Artifact Centric Business Processes (II)

Varianti Formali

Autore: Piero Cangialosi
Docente: Prof. Giuseppe De Giacomo

Dipartimento di Informatica e Sistemistica
SAPIENZA - Universitá di Roma

16 Novembre 2008

Outline I

Outline II

Ciclo di vita di un oggetto

Un *ciclo di vita di un oggetto* è un modello che cattura stati e transizioni ammissibili per un dato *tipo di oggetto*.

Definizione

Dato un tipo di oggetto o, il suo *ciclo di vita* $OLC_o = (S, s_a, S_\Omega, \Sigma, \delta)$ è composto da un insieme finito di stati S, dove $s_\alpha \in S$ è lo *stato iniziale* e $S_\Omega \subseteq S$ è l'insieme degli *stati finali*; un insieme finito di *eventi* Σ; una *funzione di transizione* $\delta : S \times \Sigma \rightarrow P(s)$. Dato $s_j \in \delta(s_i, e)$, scriviamo $s_i \xrightarrow{e} s_j$.

Modello di Processo (1)

Definizione

Un modello di processo $P = (N, E, O, S, \beta, I, O, D)$ è costituito da:

- un insieme finito di N nodi;
- una relazione $E : (N_A \cup N_C) \times (N_A \cup N_C)$;
- un insieme finito O di *tipi di oggetti*;
- un insieme finito S di *stati di oggetti*;
- una *condizione di branch* $\beta : N_D \times N \times O \rightarrow P(S)$;

Modello di Processo (2)

Definizione (cont.)

- una famiglia di funzioni $I = (instate_o : N_A \rightarrow P(S_o))_{o \in O}$, dove $instate_o(a)$ è l'insieme *input state* di a per o;
- una famiglia di funzioni $O = (outstate_o : N_A \rightarrow P(S_o))_{o \in O}$, dove $outstate_o(a)$ è l'insieme *output state* di a per o;
- una famiglia di funzioni $D = (dep_o : N_A \times S_o \rightarrow P(S_o))_{o \in O}$, dove $dep_o(a, s)$ è l'insieme *dependency state* di uno stato $s \in outstate_o(a)$.

Esempio

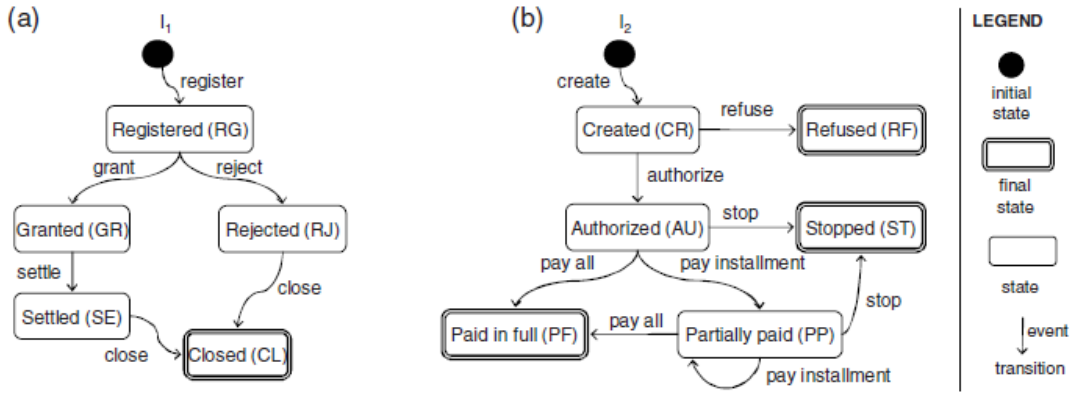


Figura: Ciclo di vita du due oggetti: (a) `Claim` (b) `Payment`

Esempio

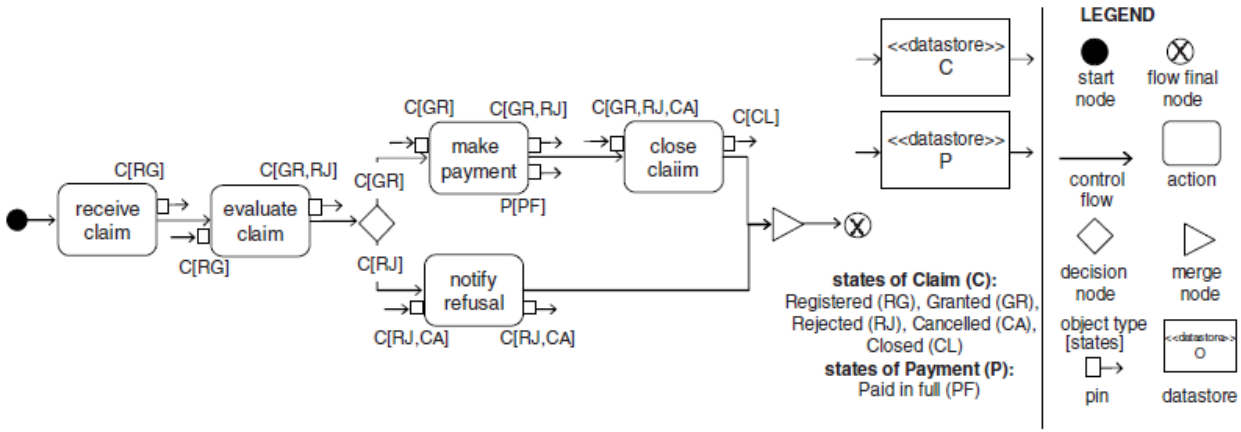


Figura: Esempio di modello di processo di business `Claim handling`

Object Provider

Definizione

L'azione a_1 è un *object provider* per a_2 rispetto ad o ed s, scritto $a_1 \lhd_o^s a_2$, se esiste un cammino $p = a_1, \ldots, a_2$ tale che:

- non c'è nessun'altra azione $a' \in N_A$ con un output pin di tipo o in p, e
- per tutti i nodi di decisione $d \in N_D$ nel cammino p e per tutti i nodi $n \in N$ in p, $d, n \in E$ implica che $s \in \beta(d, n, o)$.

