diagrams to specify sets of curricula, and we show how to translate them into Promela programs. Such programs can be used by the SPIN model checker [10] to verify whether the curriculum respects the DCML model. Model checking can also be applied for checking the achievement of the user's learning goals and the presence of competency gaps.

In [3] we extend the current proposal so as to include a representation of the *proficiency level* at which a competency is owned or supplied, as suggested in [6]. The key idea is to associate to each competency a variable, having the same name as the competency, which can be assigned natural numbers as values. The value denotes the proficiency level; zero means absence of knowledge. The next step will be to give a structure to competencies, e.g. by defining a proper ontology, for allowing more flexible forms of reasoning and verification.

We are currently working on an 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.

Acknowledgements. The authors would like to thank Cristina Baroglio, and also Viviana Patti and Ingo Brunkhorst, 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. 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 2007 (2007)
2. 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)
3. Baldoni, M., Baroglio, C., Marengo, E.: Curricula Modeling and Checking. In: Proc. of AI*IA 2007, Rome, Italy. LNCS (LNAI), Springer, Heidelberg (2007)
4. 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)
5. 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)
6. De Coi, J.L., Herder, E., Koesling, A., Lofi, C., Olmedilla, D., Papapetrou, O., Sibershi, W.: A model for competence gap analysis. In: Proc. of WEBIST 2007, INSTICC Press (2007)
7. Emerson, E.A.: Temporal and model logic. Handbook of Theoretical Computer Science B, 997–1072 (1990)
8. Farrell, R., Liburd, S.D., Thomas, J.C.: Dynamic assebly of learning objects. In: Proc. of WWW 2004, New York, USA (2004)
9. Henze, N., Krause, D.: Personalized access to web services in the semantic web. In: Cruz, I., Decker, S., Allemang, D., Preist, C., Schwabe, D., Mika, P., Uschold, M., Aroyo, L. (eds.) ISWC 2006. LNCS, vol. 4273, Springer, Heidelberg (2006)
10. Holzmann, G.J.: The SPIN Model Checker. Addison-Wesley, Reading (2003)
11. Melia, M., Pahl, C.: Automatic Validation of Learning Object Compositions. In: Proc. of IT&T'2005: Doctoral Symposium, Carlow, Ireland (2006)
12. Singh, M.P.: Agent communication languages: Rethinking the principles. IEEE Computer 31(12), 40–47 (1998)
13. 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)

M. Baldoni, C. Baroglio, I. Brunkhosrt, E. Marengo, and V. Patti. Reasoning- based Curriculum Sequencing and Validation: Integration in a Service- Oriented Architecture. In E. Duval, R. Klamma, and M. Wolpers, editors, Proc. of EC- TEL 2007 - Second European Conference on Technology Enhanced Learning, number 4753 in LNCS, pages 426- 431. Springer, 2007.

Reasoning-Based Curriculum Sequencing and Validation: Integration in a Service-Oriented Architecture

Matteo Baldoni[1], Cristina Baroglio[1], Ingo Brunkhorst[2], Elisa Marengo[1], and Viviana Patti[1]

[1] Dipartimento di Informatica, Università degli Studi di Torino
C.so Svizzera, 185 — I-10149 Torino, Italy
{baldoni,baroglio,patti}@di.unito.it,elisa.mrng@gmail.com
[2] L3S Research Center, University of Hannover, D-30539 Hannover, Germany
brunkhorst@l3s.de

Abstract. We present a service-oriented personalization system, set in an educational framework, based on a semantic annotation of courses including prerequisites and learning objectives. The system supports users in planning personalized curricula and in verifying the compliance of curricula against a model describing the designer goals. We have developed a prototype of the planning and validation services, by using SWI-Prolog and the SPIN model checker as reasoning engines. The services are supplied and combined in the Personal Reader framework.

1 Introduction

So far, reasoning in the Semantic Web is mostly reasoning about knowledge expressed in some ontology. However, personalization may involve also other kinds of reasoning and knowledge representation. Moreover, the next Web generation promises to deliver Semantic Web Services, that can be retrieved and *combined* in a way that satisfies the user. It opens the way to many forms of *service-oriented personalization*. Web services provide an ideal infrastructure for enabling *interoperability* among personalization applications and for constructing Plug&Play-like environments, where the user can select and combine the preferred kinds of services. Based on our previous work [2,3], our current goal is to implement such results in an organic system, where different reasoning-based personalization services can be combined for supporting the user in building a curriculum from a set of *learning resources* (courses). We achieve this by developing a Planner and a Validation Service within the Personal Reader(PR) Framework, where a service oriented approach to personalization is taken [12].

