Un nuovo modello per la gestione di versioni di progetto e versioni temporali di schema nelle basi di dati object-oriented

Fabio Grandi[1,2], Federica Mandreoli[1], and Maria Rita Scalas[1,2]

[1] C.S.I.TE.- C.N.R.
[2] D.E.I.S.
University of Bologna, Viale Risorgimento 2,
I-40136 Bologna, Italy
<fgrandi,fmandreoli,mrscalas>@deis.unibo.it

Sommario Il problema della gestione di versioni di schema (*schema versioning*) nelle basi di dati object-oriented è stato studiato nell'ambito di due principali filoni di ricerca. Il primo di essi riguarda sistemi statici (non temporali), per i quali esistono numerose soluzioni per il supporto di versioni progettuali di schema (*versioni consolidate*), sulla base delle esigenze di domini applicativi quali il CAD/CAM e l'ingegneria del software. Il secondo filone di ricerca riguarda invece le basi di dati temporali. In questo ambito, per soddisfare le richieste avanzate da altre tipiche applicazioni object-oriented, quali GIS e multimediale, sono state presentate alcune proposte di gestione di versioni temporali di schema. In questo lavoro ci proponiamo di integrare i due approcci, introducendo un modello generalizzato orientato agli oggetti per la gestione di versioni di schema sia progettuali sia temporali. Il modello proposto estende le possibilità applicative di un singolo sistema arricchendo l'espressività delle versioni e le potenzialità dischiuse dal loro trattamento. A tal fine è stato formalmente definito un insieme completo di primitive per il cambiamento di schema il cui utilizzo sarà esemplificato nel lavoro.

1 Introduzione

La maggior parte delle basi di dati e dei sistemi software hanno strutture complesse soggette in generale a continui cambiamenti. Fra questi rientrano certamente le basi di dati object-oriented che sono stati sviluppati principalmente per modellare applicazioni altamente dinamiche, dove non solo i dati ma anche le loro strutture (*schemi*) subiscono modifiche. Il concetto di versione di schema (*schema version*) nasce dalla necessità di poter mantenere dati aventi strutture diverse, ovvero contemporaneamente consistenti con diversi schemi. In generale, dal punto di vista della semantica delle applicazioni, gestire e memorizzare più versioni di uno schema significa avere a che fare con una delle possibili rappresentazioni della struttura della realtà applicativa modellata.

Le versioni di schema sono state considerate per scopi molteplici in letteratura [7]. Per quanto riguarda le basi di dati object-oriented, in molti settori

applicativi avanzati, quali CAD/CAM e ingegneria del software, si è registrato ben presto il requisito di poter gestire contemporaneamente diverse versioni, anche strutturali, dei dati utili alla progettazione. Nella fattispecie, le versioni di schema rispondono alla necessità di fornire supporto agli aspetti dinamici dei processi di progetto e di ingegnerizzazione. Se lo schema iniziale corrisponde alle idee di base riguardo alla progettazione di un artefatto, ha poi luogo un processo interattivo di revisione delle scelte iniziali. Processo che solitamente coinvolge numerosi utenti in collaborazione o anche in competizione fra di loro. Il prodotto di un simile processo può essere tanto versioni successive quanto anche versioni parallele (alternative di progetto). Scopo finale dell'intero processo è in ogni caso pervenire ad uno schema definitivo da utilizzare nella produzione. In questo campo sono stati pubblicati numerosi lavori (si veda [12] per una breve rassegna di alcuni approcci). Caratteristica comune ai diversi approcci è un'organizzazione delle versioni di schema basata su un grafo (di tipo DAG), in cui le linee di derivazione delle versioni possono divergere (*branching*), in modo da rappresentare scelte alternative, e finanche risultare confluenti (*merging*). Nel seguito faremo pertanto riferimento a questo tipo di approccio come approccio *branching*, e chiameremo *versioni consolidate* le versioni in esso elaborate, rappresentando esse il risultato "stabilizzato" di una sottofase del processo. In tutte queste proposte ogni aspetto temporale è del tutto ignorato.

Recentemente però l'adozione dello schema versioning si è rivelato assai utile anche per altre tipiche applicazioni object-oriented, meno classiche rispetto al "versioning progettuale", come i GIS [3] e la gestione di dati multimediali [14]. Tali applicazioni hanno sovente requisiti temporali, in quanto richiedono che l'evoluzione di alcuni oggetti sia monitorata nel tempo e documentata all'interno del sistema. La gestione di versioni di schema interviene nel momento in cui l'evoluzione degli oggetti coinvolge anche la loro struttura (schema), al di là del semplice cambiamento di valore nei loro attributi. Se, viceversa, nell'ambito delle basi di dati relazionali la gestione di versioni temporali di schema è stata diffusamente studiata [6, 19], sono tuttavia pochissimi i lavori riguardanti le basi di dati object-oriented che la prendono in considerazione. Per primi Cellary et al. [5] propongono un modello formale che contempla versioni di oggetti e schemi ma che, benché metta in risalto l'importanza di rappresentare "stati consecutivi della realtà modellata", non considera alcuna rappresentazione esplicita del tempo e degli aspetti temporali. Nel modello temporale TIGUKAT [8], le storie degli oggetti e degli schemi sono invece trattate in maniera uniforme. Infine, in [9], abbiamo proposto un modello formale OODM$|_{\mathcal{SV}}$, estensione dello standard ODMG 2.0 [4], che consente di rappresentare e gestire versioni temporali di schema.