Gli object provider per a_2 rispetto ad o ed s sono tutte quelle azioni che scrivono o nello stato s nel datastore corrispondendte esattamente prima che a_2 lo legga.

Transizione Indotta (1)

Definizione

Una *transizione indotta* di o in P è una tripla (a, s_{src}, s_{tgt}), tale che:

- $a \in N_A$, $s_{src} \in instate_o(a)$ e $s_{tgt} \in outstate_o(a)$, e
- $dep_o(a, s_{tgt}) = \emptyset$ oppure $s_{src} \in dep_o(a, s_{tgt})$, e
- esiste un'azione $a' \in N_A$, tale che $a' \lhd_o^{s_{src}} a$ e $s_{src} \in outstate_o^{eff}(a')$, dove $outstate_o^{eff} : N_A \rightarrow P(S_o)$ definisce l'insieme *effective output states* di un'azione: $outstate_o^{eff}(a') = \{S \in S_o | s \in outstate_o(a')$ e (a' non ha input pin di tipo o oppure (a', s', s) è una transizione indotta di o in P per qualche $s' \in S_o)\}$.

Transizione Indotta (2)

Le transizioni indotte di o in P non rappresentano nient'altro che tutte le transizioni che possono avvenire per gli oggetti di tipo o durante l'esecuzione di P.

Definizione

Dati $P = (N, E, O, S, \beta, I, O, D)$ e $OLC_o = (S, s_a, S_\Omega, \Sigma, \delta)$.

- Uno *stato inziale* di o in P è uno stato $s_{first} \in S_o$, tale che $s_{first} \in outstate_o(a)$ per qualche azione $a \in N_A$ che non ha input pin di tipo o;
- Uno *stato finale* di o in P è uno stato $s_{last} \in S_o$, tale che esiste un'azione $a \in N_A$ dove $s_{last} \in outstate_o^{eff}(a)$ ed esiste un cammino $p = a, \ldots, f$ da a ad un nodo di arrivo $f \in N_{FF}$, tale che:
  - non ci sono altre azioni $a' \in N_A$ con un output pin di tipo o nel cammino p, e
  - per ogni nodo di decisione $d \in N_D$ nel cammino p e per ogni nodo $n \in N$ in p, $(d, n) \in E$ implica che $s_{last} \in \beta(d, n, o)$.

Conformità rispetto al Ciclo di Vita

Definizione

Dati $P = (N, E, O, S, \beta, I, O, D)$ e $OLC_o = (S, s_a, S_\Omega, \Sigma, \delta)$, diciamo che P soddisfa la *conformità rispetto al ciclo di vita* per OLC_o se valgono le seguenti condizioni:

- per ogni transizione indotta $t = (a, s_{src}, s_{tgt})$ di o in P, $s_{tgt} \in \delta(s_{src}, e)$ per qualche $e \in \Sigma$ *(conformità rispetto alle transizioni)*,
- per ogni stato iniziale s_{first} di o in P, $s_{first} \in \delta(s_\alpha, e)$ per qualche $e \in \Sigma$ *(conformità rispetto allo stato iniziale)*,
- per ogni stato finale s_{last} of o in P, $s_{last} \in S_\Omega$ *((conformità rispetto allo stato finale)*.

Copertura del Ciclo di Vita

Definizione

Dati un modello di processo $P = (N, E, O, S, \beta, I, O, D)$ e un ciclo di vita o $OLC_o = (S, s_a, S_\Omega, \Sigma, \delta)$ per oggetti di tipo o, diciamo che P soddisfa la *copertura del ciclo di vita* per OLC_o se valgono le seguenti condizioni:

- per ogni $s_{src} \in S \setminus \{s_\alpha\}$, $e \in \Sigma$ e $s_{tgt} \in \delta(s_{src}, e)$, c'è una transizione indotta (a, s_{src}, s_{tgt}) di o in P per qualche $a \in N_A$ *(copertura delle transizioni)*.
- per ogni $s \in \delta(s_\alpha, e)$ per qualche $e \in \Sigma$, s è uno stato iniziale di o in P *(copertura dello stato iniziale)*.
- ogni stato finale $s_\Omega \in S_\Omega$ è uno stato finale di o in P *(copertura dello stato finale)*.

Una volta che si hanno a disposizione tutte le definizioni formali, non resta altro da fare che definire un procedimento per generare, a partire dal ciclo di vita, un modello di processo di business. Questo procedimento, nel caso di un singolo oggetto (e dunque di un singolo ciclo di vita), può essere completamente automatizzato.
Descriveremo ora il procedimento nel caso semplice. La tecnica è composta da 4 step:

Step 1

Dato il ciclo di vita $OLC_o = (S, s_a, S_\Omega, \Sigma, \delta)$ di un oggetto o vengono analizzate tutte le sue transizioni $(s_i^{o_1}, \ldots, s_i^{o_n}) \xrightarrow{e} (s_j^{o_1}, \ldots, s_j^{o_n})$ al fine di creare un insieme di azioni N_A.