While in early times learning resources were simply considered as "contents", strictly tied to the platform used for accessing them, attention has been posed on the issue of *re-use* and of a *cross-platform* use of educational contents. The proposed solution is to adopt a *semantic annotation* of contents based on standard languages, e.g. RDF and LOM, and then to create learning resources formed by *educational contents* plus *semantic meta-data*, which supply information on the resources at a *knowledge level* (e.g. concepts from an ontology of the educational domain). Learning Resources are interpreted as *actions* [3,4], capturing the *learning objectives* and *pre-requisites*. Thus, we

E. Duval, R. Klamma, and M. Wolpers (Eds.): EC-TEL 2007, LNCS 4753, pp. 426–431, 2007.

can rely on a classical theory of actions and apply different reasoning methods, like planning, for building personalized curricula. Our interpretation of learning resources also enables the use of model checking techniques for developing a validation service that detects if a curriculum is compliant w.r.t an abstract model, which encodes the *curricula-design goals*.

Motivating Scenario. Curriculum planning and validation offer useful support in many practical contexts for helping both students and teaching institutions. Since taking courses at different Universities is becoming more common in Europe, creating a personalized curriculum becomes a difficult task for students. Even if the students know what competency they would like to acquire, it's harder to find the courses that help to acquire it: an automatic system that can suggest a pathway through the course repository can be very helpful. The need for support in building personalized paths through learning resources has to be *combined* with the ability to ensure the compliance of the resulting curriculum with *curricula-design goals*, expressed by the teachers or by the *institution* offering the courses. Curricula models specify general rules for building learning paths, e.g. constraints designed by the University for guaranteeing the achievement of certain learning goals. These constraints are to be expressed in terms of knowledge elements, and maybe also on features that characterize the resources. For a provider or university, which needs to certify that a specific offered curricula for achieving a certain educational goal respects some European guidelines, we could define the guidelines as a set of constraints at an abstract level, i.e. as relations among a set of competencies which should be offered in a way that meets some given scheme. The automatic checking of compliance combined with curriculum planning could be used for implementing processes like cooperation among institutes in curricula design and integration, which are the focus of the *Bologna Process* [11], promoted by the EU.

2 Curricula Representation and Reasoning

All the different kinds of objects that we need to tackle (learning resources, curricula, and curricula models) are described on the basis of a set of *competencies*, i.e. terms identifying specific *knowledge elements*. Competencies can be thought of, and implemented, as concepts in a shared ontology. In our implementation, competencies have been semi-automatically extracted, and then stored in a RDF file (see the next section).

Learning Resources and Curricula. A *curriculum* is a sequence of *learning resources* that are homogeneous in their representation. Based on work in [3,4], we rely on an *action theory*, and take the abstraction of resources as *atomic actions*. A learning resource is modelled as an action for acquiring some competencies, called *effects*. For understanding the contents supplied by a learning resource, the user can be required to own other competencies, called *preconditions*. Both preconditions and effects can be expressed by means of a *semantic annotation* of the learning resource [4]. Since we will focus on university curricula, we will refer to learning resources as "courses". Given the above interpretation of learning resources, a *curriculum* is modelled as a *plan*, i.e. as a sequence of actions, whose execution causes transitions from a state to another, until some final state is reached. The *initial state* (possibly empty) contains all the

competences that we suppose available before the curriculum is taken, e.g. the knowledge that the student already has. This set typically *grows* as the student studies and learns. Curricula are usually designed so to allow the achievement of a *learning goal*; in such cases the *final state* should contain specific knowledge elements, e.g. all those that compose the user's learning goal. A transition between two states is due to the application of the action corresponding to a learning resource. For an action to be applicable, its preconditions must hold in the state to which it should be applied. The application of the action consists in an *update* of the state. We assume that competences can only be added to states. The intuition behind this assumption is that the act of using a new resource will never erase from the students' memory the concepts acquired insofar.

Curricula Models. We would like to restrict the set of possible sequences of resources composing a curriculum, by imposing constraints on the *order* by which knowledge elements are added to the states, e.g. "a knowledge element α is to be acquired before a knowledge element β", or by specifying some *educational objective* to be achieved, in terms of knowledge that must be contained in the final state, e.g. "a knowledge element α must be acquired sooner or later". Therefore, we represent a *curricula model* as a set of temporal constraints building upon knowledge elements. A model is independent from the available resources and it can be reused in different contexts. A natural choice for representing temporal constraints on action paths is linear-time temporal logic (LTL) [10]. This kind of logic allows to verify if a property of interest is true for all the possible executions of a model (in our case the specific curriculum). This is often done by means of model checking techniques [8]. A curriculum as we represent it is, actually, a Kripke structure, that identifies a set of states with a transition relation for passing from a state to another. Since in our domain we assume that knowledge only grows, states will always contain *all* the competencies acquired up to that moment. The transition relation is given by the actions that are contained in the curriculum that is being checked. The LTL logic can be used to verify if a given formula holds starting from a state or if it holds for a set of states. For instance, in order to specify in the curricula model constraints on *what* to achieve, we can use the formula $\Diamond\alpha$ ($\Diamond$ is the *eventually* operator), meaning that a set of knowledge elements will be acquired sooner or later. Instead, constraints on *how* to achieve the educational objectives, such as "a knowledge element β cannot be acquired before the knowledge element α is acquired", can be expressed by the LTL formula $\neg\beta\ U\ \alpha$, where U is the *weak until* operator. Writing curricula models directly in LTL is not an easy task for the user. We are developing a graphical language, called DCML (Declarative Curricula Model Language) [5], inspired by DecSerFlow [15]. By means of DCML the user can easily write curricula models, maintaining a rigorous meaning due to the logic grounding of the language.