Scopo principale del presente lavoro è l'integrazione dell'approccio *branching* nell'ambito della gestione di versioni temporali di schema. A tal fine presenteremo un modello di gestione di versioni di schemi (schema versioning) *generale*, in grado di fornire supporto ad entrambe le modalità di definizione delle versioni. La sua definizione sarà basata su OODM$|_{\mathcal{SV}}$ [9], che verrà opportunamente esteso con l'introduzione di *versioni consolidate*. La disponibilità di entrambi i tipi

di versione all'interno di un unico sistema permette di migliorare qualitativamente l'espressività della stessa nozione di versione, aumentando le potenzialità applicative e in definitiva la flessibilità del sistema.

Consideriamo, per esempio, il settore dell'ingegneria e delle attività di progettazione [7, 12, 13, 15] in cui la modalità primaria di gestione delle versioni rimane comunque l'uso di versioni consolidate e la manipolazione delle linee gerarchiche di derivazione delle stesse. Qui l'aggiunta delle versioni temporali può essere utilizzata per modellare la storia di tutti i cambiamenti di schema intermedi applicati ad una versione consolidata prima del rilascio di una nuova versione. Come già evidenziato in [8], questa innovazione aggiunge senz'altro flessibilità al sistema in uso, consentendo una documentazione accurata di ogni scelta effettuata e in definitiva un completo controllo (anche a posteriori o addirittura in fase previsionale) dell'intero processo di progettazione.

D'altro canto, in altri ambiti applicativi (quali quelli dei GIS) la modalità primaria di gestione delle versioni è quella temporale, essendo usata per rappresentare esplicitamente la storia dei cambiamenti strutturali avvenuti nella realtà sottostante. In questo caso si può pensare ad un uso di versioni parallele per analizzare più scenari distinti ad un tempo, come richiesto nelle attività di pianificazione. Si possono introdurre ad esempio versioni parallele per verificare comparativamente le evoluzioni alternative che una porzione di territorio contenente uno stretto marino attualmente servito da linee di traghetti potrà subire in seguito alla costruzione di un ponte sospeso piuttosto che alla realizzazione di un tunnel sottomarino. Da un punto di vista prettamente tecnico, ciò significa consentire la biforcazione dell'asse temporale corrispondente al tempo di validità, supporto necessario all'esplorazione di corsi degli eventi ipotetici e alternativi (nel passato come nel futuro) ma impraticabile nei sistemi con sole versioni temporali, nei quali esiste un'unica linea temporale.

In ogni caso, le due modalità di gestione delle versioni di schema sono fra loro complementari: mentre la gestione delle versioni consolidate è una modalità a grana grossa, quello temporale si basa su un meccanismo a grana più fine.

Il resto del lavoro è organizzato come segue. Il Paragrafo 2 introduce una nuova nozione di base di dati capace di supportare il processo di sviluppo e manutenzione delle diverse versioni di schema. Nel Paragrafo 3 presentiamo l'approccio adottato per la manipolazione delle versioni di schema accompagnato da un esempio significativo del loro impiego. Il Paragrafo 4 è dedicato alle conclusioni e a prospettive future.

1.1 Un richiamo al modello OODM$|_{\mathcal{SV}}$

OODM$|_{\mathcal{SV}}$ [9] è un modello formale per la gestione di versioni temporali di schema nelle basi di dati object-oriented. La sua definizione parte dalla Release 2.0 del modello ODMG [4]. La componente di base su cui la sua architettura si fonda è la *versione temporale* (Temporal Version $\mathcal{TV}$) che rappresenta uno stato del mondo temporalmente omogeneo, composto da una *versione di schema* (Schema Version $\mathcal{SV}$) e dalla corrispondente *versione di dati memorizzati* (Stored Data Version $\mathcal{SDV}$). La versione di schema consta di tutte le necessarie definizioni strutturali

(ovvero interfacce, classi, operazioni, etc.). La corrispondente versione di dati memorizzati contiene invece le istanze di tutti i tipi definiti nella versione di schema.

In [9] e nel presente lavoro consideriamo un database *bitemporale*. Ogni versione è dotata di una propria *pertinenza temporale*, che consiste in un sottoinsieme del dominio bitemporale ottenuto come prodotto cartesiano dei domini di tempo di transazione (transaction time) e di tempo di validità (valid time). Il tempo (come si conviene nelle basi di dati temporali) può assumere valori in un insieme discreto $\mathcal{TIME} = \{0, 1, \ldots, now, \ldots, \infty\}$ di *chronon* [11]. Essendo la semantica del dominio $\mathcal{TIME}$ differente a seconda che si consideri il tempo di transazione o di validità [17], distingueremo nel seguito $\mathcal{TIME}_t = \{0, 1, \ldots, now, \ldots, \infty\}_t$ da $\mathcal{TIME}_v = \{0, 1, \ldots, now, \ldots, \infty\}_v$, con ovvio significato dei pedici. L'associazione tra le versioni e la loro pertinenza temporale è garantita dal concetto di *database bitemporale*, definito come una funzione che mappa il dominio del tempo in versioni temporali.