for each transition $(s_i^{o_1}, ..., s_i^{o_n}) \xrightarrow{e} (s_j^{o_1}, ..., s_j^{o_n})$ in OLC
  for each o_k where $1 \leq k \leq n$
    if $(s_i^{o_k} \neq s_\alpha^{o_k}$ and $s_i^{o_k} \neq s_j^{o_k})$ then pp_{o_k} = transition
    else if $(s_i^{o_k} = s_\alpha^{o_k}$ and $s_i^{o_k} \neq s_j^{o_k})$ then pp_{o_k} = creation
    else if $(s_i^{o_k} = s_j^{o_k})$ then pp_{o_k} = no-transition
  if (there exists action a_e in N_A that matches pin pattern $\{pp_{o_1}, ..., pp_{o_n}\}$)
    for each o_k where $1 \leq k \leq n$
      if (pp_{o_k} == transition or pp_{o_k} == creation)
        add $s_j^{o_k}$ to $outstate_{o_k}(a_e)$
      if (pp_{o_k} == transition)
        add $s_i^{o_k}$ to $instate_{o_k}(a_e)$ and $s_i^{o_k}$ to $dep_{o_k}(a_e, s_j^{o_k})$
  else
    add new action a_e to N_A
    for each o_k where $1 \leq k \leq n$
      if (pp_{o_k} == transition or pp_{o_k} == creation)
        add an output pin of type o to a_e and add $s_j^{o_k}$ to $outstate_{o_k}(a_e)$
      if (pp_{o_k} == transition)
      add an input pin of type o to a_e, add $s_i^{o_k}$ to $instate_{o_k}(a_e)$ and $s_i^{o_k}$ to $dep_{o_k}(a_e, s_j^{o_k})$

Figura: Generazione del pattern per le azioni

Esempio Step 1

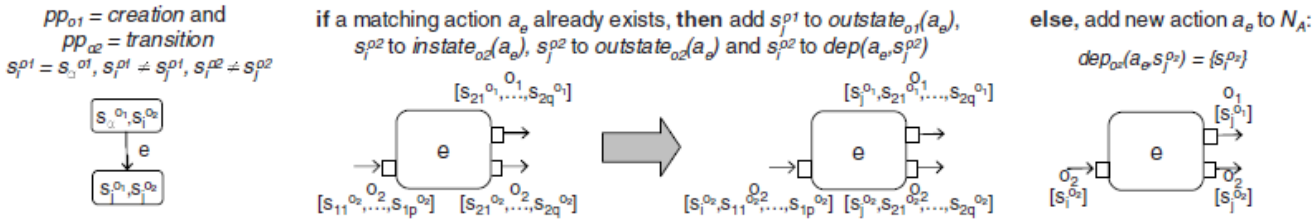


Figura: Esempio di generazione di un'azione

Step 2

Una volta creato tutto il set di azioni, deve essere creata una relazione di precedenza tra di essere. Utilizziamo a tal fine la seguente definizione:

Definizione

Data un'azione $a_1 \in N_A$ con output pin di tipo $o_{11}, \ldots, o_{1k}$ e un'azione $a_2 \in N_A$ con pin di input di tipo $o_{21}, \ldots, o_{2m}$, a_1 precede a_2 nella relazione degli stati degli oggeti, scritto $a_1 \succ_o a_2$, se e solo se per ogni tipo di oggetto $o \in \{o_{11}, \ldots, o_{1k}\} \cap \{o_{21}, \ldots, o_{2m}\}$, $outstate_o(a_1) \cap intstate_o(a_2) \neq \emptyset$.

Step 3

A questo punto vengono generati tutti i frammenti del processo iterando le regole (mutuamente esclusive) 3.1-3.4 mostrate in figura:

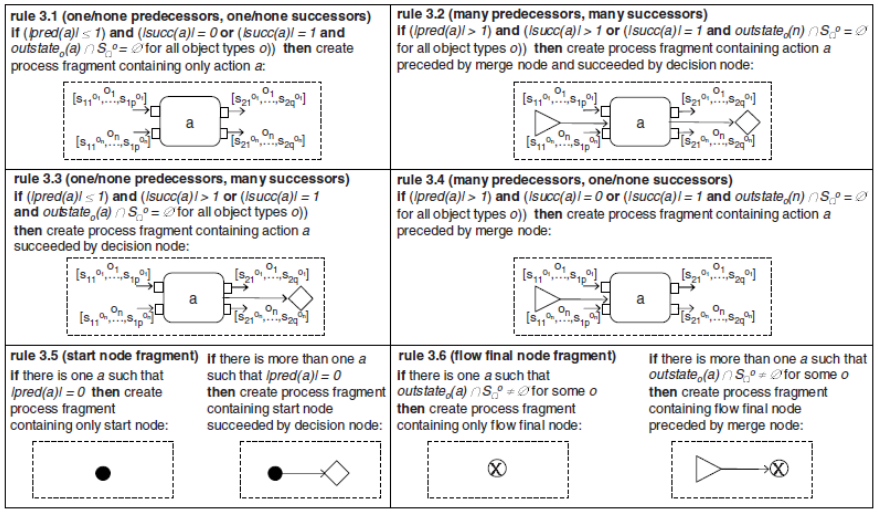
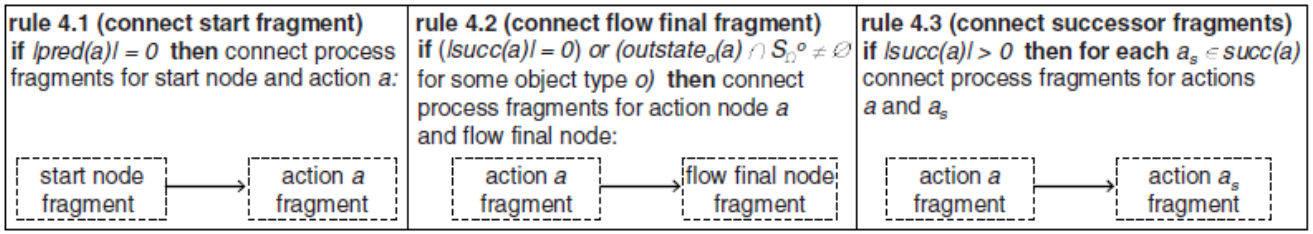
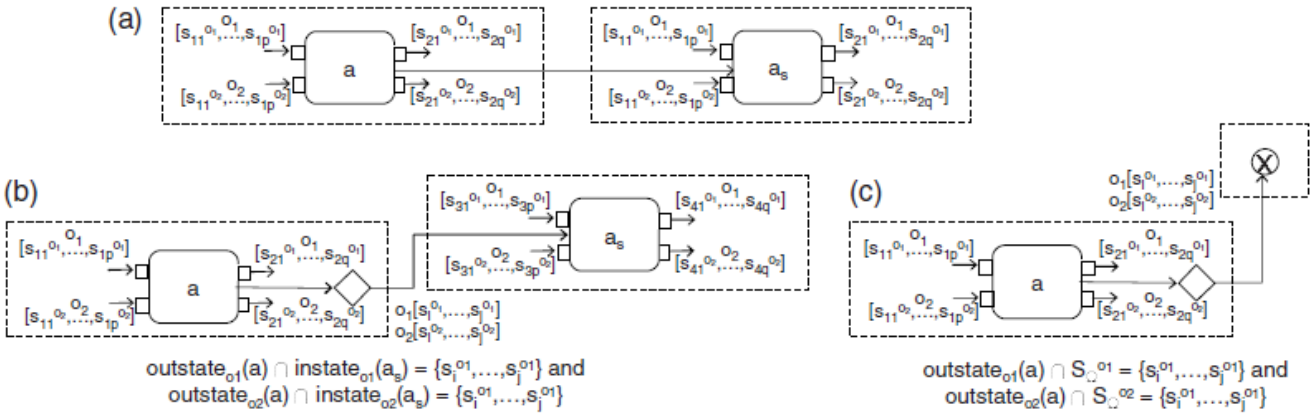


