Evoluzioni di Ontologie in Frame Logic

Francesco Mele[1], Antonio Sorgente[1], Giuseppe Vettigli[1]
[1] C.N.R. Istituto di Cibernetica "E. Caianiello", Via Campi Flegrei, 34 – Pozzuoli, Naples, Italy.
{f.mele, a.sorgente, g.vettigli}@cib.na.cnr.it

Abstract. In quest'articolo presentiamo una metodologia e un framework per le evoluzioni di ontologie definite in Frame Logic. Il sistema definito permette di valutare operazioni di modifica di un'ontologia al fine di garantire la consistenza dell'ontologia stessa. Il framework esegue controlli mediante un insieme di operazioni di base e inoltre è in grado di eseguire operazioni di controllo su regole evolutive definite dall'utente per uno specifico dominio.

Keywords: Frame Logic, Ontologie, Evoluzioni di Ontologie

1 Introduzione

Ogni qualvolta è modificata un'ontologia si eseguono delle operazioni di ridefinizione, aggiunta o eliminazione di classi, d'istanze e di relazioni fra tali entità. Tali modifiche possono generare delle inconsistenze di vario tipo che devono essere eliminate per assicurare l'*integrità* della rappresentazione stessa. I cambiamenti delle entità dell'ontologia appartengono a un processo che si sviluppa in un tempo, fisico o convenzionale, che attraversa vari stadi, in ognuno dei quali possono presentarsi delle inconsistenze. Un processo di trasformazione di un'ontologia nel tempo, *O(t), O(t+1), O(t+2),…,O(t+n)*, del tipo descritto, viene detto evoluzione.

In questo lavoro ci occupiamo del controllo dell'evoluzione di un'ontologia, attività che consiste nell'individuare inconsistenze e della loro eliminazione.

Le metodologie attuali prevedono tre tipi di controlli di consistenza [3,6]:

1. consistenza strutturale, la quale assicura che l'ontologia sia definita rispetto al modello di riferimento adottato e ai costrutti del linguaggio di rappresentazione scelto;
2. consistenza logica, la quale assicura che un'ontologia sia semanticamente corretta, ad esempio che non contenga informazioni contraddittorie;
3. consistenza definita dall'utente, quest'ultima assicura che l'ontologia sia consistente rispetto a particolari regole di modifiche (regole evolutive) definite dall'utente.

Il primo tipo di controllo di consistenza è usato per garantire che le operazioni di modifica, eseguite sull'ontologia corrente *O(t)*, conservino una certa espressività di rappresentazione al fine di mantenere lo stesso livello di complessità computazionale dell'ontologia di partenza *O(t)* (alcuni classificatori come quello della versione

standard di OWL [5] si limita a segnalare al programmatore che è stata aumentata o diminuita l'espressività linguistica).

Il controllo di consistenza logica, invece, garantisce che dopo una modifica, l'ontologia non presenti delle contraddizioni di tipo logico. Ad esempio se in un'ontologia due concetti `A` e `B` sono definiti disgiunti, allora non può essere inserito un oggetto `I` che è sia istanza di `A` che di `B`. I tipi di controllo 1. e 2. sono indipendenti dal tipo di conoscenza

Agli approcci di controllo di consistenza dipendenti dal dominio di conoscenza (3. della lista), invece, appartengono quelli di consistenza definiti dall'utente. Questi ultimi a loro volta si dividono in controlli generici e controlli dipendenti dal dominio. Quelli generici si rappresentano mediante vincoli che sono utilizzati per definire best-practices per la definizione di ontologie. Appartengono a questa categoria i vincoli definiti nella metodologia OntoClean [2] per le nozioni di rigidità, d'identità e di unità. I controlli di consistenza, definiti dall'utente dipendenti dal dominio, sono definiti mediante una data politica di modifica in un dominio specifico. Ad esempio si possono definire delle limitazioni sulla creazione di sottoclassi imponendo che ogni sottoclasse di una classe `A` abbia almeno un nuovo attributo rispetto alla classe `A`.

2 Fasi di uno stadio evolutivo

In questo lavoro ci siamo occupati in particolare dei controlli di consistenza definiti dall'utente: la loro rappresentazione, come scoprire e risolvere inconsistenze.

Nell'approccio che proponiamo, vi è una separazione delle attività di formulazione (la proposta dell'evoluzione) da quelle di controllo della consistenza dell'evoluzione: la formulazione evolutiva avviene per opera di un operatore umano, che per scopi di revisione delle conoscenze, elimina o inserisce nuove entità nell'ontologia, mentre il controllo della consistenza avviene da parte di un sistema software che agisce con regole indipendenti o dipendenti dal dominio (in avanti forniremo in dettaglio i tipi regole e come esse sono formulate).