L'idea di *contestualizzazione* sta alla base di tutto il modello. In OODM$|_{\mathcal{SV}}$ infatti ogni elemento il cui significato non sia limitato ad una versione temporale specifica è definito globalmente a livello di database. Ciò accade ai tipi e ai loro valori legali e all'insieme di tutti i nomi disponibili, come i nomi di interfaccia (denotato da $\mathcal{IN}$). Una qualunque versione temporale è quindi formata da elementi scelti fra quelli globalmente definiti a livello di database. La contestualizzazione riguarda l'insieme dei nomi di interfaccia disponibili in una particolare versione temporale. La composizione di tale insieme è infatti dinamica: cresce ogni qualvolta viene creata una nuova versione di schema grazie all'aggiunta di un nome di interfaccia e cala ogni qualvolta viene cancellata un'interfaccia esistente. Sulla base dei nomi di interfaccia disponibili in una data versione temporale, detta versione dei nomi di interfaccia (Interface Name Version, $\mathcal{INV}$), rimane definito l'insieme dei tipi ($\mathcal{T}|_{\mathcal{INV}}$) e dei loro valori legali oltre alla gerarchia delle versioni di interfaccia. Si noti che il modello dei dati di ODMG 2.0 contempla, fra le specificazioni di tipi, la nozione di classe (*class*) distinta dalla nozione di interfaccia (*interface*). Una interfaccia definisce solo il comportamento astratto del tipo di un oggetto mentre una classe ne definisce sia il comportamento che lo stato astratto. Dato che a metalivello il tipo *Class* é un sottotipo di *Interface*, in OODM$|_{\mathcal{SV}}$ si preferisce utilizzare un unico termine "interface" per denotare interfacce proprie (tipi senza estensione, nella terminologia object-oriented comune) o classi (tipi con estensione). Per questo motivo le modifiche di schema agenti sulle classi sono assimilate alle modifiche agenti sulle interfacce e vengono realizzate tramite operatori comuni (**AddInterface**, **DeleteInterface** e **ChangeInterfaceName**).

2 Un meccanismo generalizzato per la gestione di versioni

La definizione di database inclusa nel nostro modello è il frutto di una rivisitazione della nozione di database solitamente considerata in un sistema statico (non temporale) [1]. Quella che qui proponiamo consente di assistere in maniera

completa il processo di sviluppo e manutenzione dello schema permettendo sia versioni consolidate che temporali. A tal fine consideriamo uno scenario in cui siano memorizzate diverse versioni consolidate, con la possibilità di derivarne una nuova a partire da quelle già esistenti. Un database risulterà quindi un grafo (di tipo DAG) composto da un insieme di nodi e di archi che corrispondono rispettivamente alle versioni consolidate e alle relazioni gerarchiche di derivazione. La radice del grafo è costituita dalla prima versione consolidata creata dall'utente. Il nostro modello consente, per ogni versione consolidata, di tenere traccia di tutti i cambiamenti di schema "locali" grazie alla gestione di versioni bitemporali di schema dove la versione temporale rappresenta la *granularità* del versioning [9].

Adotteremo in questo contesto un metodo di assegnazione di nomi definiti dall'utente per associare etichette simboliche (label) alle versioni consolidate. Le versioni temporali di schema saranno quindi univocamente identificate dalla combinazione di questo metodo di denominazione con l'uso di un *timestamp* per la selezione temporale nel contesto di una versione consolidata. La Figura 1

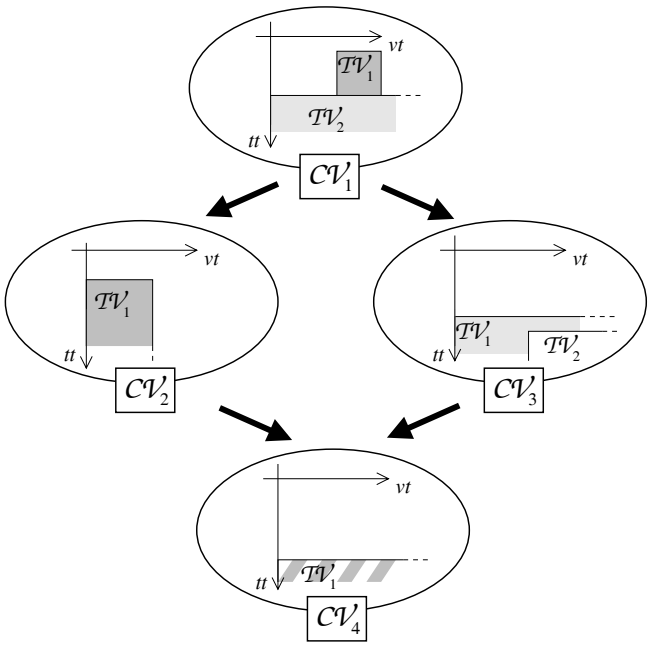

Figura1. Un esempio di database

mostra un database composto da quattro versioni consolidate rappresentate come ovali (le label sono $\mathcal{CV}_1, \ldots, \mathcal{CV}_4$). La storia di ogni versione consolidata è mantenuta tramite l'uso di versioni temporali opportunamente posizionate lungo

degli assi di tempo di transazione/validità "privati", raffigurati all'interno degli ovali.