Figura: Generazione dei frammenti del processo

Step 4

Infine, i frammenti creati vengono connessi utilizzando le regole mostrate in figura:





Theorem

Un modello di processo $P = (N, E, O, S, \beta, I, O, D)$ *generato sulla base di un ciclo di vita* $OLC_o = (S, s_a, S_\Omega, \Sigma, \delta)$ *per un tipo di oggetto o soddisfa la conformità e la copertura rispetto ad* OLC_o

Evento di Sincronizzazione

Definizione

Dati due cicli di vita riferiti a due oggetti distinti $OLC_{o1} = (S_1, s_{a1}, S_{\Omega 1}, \Sigma_1, \delta_1)$ e $OLC_{o2} = (S_2, s_{a2}, S_{\Omega 2}, \Sigma_2, \delta_2)$, un evento $e \in \Sigma_1 \cup \Sigma_1$ è chiamato *evento di sincronizzazione*.

Composizione di Cicli di Vita

Definizione

Dati due cicli di vita riferiti a due oggetti distinti $OLC_{o1} = (S_1, s_{a1}, S_{\Omega 1}, \Sigma_1, \delta_1)$ e $OLC_{o2} = (S_2, s_{a2}, S_{\Omega 2}, \Sigma_2, \delta_2)$, la loro composizione dei clicli di vita è $OLC_{o1} = (S_1 \times S_2, (s_{a1}, s_{a2}), S_{\Omega 1} \times S_{\Omega 2}, \Sigma_1 \cup \Sigma_2, \delta)$ dove:

$$\delta((s_1, s_2), e) = \begin{cases} \delta_1(s_1, e) \times \delta_2(s_2, e) & \leftarrow e \in \Sigma_1 \cap \Sigma_2 \\ \delta_1(s_1, e) \times \{s_2\} & \leftarrow e \in \Sigma_1 \setminus \Sigma_2 \\ \{s_1\} \times \delta_2(s_2, e) & \leftarrow e \in \Sigma_2 \setminus \Sigma_1 \end{cases}$$

Esempio

Dall'analisi del nostro scenario d'esempio risulta evidente che un pagamento può essere creato solo se la richiesta è stata accettata. Dunque, creiamo un evento di sincronizzazione $\texttt{grant}^C|\texttt{create}^C$ e lo utilizziamo per sostituire gli eventi `grant` e `create` nei cicli di vita degli oggetti `Claim` e `Payment` rispettivamente (analogamente per $\texttt{settle}^C|\texttt{pay all}^C$).