Curriculum Planning and Validation. Given a semantic annotation with preconditions and effects of the courses, classical planning techniques are exploited for creating *personalized curricula*, in the spirit of the work in [3,4]. Intuitively the idea is that, given a repository of annotated learning resources the user expresses a *learning goal* as a set of *knowledge elements* he/she would like to acquire, and possibly also a set of already owned competencies. Then, the system applies planning to build a sequence of learning resources that will allow him/her to achieve the goal. The planning methodology that

we implemented (see Section 3) is a simple *depth-first forward planning* where actions cannot be applied more than once. An early prototype was presented in [1].

There are two main validation tasks that can be performed on curricula and curricula models. The simplest one consists in *checking the soundness* w.r.t. the learning dependencies and the learning goal of curricula which are built *by hand* by users themselves. Usually, soundness verification is performed manually by the learning designer, with hardly any guidelines or support [9]. Not all sequences which can be built starting from a set of learning resources are lawful. It is important to verify that all the *competencies*, that are necessary to fully understand the contents, offered by a learning resource, are introduced or available before that resource is accessed. In other words, a course can appear at a certain point in a sequence only there are no *competency gaps*. These implicit "applicability constraints" capture dependencies that are innate to the nature of the taught concepts. Given the interpretation of resources as actions, the verification of the *soundness of a curriculum*, w.r.t. the learning dependencies and the learning goal, can be interpreted as an *executability check* of the curriculum.

The other interesting verification task consists in checking if a curriculum (possibly automatically generated by a personalization service) is *compliant against the course design goals* [7]. A curriculum personalized w.r.t. the user desires, that is proved to be sound, cannot automatically be considered as being *valid* w.r.t. a particular *curricula model* describing some designer goal. The curricula model imposes further constraints on *what* to achieve and *how* achieving it. In our validation service (Section 3) the verification tasks are performed by using the SPIN model checker [13]. SPIN is used for verifying systems that can be represented by *finite state structures*, where the specification is given in an LTL logic. The verification algorithm is based on the exploration of the state space. This is exactly what we need for performing all the verification tests that we mentioned, provided that we can translate the curriculum in the internal representation used by the model checker (in SPIN such representation is given in Promela).

3 Implementation in the Personal Reader Framework

The Personal Reader Platform provides a framework for implementing personalization in the Semantic Web in a service-oriented approach, allowing to investigate how (semantic) web service technologies can provide a suitable infrastructure for building personalization applications. So called *Personalization Services* (PServices) [12] are the basic building blocks for implementing plug-and-play like personalization services in this architecture, they are semantic in the sense that they communicate solely on the basis of *RDF* documents. Besides PServices, the PR framework also includes other kinds of components, namely *Syndication Services*, *User Interfaces* and a *Connector*.

Corpus of Courses and Metadata Description. Despite some manual post-processing for fixing inconsistencies, we extracted a corpus of courses and the related meta-data by extracting real data from the Hannover University database via an automatic extraction with the Lixto [6] tool. We focussed only on a subset of the courses and manually post-processed the data, resulting in corpus with 65 courses, with 390 effects and 146 preconditions. Metadata contains also course names, semester, credit points, the type of course (e.g. laboratory, etc.), schedule and location.

Reasoners as PServices. We implemented two independent PServices for our system, the "Curriculum Planning PService", and the "Curriculum Validation PService" (Fig. 1), whic can be used by other applications as well.