Definizione 1 (Database) *Sia $\mathcal{TVS}$ l'insieme di tutte le possibili Versioni Temporali $\mathcal{TV}$ e sia $\mathcal{L}$ un insieme di label per le versioni consolidate. Un Database è un DAG denominato $\mathcal{DBG} = (\mathcal{DB}, \mathcal{E})$ in cui $\mathcal{E} \subseteq \mathcal{L} \times \mathcal{L}$ e $\mathcal{DB}$ è una funzione così definita:*

$$\mathcal{DB} : (\mathcal{L} \times \mathcal{TIME}_t \times \mathcal{TIME}_v) \rightarrow \mathcal{TVS} \cup \{\emptyset\}$$

che associa ciascuna ennupla composta da una label e da un chronon bitemporale ad una versione temporale, se questa esiste, ossia:

$$\mathcal{DB}\,(l, tt, vt) = \begin{cases} \mathcal{TV} \text{ se } \exists \mathcal{TV} \text{ nella versione} \\ \quad \text{consolidata con label } l \\ \quad \text{valida in } tt \text{ e } vt \\ \emptyset \quad \text{altrimenti} \end{cases}$$

ove $l \in \mathcal{L}$, $tt \in \mathcal{TIME}_t$, $vt \in \mathcal{TIME}_v$.

Strumenti per l'accesso ad una particolare versione possono essere facilmente derivati sulla base della definizione precedente. Applicando la definizione della funzione $\mathcal{DB}$ ad una label l si può infatti ottenere il database bitemporale associato alla versione consolidata referenziata con l. Tale risultato rappresenta l'intera storia dei cambiamenti successivamente applicati allo schema definito con la creazione della versione consolidata. Questo schema è senz'altro quello della versione temporale col tempo di transazione più antico, che è la prima versione temporale ad essere stata introdotta nella versione consolidata.

3 Gestione di versioni come supporto all'evoluzione dello schema

Le ragioni che inducono all'introduzione di una nuova versione di schema sono differenti e dipendono dal significato associato al concetto di versione stessa. In alcuni casi, come nel campo dell'ingegneria e delle attività di progettazione, l'introduzione di una nuova versione consolidata è dettata dalla necessità di disporre di una nuova versione "completa" e "stabile" la cui definizione è basata su differenti assunzioni o criteri di progettazione. Essa può anche nascere dall'integrazione di versioni consolidate già esistenti. Questo processo di creazione è completamente gestito dal progettista che decide quando introdurre una nuova versione e come questa deve essere composta. In altri casi, come nelle basi di dati temporali, una nuova versione viene creata ogni qualvolta è richiesta una modifica allo schema, grazie ad una politica di aggiornamento non distruttiva. La nuova versione, che può riflettere una versione precedente o successiva della struttura della realtà modellata, entra a far parte della memorizzazione della storia di tutte le modifiche effettuate. In questo caso, il sistema gestisce in modo

Tabella1. Lista delle primitive per il cambiamento di schema supportate nel modello

Modifiche di schema agenti sui nodi $\mathcal{SC}_{on\_node}$			
sugli attributi	**AddAttribute**	sulle relationship	**AddRelationship**
	DeleteAttribute		**DeleteRelationship**
	ChangeAttrName		**ChangeRelName**
	ChangeAttrType		**ChangeRelType**
sulle operazioni	**AddOperation**	sulle eccezioni	**AddException**
	DeleteOperation		**DeleteException**
	ChangeOpName		**ChangeExceName**
	ChangeOpCode		**ChangeExceType**
	AddExceToOp		
	DeleteExceFromOp		
sulle relazioni gerarchiche	**AddSuperinterface**	sulla collezione dei tipi	**AddInterface**
	DeleteSuperinterface		**DeleteInterface**
			ChangeInterfaceName
Modifiche di schema agenti sugli archi $\mathcal{SC}_{on\_edge}$			
di tipo "Pick" $\mathcal{SC}_{pick}$	**PickAttribute**	di tipo "New"	**NewNode**
	PickRelationship		**NewEdge**
	PickOperation		
	PickException	di tipo "Merge" $\mathcal{SC}_{merge}$	**MergeVersion**
	PickInterface		

trasparente all'utente il tempo di transazione mentre il progettista specifica la validità (valid-time element [11]) delle nuove versioni. Il nostro modello fornisce supporto per l'introduzione e la gestione di versioni per tutte, e non solo, le ragioni fin qui elencate attraverso la manipolazione di versioni consolidate e temporali. Definiamo, quindi, un insieme di *primitive per le modifiche di schema* che consente all'utente di eseguire le seguenti operazioni:

1. aggiornare una versione consolidata mediante l'applicazione di modifiche di schema mantenendo la storia di tutte le versioni passate e future;
2. generare una nuova versione consolidata;
3. integrare in una versione consolidata caratteristiche di altre versioni consolidate;
4. aggiungere una derivazione semantica fra versioni consolidate.

La lista delle primitive proposte include l'insieme completo delle operazioni usualmente considerate per modificare gli elementi del modello dei dati adottato [2], oltre ad alcune operazioni speciali per la gestione diretta della gerarchia di derivazione degli schemi [16]. Sulla base di questa distinzione, in Tabella 1 mostriamo l'elenco delle primitive per il cambiamento di schema suddiviso in due categorie: "Modifiche di schema agenti sui nodi" ($\mathcal{SC}_{on\_node}$) e "Modifiche di schema agenti sugli archi" ($\mathcal{SC}_{on\_edge}$). I nomi delle due categorie hanno origine dalle diverse operazioni applicabili ad un DAG che, nel nostro modello, rappresenta il reticolo delle versioni consolidate.