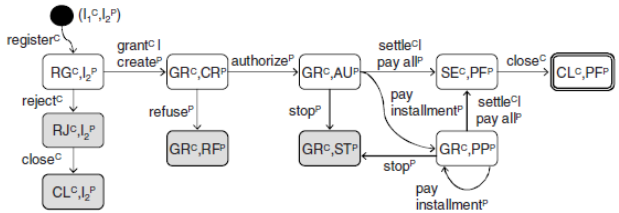


Figura: Composizione dei cicli di vita di `Claim` e `Payment`

Esempio 2

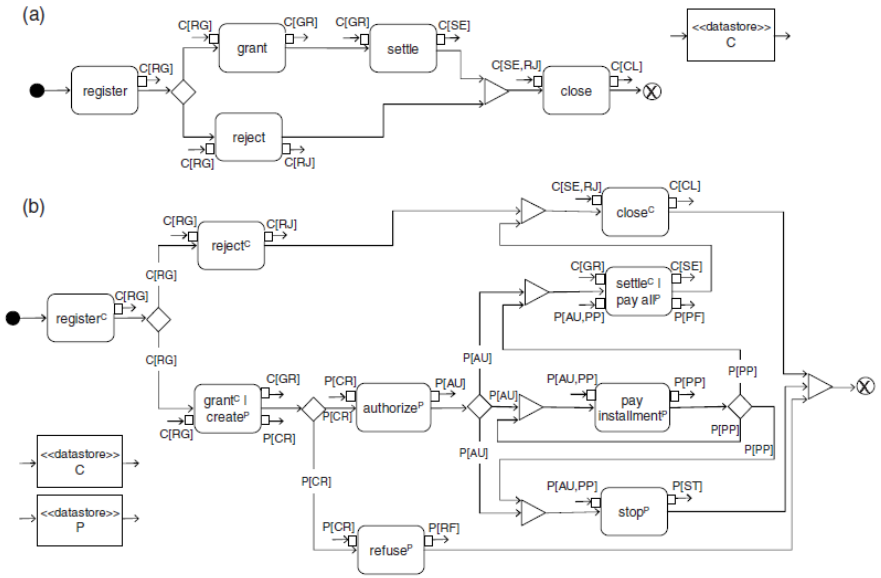


Figura: Esempio di modello di processo per `Claim handling`

Definizione

Una *classe di artefatto* è una tupla $(C, \mathbf{A}, \tau, Q, s, F)$ dove $C \in \mathcal{C}$ è un nome di classe, $\mathbf{A} \subseteq A$ è un insieme finito di attributi, $\tau : \mathbf{A} \rightarrow \Gamma$ è una funzione totale, $Q \subseteq \mathtt{STATES}$ è un insieme finito di stati, e $s \in Q$, $F \in Q$ sono gli stati *iniziale* e *finale*.

Definizione

Un *oggetto* di una classe $(C, \mathbf{A}, \tau, Q, s, F)$ è una tripla (o, μ, q) dove $o \in \mathbf{ID}_C$ è un identificatore, μ è una funzione parziale che assegna ad ogni attributo A in $\mathbf{A}$ un elemento del suo dominio $\mathrm{DOM}(\tau(A))$, e $q \in Q$ è lo stato corrente. Un oggetto (o, μ, q) è *iniziale* se $q = s$ e μ è indefinito per ogni attributo, e *finale* se $q \in F$.

CLASSE GUESTCHECK

ATTRIBUTI

cliente_p:Stringa

#_clienti:intero

#_tavolo:intero

data:Stringa

conto:intero

piatto:menu

specialità:spec

STATI

Attivo (iniziale)

Completato (finale)

OGGETTO GUESTCHECK

```
----------------------------------
ID:id9724
STATO:Attivo
ATTRIBUTI
cliente_p:Piero Cangialosi
#_clienti:1
#_tavolo:58
data:4 novembre 2008
conto:0
piatto:null
specialità:null
```

Schemi e Istanze

Definizione

Uno *schema* Γ è un insieme finiti di classi di artefatti con nomi distinti tali che ogni classe referenziata in Γ fa a sua volta parte di Γ.

Definizione

Sia Γ uno schema. Un'*istanza* di Γ è un mapping I che assegna ad ogni classe C in Γ un insieme finito, valido, e completo di oggetti di classe C. $inst(\Gamma)$ denota tale insieme. Un'istanza $I \in inst(\Gamma)$ è *iniziale*, (rispettivamente *finale*) se ogni oggetto in I è inziale (rispettivamente finale).

Termini e Atomi

Definizione

Un *atomo* per uno schema Γ assume una delle seguenti forme:

- $t_1 = t_2$, dove t_1,t_2 sono termini della classe C in Γ,
- DEFINED(t, A)
- NEW(t, A)
- $s(t)$ (*atomo di stato*), dove t è un termine della classe C ed s uno stato di C.

DEFINED(t, A) è vero se l'attributo A dell'artefatto t ha un valore e NEW(t, A) è vero se l'attributo A dell'artefatto t contiene un identificatore non in I.

Definizione di Servizio

Definizione

Un servizio per uno schema Γ è una tupla (n, V_r, V_w, P, E), dove $n \in \Theta$ è un nome di servizio, V_r, V_w insiemi finiti di variabili di classi in Γ, P una condizione stateless su V che non contiene il predicato NEW, ed E l'insieme di effetti condizionali.

Esempio di Servizio

```
---------------------------------
SERVIZIO: AGGIUNGI_PIATTO
WRITE:{x:GuestCheck}
READ:{y:menu,z:spec}
PRE:notDEFINED(x,piatto) and notDEFINED(x,specialità)
EFF:DEFINED(x,piatto) and DEFINED(x,specialità)
and x.piatto=y.nome and x.specialità=z:nome
```

Definizione

Dato uno schema Γ e un insieme di servizi Θ, una *regola di business* è un'espressione che assume una delle seguenti forme:

- **se** φ **invoca** $\sigma(x_1, \ldots, x_l; y_1, \ldots, y_k)$, oppure
- **se** φ **cambia stato in** ψ

dove φ è una condizione sulle variabili $x_1, \ldots, x_l; y_1, \ldots, y_k$, σ un servizio in Θ, tale che $x_1, \ldots, x_l$ sono tutte le variabili da modificare e $y_1, \ldots, y_k$ tutte le variabili *read-only* di σ, e ψ una condizione che consiste solo in atomi di stato positivi su $x_1, \ldots, x_l$.