Il processo di valutazione di un'evoluzione (Fig. 1) inizia dopo che l'utente ha definito delle operazioni di modifica (*Proposte di modifica*) sull'ontologia *O(t)* (ontologia consistente al tempo *t*). Dalla prassi comune dei processi di creazione e modifica di ontologie, si possono individuare un insieme di operazioni di base le quali permettono di aggiungere e rimuovere: una classe; una relazione tassonomica tra le classi; un attributo e sue proprietà; un'istanza di una classe.

Dopo che le modifiche sono state eseguite sull'ontologia corrente (*O(t)*), viene definita una versione aggiornata (temporanea) dell'ontologia (*O(t+1)*). Su *O(t+1)* sono eseguiti i controlli di consistenza previsti per ciascun tipo di operazione di modifica.

Se nell'ontologia sono scoperte delle inconsistenze, allora inizia il processo di risoluzione di esse (*Risoluzione delle inconsistenze*). Il sistema cerca di risolvere le inconsistenze, prima automaticamente, mediante regole che rimuovono l'inconsistenza in relazione ad alcuni frequenti errori di revisione dei programmatori (si veda in paragrafo 3.3), poi interagendo con il programmatore stesso. In quest'ultimo caso il sistema mostra l'inconsistenza riscontrata e le scelte possibili per

la risoluzione. Risolte le inconsistenze, le modifiche diventeranno effettive e viene confermata la nuova versione *O(t+1)* dell'ontologia. Per ogni passaggio evolutivo da *O(t)* ad *O(t+1)* sono annotate tutte le operazioni eseguite sull'ontologia, sia quelle definite dall'utente, sia le operazioni eseguite dal sistema per la risoluzione delle inconsistenze.

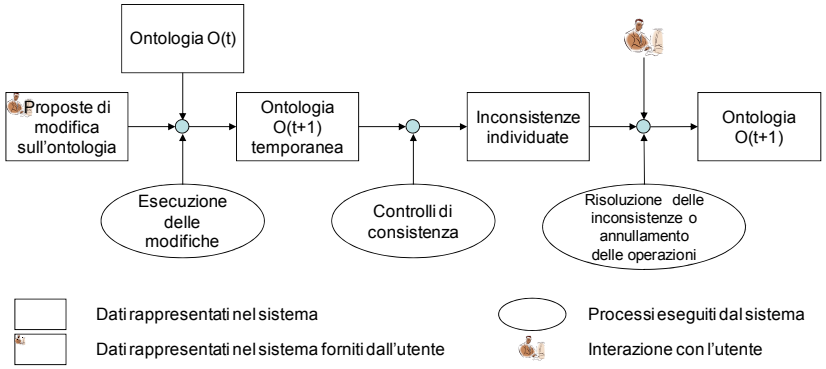

Fig. 1.: Fasi di uno stadio evolutivo

Il linguaggio di riferimento per l'implementazione del sistema è quella della Frame-Logic[4], in particolare è stato utilizzato il linguaggio Flora2 [1]. Per la lettura dei programmi presentati, elenchiamo alcuni costrutti base del linguaggio Flora2. *X::Y* (la classe *X* è sottoclasse della classe *Y*), *X:Y* (*X* è un'istanza della classe *Y*), *X =>Y* (*X* è un attributo di tipo *Y*), *X->Y* (*Y* è il valore dell'attributo *X*), *X *=>Y* (come *X =>Y* e l'attributo è ereditato dalle sottoclassi). In Flora2, qualsiasi concatenazione di letterali alfanumerici, inizianti con un carattere, preceduti dal simbolo *?* rappresentano variabili(*?X*, *?aY*, *?a1*, etc..). I simboli ":-", la virgola (",") e il punto e virgola (";") ha la stessa interpretazione degli omologhi costrutti prolog.

3 Regole di evoluzione in Frame Logic

Le operazioni di revisione di un'ontologia sono rappresentate nel sistema mediante regole Flora2. Un esempio di operazione è la seguente:

```
addSlot(?C,?SName,?SType,?CMin,?CMax,true,?User,?Mod):-
        insert{(?C[?SName{?CMin:? Max}*=>?SType])@?Mod}.
```

La regola definisce l'operazione `addSlot` che aggiunge alla classe `?C` lo slot `?SName` di tipo `?SType` con cardinalità minima `?Cmin` e massima `?CMax`. La costante `true` indica che lo slot è ereditato dalle sottoclassi di `?C` (`false` altrimenti). `?User` è l'utente che ha eseguito l'operazione. `?Mod` stabilisce se l'operazione di revisione è definitiva per il sistema.