La collezione delle modifiche agenti sui nodi coincide con l'insieme completo delle modifiche applicabili ad uno schema basato su ODMG. Esse consentono l'aggiornamento delle versioni consolidate sulla base di una politica di gestione delle versioni temporali, così come proposto in [9]. Di conseguenza, l'applicazione di una di queste primitive comporta l'aggiunta di una nuova versione temporale alla versione consolidata in questione (nessuna nuova versione consolidata è generata). Alla nuova versione temporale viene associata una pertinenza temporale (ovvero un timestamp) utilizzabile in seguito per la selezione degli schemi. E' così possibile accedere ai dati attraverso differenti versioni temporali di schema (passate, presenti e future). Tutti gli altri meccanismi per la gestione delle versioni (si veda l'elenco sopraindicato) sono inclusi nella collezione delle modifiche agenti sugli archi e sono necessari per un controllo delle versioni guidato dall'utente. Per essi, introduciamo una ulteriore distinzione basata sul tipo di cambiamento:

- le *modifiche di tipo "new"* permettono di introdurre una nuova versione consolidata (**NewNode**) o una nuova relazione di derivazione nella gerarchia (**NewEdge**);
- le *modifiche di tipo "pick"* consentono di integrare in una versione consolidata caratteristiche di un'altra versione. La principale differenza fra le modifiche di schema di tipo "pick" e le corrispondenti modifiche di tipo "add" incluse nelle modifiche applicabili ai nodi (ad es. **PickAttribute** vs. **AddAttribute**) è che le prime considerano elementi già popolati da valori che vengono quindi ereditati dalla versione ricevente, mentre le seconde semplicemente aggiungono nuovi elementi con associati valori nulli;
- l'unica *modifica di tipo "merge"* integra versioni consolidate esistenti in una nuova versione. Integrare due o più versioni consolidate significa combinarle considerando anche i dati in esse contenuti.

Come già evidenziato in [16], la propagazione delle modifiche agli oggetti è un aspetto importante della gestione delle versioni. A differenza del lavoro citato, nel presente lavoro consentiamo che due versioni dello stesso oggetto possano essere fuse in una nuova versione. In generale, un oggetto, referenziato attraverso il suo OID, è rappresentato da più di una versione nel DAG e (in analogia con i database "standard") una versione temporale può memorizzare al più una versione dello stesso oggetto. Quindi, quando due versioni consolidate che memorizzano due versioni dello stesso oggetto devono essere fuse, è necessario approdare ad un'unica versione dell'oggetto. Tale versione deve possibilmente

ereditare il maggior numero di informazioni contenute nelle versioni da cui ha origine.

Si supponga, ad esempio, che un'industria aeronautica sia interessata a progettare un nuovo tipo di aereo. Il primo passo consiste nel definire una bozza della struttura (o schema) dell'aereo le cui istanze sono bozze di aerei. Quindi, diversi gruppi lavoreranno a differenti parti dell'aereo come la fusoliera e i motori. Ogni gruppo definirà la struttura della parte assegnata migliorando così le bozze dell'aereo con le istanze delle parti sviluppate. Al termine di questo processo, lo schema finale per l'aereo sarà rappresentato dall'integrazione delle diverse parti sviluppate dai singoli gruppi. Ciò che un progettista si aspetta da un "buon" sistema di schema versioning è che il progetto finale dell'aereo sia assemblato a partire dallo schema scheletro definito nella prima bozza, dove possano trovar posto le istanze delle singole parti sviluppate dai singoli gruppi.

3.1 Semantica delle modifiche di schema

Introduciamo in questo sottoparagrafo la semantica delle modifiche di schema elencate in Tabella 1. Il versioning bitemporale di schema garantito dal nostro modello assicura che ogni primitiva per il cambiamento di schema non operi seguendo una politica di "update-in-place" ma manipoli e produca sempre versioni temporali. Sebbene questo approccio sia quello usualmente adottato nel contesto temporale, esso rappresenta un aspetto innovativo nel branching. Infatti, le versioni consolidate solitamente gestite nel contesto branching sono di tipo "istantanea" (snapshot), per le quali non viene mantenuta alcuna storia. Nel nostro modello, le versioni temporali gestite a granularità più fine permettono di mantenere nell'ambito delle versioni consolidate la storia completa delle modifiche di schema.

Ogni modifica di schema coinvolge versioni consolidate; fra queste distinguiamo una o più versioni *sorgenti* (già esistenti) da una versione *destinataria* (che può anche non esistere in precedenza). Nel nostro approccio una delle versioni sorgenti coincide sempre con la versione destinataria eccetto per il caso della creazione di una nuova versione consolidata. Adottando questa soluzione, esiste solo una specifica primitiva che consente di creare una nuova versione, mentre tutte le altre primitive agiscono come "aggiornamenti" della versione destinataria (che è coinvolta dall'integrazione con le altre sorgenti). L'esecuzione di una primitiva per la modifica di schema può essere riassunta nei seguenti passi:

1. per ogni versione sorgente, viene selezionata una delle sue versioni temporali *correnti*. A tale fine, l'utente specifica un parametro di validità chiamato *schema selection validity*;
2. viene generata una nuova versione temporale applicando la primitiva richiesta alle versioni temporali selezionate;
3. viene aggiornata (o generata) la versione destinataria includendo la nuova versione temporale con la relativa pertinenza temporale. Per quanto riguarda quest'ultima, la validità della nuova versione viene specificata esplicitamente dall'utente attraverso lo *schema change validity*, che è un valid-time element

[11], mentre la pertinenza del tempo di transazione è definita dal sistema come l'intervallo $[now, \infty]$.