Esempio di Regole

- **se** DEFINED(x, `piatto`) $\wedge$ DEFINED(x, `specialità`) $\wedge$ Attivo(x) **cambia stato in** Completato(x)
- **se** $\neg$ DEFINED(x, `piatto`) $\wedge \neg$ DEFINED(x, `specialità`) $\wedge$ Attivo(x) **invoca** Aggiungi_Piatto(x)

Questi sono intutivi problemi di decisione di fondamentale importanza quando si costruiscono sistemi artifact-based:

- **Q1:** Esiste un oggetto *o* ed un cammino o-focused valido che termina in uno stato finale?
- **Q2:** Esistono cammini validi che siano dead-end?
- **Q3:** Vi sono attributi ridondanti?

Theorem

*Sia $W = (\Gamma, \Theta, R)$ un sistema di artefatti. **Q1**, **Q2**, e **Q3** per una classe C sono indecidibili. Se W non contiene il predicato **NEW**, **Q1**, **Q2**, e **Q3** sono in* PSPACE*, e inoltre sono* PSPACE*-completi se riferiti a cammini focalizzati su un oggetto o.*

Introduciamo un nuovo tipo di predicato di stato: Un atomo *previous-or-current-state* assume la forma $[prev\_curr]s(t)$. Sia $l_0, l_1, \ldots, l_n$ un cammino. Allora [prev_curr]$s(t)$ è vero se t si trova nello stato s ora o precedentemente.

Sia $W = (\Gamma, \Theta, R)$ un sistema monotono. Assumiamo che:

(i) Ogni servizio in Θ sia *deterministico* (ha un solo effetto condizionale il cui antecedente è `true`).

(ii) La precondizione non contiene atomi negati ma può avere atomi della forma [prev_curr]$s(t)$

(iii) L'antecedente di ogni regola di **R** è positivo e può contenere atomi della forma [prev_curr]s(t).

Theorem

Sia A un attributo della classe C in Γ*, ed o un* $\mathbf{ID}_C$*. Ci sono algoritmi che risolvono in tempo lineare i seguenti problemi:*

- *Dato un attributo A in C, se esiste un cammino o-focused* $I_0, I_1, \ldots, I_n$ *in W che termina in uno stato finale tale che* $I_n(o)$ *sia definito.*
- *Dato un attributo A in C, se esiste un cammino o-focused* $I_0, I_1, \ldots, I_n$ *in W che termina in uno stato finale.*

Theorem (4.3)

*Sia $W = (\Gamma, \Theta, R)$ un sistema monotono. In relazione a cammini di o che terminano in uno stato finale il quesito **Q1** è NP-completo nei seguenti casi:*

(a) *Le condizioni (ii), (iii) sono soddisfatte da W ma non la condizione (i).*

(b) *Le condizioni (i), (ii), (iii) sono soddisfatte da W, ma la negazione è permessa nelle precondizioni dei servizi.*

(c) *Le condizioni (i), (ii), (iii) sono soddisfatte da W, ma la negazione è permessa negli antecedenti delle regole.*

(d) *se si usano atomi di stato s(t) al posto della loro versione previous-or-current-state.*

Theorem

Sia $W = (\Gamma, \Theta, R, C)$ un sistema monotono e C una classe di artefatto in Γ. Allora è $\prod_2^P$-COMPLETO verificare se ci sono cammini dead-end per C in W.

Theorem

Sia $W = (\Gamma, \Theta, R, C)$ un sistema monotono. Il problema di decidere se esiste un attributo A della classe $C \in \Gamma$ che sia ridondante è coNP-COMPLETO *per tutti i casi del teorema 2.2.3.*

Questa sezione tratta la realizzazione automatica di insiemi di regole a partire da pre-schemi, ovvero coppie classe-servizi. La ricerca si basa sul concetto di *sicurezza* di un insieme di regole, ovvero la garanzia che ogni esecuzione all'interno dello schema termini in uno stato che soddisfa una determinata condizione.

L'artefatto `PurchaseOrder`

Artifact Class: `PurchaseOrder`
Associated Attributes:
- `prodName`: string
- `prodType`: {"hw", "sw"}
- `bid`: integer
- `profitMargin`: [0..100] // as a percentage
- `approved`: bool
- `execApproved`: bool
- `scheduleDate`: date
- `archived`: bool

Figura: La classe `PurchaseOrder`

Una famiglia di servizi per `PurchaseOrder`

`Estimate`: `profitMargin`
  PRE: `DEF(prodName)` $\wedge$ `DEF(prodType)` $\wedge$ `DEF(bid)`
  EFF:
    `bid` $\leqslant 400 \rightarrow$ `profitMargin` $\leqslant 25\%$
    `bid` $> 350 \rightarrow$ `profitMargin` $> 20\%$
`RoutineApproval`: `approved`
  PRE: `DEF(bid)` $\wedge$ `DEF(profitMargin)`
  EFF:
    `bid` $\leqslant 100 \rightarrow$ `approved` $= true$
    `prodType` $=$ "sw" $\rightarrow$ `approved` $= true$
    `prodType` $=$ "hw" $\wedge$ `profitMargin` $> 10\%$
      $\rightarrow$ `approved` $= true$
    "else" $\rightarrow$ `approved` $= false$
`ExecApproval`: `execApproved`
  PRE: `approved` $= false$
  EFF: $true \rightarrow$ `DEF(execApproved)`
`Schedule`: `scheduleDate`
  PRE: `DEF(prodName)`
  EFF: $true \rightarrow$ `DEF(scheduleDate)`
`Archive`: `archived`
  PRE: `DEF(scheduleData)` $\vee$ `execApprov` $= false$
  EFF: $true \rightarrow$ `archived` $= true$

Quesito 1

Soddisfacibilità

Dato un pre-schema P ed un goal γ, c'è una istanza iniziale o della classe ed un cammino in P il cui nodo finale (inteso sempre come fotografia del sitema) soddisfi γ?