La richiesta di un utente `U` di aggiungere alla classe `C` uno slot `S` di tipo `T` con cardinalità minima `1` e massima `2`, è rappresentato nel seguente modo:

```
ricUt(U,AddSlot,[C,S,T,1,2,true]).
```

Per l'esecuzione di ogni operazione è stata definita una funzione che individua le richieste di modifica dell'ontologia (analizzando i fatti `ricUt/3`) e attiva le operazioni di revisione corrispondente. La definizione della funzione è la seguente:

```
excute_op:-                                                    (1)
   ricUt(?User,?OpName,?Pars),                                 (2)
   def_op(?User,?OpName,?Pars,?OP),                            (3)
   ?OP=..[hilog(addSlot),?Pars,?User]
   call(?OP),                                                  (4)
   write_evoluzioni(?OP).                                      (5)
```

La funzione recupera i dati relative alle richieste utente (passo 2), `def_op/4` costruisce la chiamata all'operazione e la mette in `?OP` (conterrà un'invocazione di tipo `addSlot/8`), successivamente la invoca (passo 4) e infine immagazzina nella struttura *Evoluzioni* l'operazione eseguita (passo 5).

3.2 Controllo della consistenza

Nell'approccio adottato, per ogni operazione di richiesta di modifica è associato uno o più controlli di consistenza. Sempre in relazione all'operazione di aggiungere uno slot ad una classe, dopo il suo inserimento nell'ontologia temporanea una funzione `check/1` è attivata per eseguire il controllo di consistenza:

```
check(addSlot):-                                               (1)
  ricUt(?User,                                                 (2)
       addSlot,[(?C,?SName,?SType,?CMin,?CMax,?Inh]),
  ?Check=..[hilog(slotCheck, modC),                            (3)
                ?C,?SName,?SType,?CMin,?CMax],
  call(?Check).                                                (4)
```

La funzione analizza le operazioni eseguite valutando le asserzioni espresse mediante il predicato `ricUt/3` che descrivono le richieste degli utenti ed esegue il controllo associato. La regola `check(addSlot)` è una particolare regola in corrispondenza della quale esiste uno specifico controllo di consistenza . Nel framework realizzato sono stati implementati altri controlli di consistenza come `check(removeSlot)`, `check(addClass)`, `check(removeClass)`, etc. Il sistema, in questo modo, ha una struttura estendibile che permette di aggiungere nuovi controlli di consistenza definiti dall'utente.

3.3 Risoluzione delle inconsistenze

Ogni revisione dell'ontologia, come già riferito, può generare delle inconsistenze. Se ad esempio ad un'esistente ontologia si aggiunge uno slot, con cardinalità minima `CMin=1`, allora tutte le istanze delle classe `C` sono inconsistenti, poiché ogni istanza della classe deve avere un valore assegnato per il nuovo slot. Inoltre, se lo slot è ereditato dalle sottoclassi, la stessa inconsistenza si presenta anche sulle istanze delle sottoclassi della classe `C`. In questi casi proponiamo delle regole di ripristino della consistenza. Ad esempio, per l'inconsistenza causata della violazione delle cardinalità, abbiamo definito la regola:

```
risIncosSlotCmin(?C,?SName,?SType,?CMin,?CMax,?Inh):-
   insetrule { ?I:?Class[SName*->?V]:-
               (?Class=?C; ?Class::?C),
               ?I:?Class[not ?SName*->?_],
               default(?SType,?V)}.
```

La regola assegna un valore ad uno slot a tutte le istanze della classe.

Abbiamo implementato altre regole di ripristino delle inconsistenze in relazione all'azione di rimozione di una classe di un'ontologia. In questo caso abbiamo previsto che sono possibili più tipi di soluzioni al problema: aggancio delle sottoclassi (pendenti) alla radice dell'ontologia o alla superclasse "più vicina" alla classe eliminata oppure la loro eliminazione. Questa indeterminatezza è gestita da un'interazione del framework con il revisore dell'ontologia.

Altre risoluzioni di inconsistenza possono essere aggiunte al sistema in maniera incrementale, ossia, senza dover modificare o cancellare le esistenti regole di ripristino.

4 Conclusioni

In quest'articolo abbiamo presentato un approccio per l'evoluzione di ontologie con particolare attenzione ai controlli di consistenza definiti dall'utente. Una metodologia ed un framework sono stati definiti per permettere a un utente di gestire le fasi di evoluzione di un'ontologia: *revisione*, *controllo* e *definizione delle regole di evoluzione*. In questo modo l'utente può personalizzare il processo di evoluzione in funzione dello specifico dominio definendo nuove operazioni e regole evolutive. Come esempio di regole definite dall'utente abbiamo presentato una rappresentazione di un processo evolutivo di classi di un'ontologia che descrivono specie di artefatti.

References

1. Flora Project, http://xsb.sourceforge.net/
2. Guarino, N., Welty, C.A.: An overview of OntoClean. Handbook on ontologies, 151–159. Springer Verlag, (2004)
3. Haase P., Stojanovic, L.: Consistent evolution of owl ontologies. In Proc. The Semantic Web: Research and Applications. pp. 182–197. (2005).
4. Kifer, M., Laursen, G., Wu, J.: Logical Foundations of Object-Oriented and Frame-Based Languages. Journal of ACM. (1995).
5. OWL: Web Ontology Language, http://www.w3.org/TR/owl-features/
6. Stojanovic, L.: Methods and Tools for Ontology Evolution. PhD thesis, University of Karlsruhe. (2004)