Nel nostro modello, la formalizzazione di questi passi è demandata a tre funzioni: la funzione di aggiornamento del database (DataBase Update function - $\mathcal{DBU}$), la funzione di aggiornamento delle versioni temporali (Temporal Version Update function - $\mathcal{TVU}$) e la funzione di fusione (*merge*). Ogni nuova versione temporale deve soddisfare i seguenti requisiti:

- la sua versione di schema deve essere uno schema corretto [2]. In [9] abbiamo dimostrato che il nostro modello è riducibile ad un modello assiomatico [18] adottato nello "schema evolution" per garantire la consistenza dello schema risultante;
- la sua versione di dati memorizzati deve essere consistente rispetto ai corrispondenti tipi definiti nella versione di schema. A tale scopo, nelle funzioni $\mathcal{TVU}$ e *merge* è incorporato un meccanismo di propagazione ai dati.

La specifica completa delle funzioni $\mathcal{DBU}$,$\mathcal{TVU}$ e *merge* si può trovare in una versione estesa di questo lavoro [10]. In breve, la funzione $\mathcal{DBU}$ viene applicata qualunque sia la primitiva per il cambiamento di schema eseguita e trasforma il DAG del database in uno nuovo. A seconda dei casi, essa utilizza la sola funzione $\mathcal{TVU}$ (per le "modifiche agenti sui nodi") o entrambe le funzioni $\mathcal{TVU}$ e *merge* (per le "modifiche agenti sugli archi") per costruire la nuova versione e collocarla nel DAG.

3.2 Un esempio completo

Si consideri l'esempio dell'industria aeronautica introdotto nel Paragrafo 3. Durante la prima fase, viene definito lo scheletro dell'aereo che si vuole progettare. A tale scopo, la versione "radice" potrebbe chiamarsi `draft` ed includere una versione temporale valida dal 1940, il cui schema è composto da una classe chiamata `aircraft` (si veda la Figura 2).

Quindi, la progettazione dell'aereo potrebbe essere assegnata a tre gruppi che sviluppano separatamente i motori (`engine`), la cabina di pilotaggio (`cockpit`) e la fusoliera (`fuselage`). La modifica di schema da applicare per tre volte è la primitiva **NewNode** con `draft` come versione sorgente e $[98, \infty]$ come validità delle nuove versioni (si veda la Figura 3).

La struttura dell'aereo viene così arricchita da ogni gruppo attraverso lo sviluppo della parte assegnata. Ciò è possibile applicando le modifiche di schema agenti sui nodi, come **AddAttribute** o **AddRelationship** per definire nuove proprietà, o **AddInterface** per aggiungere nuovi elementi. Un esempio è mostrato in Figura 4 dove, nella versione consolidata `engine`, alla versione temporale corrente valida nel 1999 viene aggiunta la classe `Engine`. La nuova versione temporale ha validità assegnata $[40, \infty]$. In ogni versione consolidata, ogni versione temporale rappresenta una versione della struttura della parte assegnata con le sue parti-oggetto (le instanze). I valid-time element associati alle versioni temporali rappresentano il tempo in cui le versioni della struttura di quella parte era, è o sarà valida nella realtà.

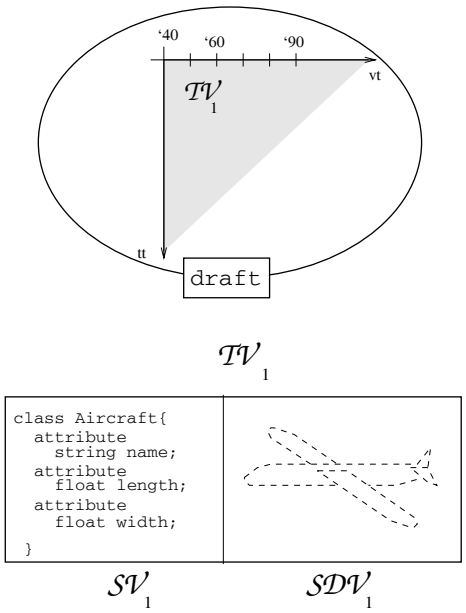

Figura2. La versione "radice"

$\mathcal{DBU}$(**NewNode**)(DBG, draft, , engine, ,['98, ∞]}
$\mathcal{DBU}$(**NewNode**)(DBG, draft, , cockpit, ,['98, ∞]}
$\mathcal{DBU}$(**NewNode**)(DBG, draft, , fuselage, ,['98, ∞]}