Istanziamo la domanda nel caso della classe `PurchaseOrder` e dei servizi ad essa associati:

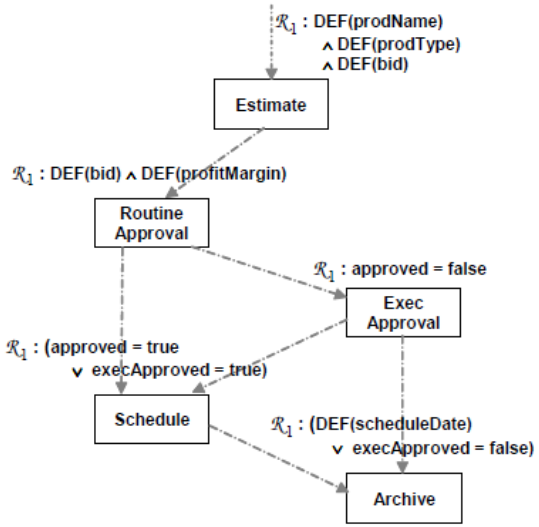
$c1$: `archived` = *true*

$c2$: DEF(`scheduleDate`) $\rightarrow$ (`approved` = *true* $\wedge$ `execApproved` = *true*))

γ_1: $c1 \wedge c2$

La clausola $c1$ asserisce che un'esecuzione soddisfa γ_1 solo se prevede l'attivazione del servizio `Archive`.
La clausola $c2$ stabilisce che un acquisto può essere schedulato solo se è stato approvato da `RoutineApproval` oppure da `ExecApproval`.

Esempio



Due delle regole dell'insieme sono:

- **if** DEF(`prodName`) $\wedge$ DEF(`prodName`) $\wedge$ DEF(`bid`) **invoke** `Estimate`
- **if** (`approved` = *true* $\vee$ `execApproved` = *true*) **invoke** `Schedule`

Quesito 2

Insieme massimale di regole γ-safe per P

Dato un pre-schema $P = (A, S)$ ed un goal γ, esiste un insieme di regole R tale che (A, S, R) sia γ-safe? Esiste un tale insieme che sia *massimale*?

Un insieme γ-safe R è massimale se permette ogni altra esecuzione permessa dagli altri insiemi di regole γ-safe.
In riferimento all'esempio precedente, si può dimostrare che R_1 è γ_1-safe mssimale per `Purchasing`.

Consideriamo ora il goal γ_2

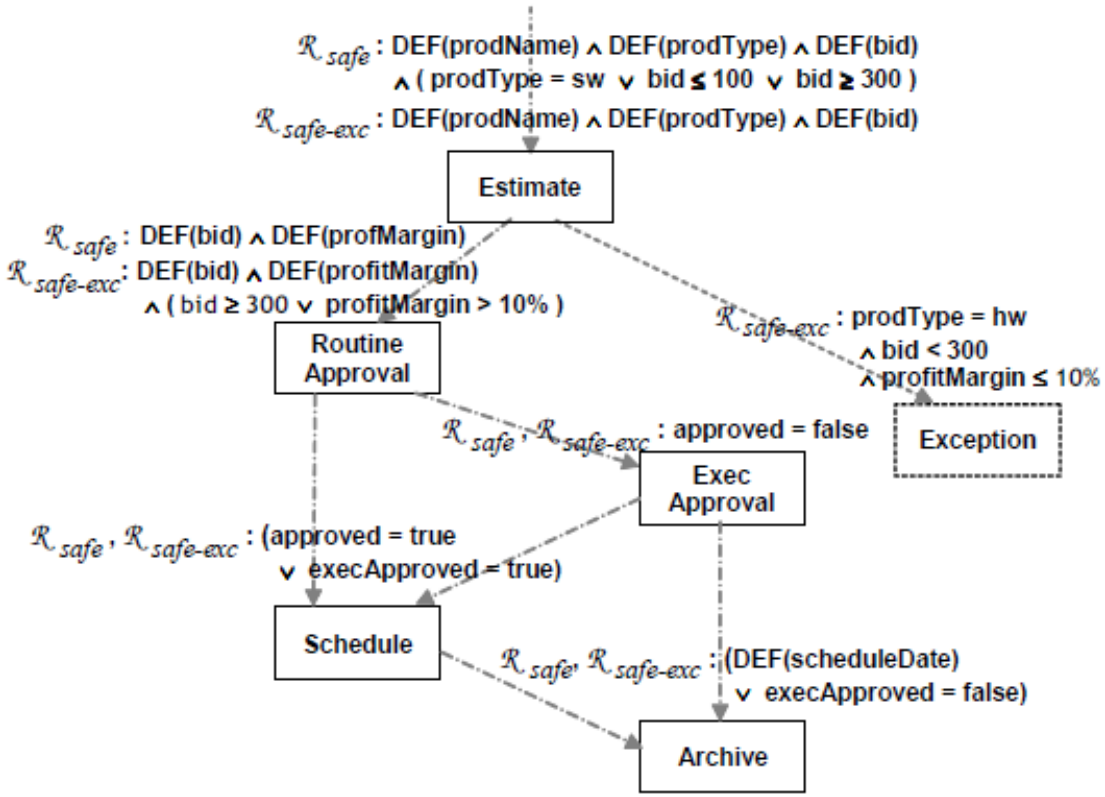
$c3$: DEF(`execApproved`) $\rightarrow$ `bid` ≥ 300

γ_2: $\gamma_1 \wedge c3$

e supponiamo che l'istanza in ingresso abbia `prodType` = *hw* e `bid` compreso tra 100 e 300. Si può facilmente verificare che in tal caso l'esecuzione può terminare in uno stato (non-estendibile) che non soddisfa γ_2. Una tale esecuzione è chiamata *dead-end*.