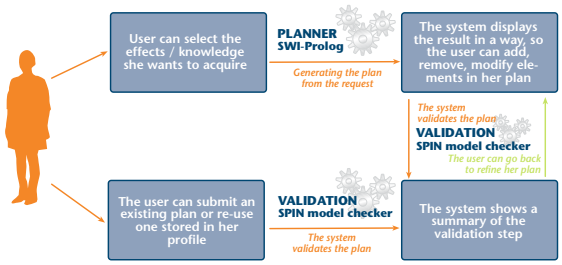


Fig. 1. The interaction with the system

The Curriculum Planning PService is basically divided in two parts: the core reasoner (the planner) and the wrapper (the web service implementation) interfacing with the PR framework. The reasoning engine that actually accomplish the curriculum planning task has been implemented in SWI Prolog by using a classical depth-first search algorithm. The initial state is set by using information about the user's context provided by the User Modelling module of the PR. SWI Prolog contains a semantic web library allowing to deal with RDF statements. Since all the inputs are sent to the reasoner in a *RDF request document*, it actually simplifies the process of interfacing the planner with the PR. The request document contains: a) links to the RDF document containing the database of courses, annotated with metadata, b) a reference to the user's context c) the user's actual learning goal, i.e. a set of knowledge concepts that the user would like to acquire, and that are part of the *domain ontology* used for the semantic annotation of the actual courses. The reasoner can also deal with information about credits provided by the courses, when the user sets a credit constraint together with the learning goal. The reasoner returns as output a *RDF response document*, which contains a list of plans that fulfill the user's learning goals and profile. Information stored in the user profile is used for *ranking higher* those plans that include the user's preferred topics.

An early prototype[1] of the Curriculum Validation PService based on the model checker SPIN has been designed and is currently being embedded in the PR. Model checking is the algorithmic verification of the fact that a finite state system complies to its specification. In our case the specification is given by the curricula model and consists of a set of temporal constraints, while the finite state system is the curriculum to be verified. The advantage of using a model checker like SPIN, rather than an ad hoc implementation is that it can handle any kind of LTL temporal formula. Moreover, we can also deal with the validation of *non-linear* curricula, i.e. curricula that contain branching points. This kind of curricula allow to account for *uncertainties* of the user. In fact a branching point corresponds to a possible choice among alternative resources.

[1] `http://www.l3s.de/~brunkhor/semweb/curriculum/`

4 Conclusion

In this work we have sketched the current state of the integration of semantic personalization web services for Curriculum Planning and Validation within the Personal Reader Framework. We are actually investigating how to extend the application with a module of geo-spatial reasoning working on meta-data like floor-plans and locations.

In [14] an analysis of pre- and post-requisite annotations of learning object is proposed with the aim of dealing with competency gap verification. In this approach, whenever an error will be detected by the validation phase, a correction engine will be activated, that produce suggestions on how to correct the wrong curriculum, by using reasoning-by-cases. The suggestions are presented to the course developer, who can decide which ones to adopt. Once a curriculum have been corrected, it must be validated again: the corrections might introduce errors. The proposal is inspired by the CocoA system [7], that allows to perform the consistency check of web-based courses.

References

1. Baldoni, M., Baroglio, C., Brunkhorst, I., Henze, N., Marengo, E., Patti, V.: A Personalization Service for Curriculum Planning. In: Proc. of the 14th Workshop on Adaptivity and User Modeling in Interactive Systems, ABIS 2006, pp. 17–20. Hildesheim, Germany (2006)
2. 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 the Semantic Web Personalization Workshop, pp. 53–62, Budva, Montenegro (2006)
3. Baldoni, M., Baroglio, C., Patti, V.: Web-based adaptive tutoring: An approach based on logic agents and reasoning about actions. Artificial Intelligence Review 1(22), 3–39 (2004)
4. Baldoni, M., Baroglio, C., Patti, V., Torasso, L.: Reasoning about learning object metadata for adapting SCORM courseware. In: Proc. of EAW'04, pp. 4–13 (2004)
5. Baldoni, M., Marengo, E.: Curricula model checking: declarative representation and verification of properties. In: Proc. of EC-TEL'07. LNCS, Springer, Heidelberg (2007)
6. Baumgartner, R., Flesca, S., Gottlob, G.: Visual web information extraction with Lixto. In: VLDB, pp. 119–128. Morgan Kaufmann, San Francisco (2001)
7. 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)
8. Clarke, O.E.M., Peled, D.: Model checking. MIT Press, Cambridge, MA, USA (2001)
9. De Coi, J.L., Herder, E., Koesling, A., Lofi, C., Olmedilla, D., Papapetrou, O., Sibershi, W.: A model for competence gap analysis. In: Proc. of WEBIST 2007 (2007)
10. Emerson, E.A.: Temporal and modal logic. Handbook of Theoretical Computer Science B, 997–1072 (1990)
11. European Commission, Education and Training. The Bologna process, `http://europa.eu.int/comm/education/policies/educ/bologna/bologna_en.html`
12. Henze, N., Krause, D.: Personalized access to web services in the semantic web. In: The 3rd Int. Semantic Web User Interaction Workshop, SWUI (November 2006)
13. Holzmann, G.J.: The SPIN Model Checker. Addison-Wesley, Reading (2003)
14. Melia, M., Pahl, C.: Automatic Validation of Learning Object Compositions. In: Information Technology and Telecommunications Conf. IT&T'2005: Doctoral Symposium (2006)
15. 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)