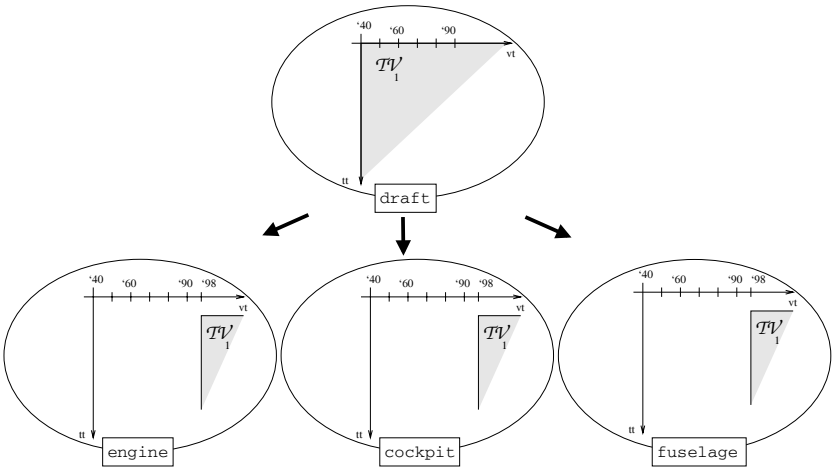

Figura3. Derivazione di tre versioni

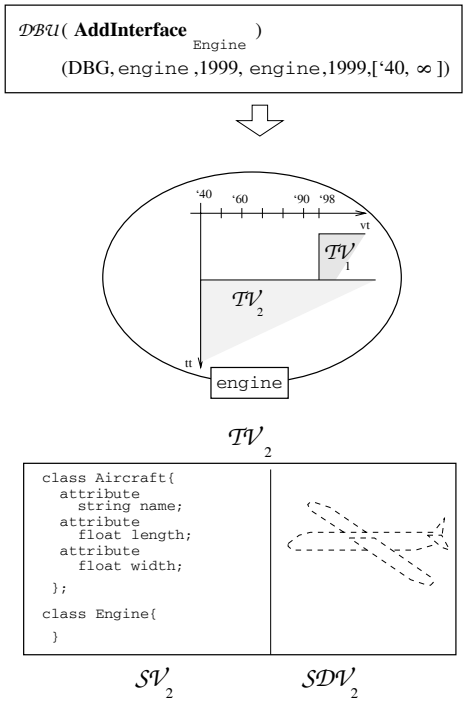

Figura4. Applicazione di una modifica ad un nodo

Lo schema finale dell'aereo è ottenuto "riempiendo" lo scheletro definito nella prima fase con le strutture delle singole parti. Ognuna di queste è una versione di schema della corrispondente versione consolidata (ed es. `engine` per il motore) selezionata attraverso lo schema selection validity (ad es. il motore del 1997). Questo compito è realizzato mediante i seguenti passi (si veda la Figura 5):

- applica la primitiva **NewNode** ad una qualsiasi delle tre parti (ovvero versione consolidata). Viene così introdotta, nella nuova versione consolidata (`aircraft`), una copia della versione temporale selezionata;
- per ognuna delle rimanenti parti, applica la primitiva **MergeVersion** per integrarla nel progetto dell'aereo.

Per ulteriori dettagli (es. per spiegazioni sui parametri delle primitive) si rimanda a [10].

4 Conclusioni e prospettive future

In questo lavoro abbiamo proposto un modello per la gestione di versioni che integra la gestione di versioni puramente temporale con l'approccio branching. La nostra proposta è basata su una nozione di database che supporta completamente il processo di sviluppo e manutenzione dello schema permettendo la gestione

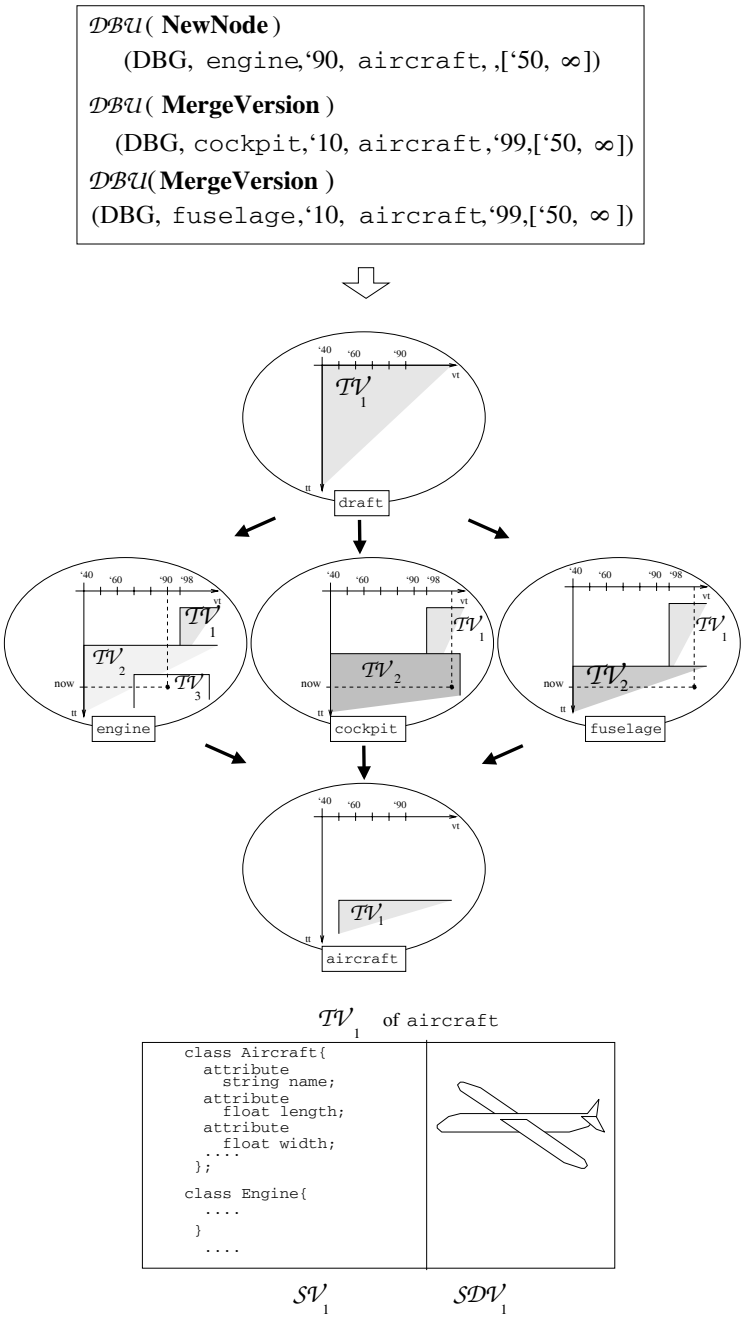

Figura5. Fusione di versioni

di versioni consolidate e temporali. Il modello è stato arricchito definendo un insieme completo di modifiche di schema per la manipolazione delle versioni. In questo modo, il nostro modello è utilizzabile per l'organizzazione delle versioni per tutte le esigenze applicative nei database object-oriented evidenziate in letteratura. Con l'aiuto di alcuni esempi, abbiamo infatti dimostrato come alcuni interessanti domini di applicazione possano beneficiare del supporto congiunto di schema versioning temporale e branching.

In futuro, intendiamo approfondire lo studio degli aspetti semantici inerenti l'integrazione di versioni. Infine, estenderemo il nostro modello considerando anche altre dimensioni per il versioning (ad es. coordinate spaziali [20]).

Riferimenti bibliografici

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, Reading, MA, 1995.
2. J. Banerjee, W. Kim, H.-J. Kim, and H. F. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In *Proc. of the ACM-SIGMOD Annual Conference*, pages 311–322, San Francisco, CA, May 1987.
3. P. A. Burrough and R. A. McDonnell. *Principles of Geographical Information Systems.* Oxford University Press, New York, NY, 1998.
4. R. G. G. Cattell, D. Barry, D. Bartels, M. Berler, J. Eastman, S. Gamerman, D. Jordan, A. Springer, H. Strickland, and D. Ware, editors. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann, San Francisco, CA, 1997.
5. W. Cellary, G. Jomier, and T. Koszlajda. Formal Model of an Object-Oriented Database with Versioned Objects and Schema. In *Proc. of the 2nd Int'l Conf. on Database and Expert Systems Applications (DEXA)*, pages 239–244, Berlin, Germany, 1991.
6. C. De Castro, F. Grandi, and M. R. Scalas. Schema Versioning for Multitemporal Relational Databases. *Information Systems*, 22(5):249–290, 1997.
7. K. R. Dittrich and R. A. Lorie. Version Support for Engineering Database Systems. *IEEE Transactions on Software Engineering*, 14(4):429–436, 1988.
8. I. A. Goralwalla, D. Szafron, M. T. Özsu, and R. J. Peters. A Temporal Approach to Managing Schema Evolution in Object Database Systems. *Data & Knowledge Engineering*, 28(1):73–105, 1998.
9. F. Grandi, F. Mandreoli, and M. R. Scalas. A Formal Model for Schema Versioning in Object-Orieted Databases. Technical Report CSITE-014-98, CSITE - CNR, November 1998.
10. F. Grandi, F. Mandreoli, and M. R. Scalas. Supporting design and temporal versions: a new model for schema versioning in object-oriented databases. Submitted for publication, 1999.
11. C. S. Jensen, J. Clifford, S. K. Gadia, P. Hayes, and S. Jajodia et al. The Consensus Glossary of Temporal Database Concepts - February 1998 Version. In O. Etzion, S. Jajodia, and S. Sripada, editors, *Temporal Databases - Research and Practice*, pages 367–405. Springer-Verlag, 1998. LNCS No. 1399.
12. R. H. Katz. Toward a Unified Framework for Version Modeling in Engineering Databases. *ACM Computing Surveys*, 22(4):375–408, 1990.
13. R. H. Katz and T. J. Lehman. Database Support for Versions and Alternative of Large Design Files. *IEEE Transactions on Software Engineering*, 10(2):191–200, 1984.

14. S. Khoshafian and A. B. Baker. *MultiMedia and Imaging Databases.* Morgan Kaufmann, San Francisco, CA, 1996.
15. G. S. Landis. Design Evolution and History in an Object-Oriented CAD/CAM Database. In *Proc. of 31st COMPCON Conference*, San Francisco, CA, March 1986.
16. S.-E. Lautemann. A Propagation Mechanism for Populated Schema Versions. In *Proc. of the 13th International Conference on Data Engineering (ICDE)*, pages 67–78, Birmingham, U.K., April 1997.
17. G. Özsoyoğlu and R. T. Snodgrass. Temporal and Real Time Databases: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, 7(4):513–532, 1995.
18. R. J. Peters and M. T. Özsu. An Axiomatic Model of Dynamic Schema Evolution in Objectbase Systems. *ACM Transaction on Database Systems*, 22(1):75–114, 1997.
19. J. F. Roddick. A Survey of Schema Versioning Issues for Database Systems. *Information and Software Technology*, 37(7):383–393, 1996.
20. J. F. Roddick, F. Grandi, F. Mandreoli, and M. R. Scalas. Towards a Model for Spatio-Temporal Schema Selection. Accepted at the STDML Workshop (in conj. with DEXA'99), 1999.