Esempio

L'insieme R_{safe} di figura 10 è invece γ_2-safe massimale.



Per evitare le situazioni di dead-end, possiamo cercare un insieme di regole che prevenga ogni caso di questo tipo, o possiamo creare un meccanismo di *detection* di tali situazioni. A tal fine, modifichiamo la specifica del sistema introducendo un nuovo servizio EXC. Sostanzialmente, questo servizio agisce come una sorta di *cestino*. Esso ha infatti una regola che lo invoca ogni qualvolta si verifica un condizione sufficiente ad implicare che l'esecuzione finirà in una situazione di *dead-end*.

Per fare ciò, introduciamo un sistema modificato:

$P^{EXC} = (A^{\mathtt{exc}}, S^{\mathtt{exc}} \cup \{EXC\})$

- $A^{\mathtt{exc}}$ è ottenuto aggiungendo ad ogni classe di A, il nuovo attributo $\mathtt{exc}$, e
- $S^{\mathtt{exc}}$ è il risultato della modifica di ogni servizio σ in S nel servizio $\sigma^{\mathtt{exc}}$, ottenuto sostituendo la precondizione ρ di σ con $\rho \wedge \neg \mathrm{DEF}(\mathtt{exc})$.

Infine, in un pre-schema P^{EXC} il goal γ diviene $\gamma \vee (\mathtt{exc} = \mathit{true})$.

Intuitivamente, un siffatto schema può essere sicuro o soddisfando il goal originario oppure deviando l'artefatto che violerebbe la condizione nel servizio EXC.

Quesito 3

Insieme massimale di regole γ-safe per P con eccezioni

Dato un pre-schema $P = (A, S)$ ed un goal γ, esiste un insieme di regole R tale che (A, S, R) sia γ-safe massimale?

Denotiamo il *core* di un servizio σ come $core(\sigma) = \langle \pi(\bar{x}), \rho(\bar{x}\bar{y}) \rangle$, dove $\bar{x}$ and $\bar{y}$ sono enumerazioni delle variabili di attributi in R e gli attributi (già modificati) in W.
Sia $\langle \pi(\bar{x}), \rho(\bar{x}\bar{y}) \rangle$ il core di un servizio σ e $\gamma(\bar{x}\bar{y})$ un goal:

Definizione

una $\forall$-*precondition* per σ e γ è una formula $\epsilon(\bar{x})$ tale che

$$\mathbf{M} \models \forall \bar{x}(\epsilon(\bar{x}) \rightarrow \forall \bar{y}(\sigma(\bar{x}\bar{y}) \rightarrow \gamma(\bar{x}\bar{y})))$$

una $\exists$-*precondition* per σ e γ è una formula $\xi(\bar{x})$ tale che

$$\mathbf{M} \models \forall \bar{x}(\xi(\bar{x}) \rightarrow \exists \bar{y}(\sigma(\bar{x}\bar{y}) \wedge \gamma(\bar{x}\bar{y})))$$

Weakest Precondition

Intuitivamente, dato un servizio σ e un goal γ, una $\forall$-precondition è una condizione **sufficiente** affinché l'invocazione di tale servizio porti il sistema in uno stato finale che soddisfa il goal γ per ogni possibile esecuzione.
Una $\exists$-precondition è invece una condizione sufficiente affinchè, data l'invocazione del servizio, esista una esecuzione che porta nello stato finale di soddisfazione del goal γ.
Una $\forall$-(o $\exists$-)precondition è *weakest* se è logicamente implicata da tutte le altre $\forall$-(o $\exists$-)precondition.

Riformuliamo il **Quesito 2** indicando un metodo di massima per risolverlo utilizzando le weakest $\forall$-precondition.

Quesito 2

Esiste un insieme di regole **R** γ-safe massimale per **P** ? In caso affermativo, esiste un algoritmo che lo costruisce?

In un approccio semplicistico, il **Quesito 2** può essere risolto in questo modo: per ogni servizio calcola la weakest $\forall$-precondition rispetto al goal desiderato e utilizza tale precondizione per costruire una regola dell'insieme.

Theorem

*Se il problema **Q2** è risolvibile per L, allora la teoria del primo ordine di **M** è decidibile.*

Introducendo invece una restrizione di *non ripetibilità*, in cui limitiamo un servizio ad essere invocato solo una volta, ed utilizzando un sottoinsieme della logica FOL senza l'uso dei quantificatori, che chiamiamo L^{QF}, giungiamo ad un altro risultato.

Theorem

*Sotto la restrizione di non ripetibilità, **Q2** è risolvibile per L^{QF} se la teoria di **M** è decidibile ed ammette l'eliminazioni dei quantificatori.*

Theorem

*Sotto la restrizione di non ripetibilità, **Q2** è risolvibile per L^{QF} quando L ammette (1) ordinamento totale denso o discreto (con $\leq$), (2) artimetica lineare, (3) aritmetica reale (con $\leq, +, \times$).*

Utilizziamo il modello sul quale si basano gli esempi, ovvero che assume una struttura FOL sottostante con un dominio che ammette un ordinamento denso lineare, che denotiamo con $L^{\leq}$. Assumiamo poi che ogni servizio definisca un solo attributo, e che se tale servizio definisce A_n allora ha bisogno di leggere solo attributi A_i con $i < n$. Le pre-condizioni dei servizi e gli antecedenti delle regole sono formule in $L^{\leq}$, e le post-condizioni assumono la forma di *effetti condizionali*. Chiamiamo questo modello $W^{QF,<}$.

Mostriamo dunque un algoritmo che, dato $P = (A, S)$ e un goal γ in $W^{QF,<}$, costruisce un insieme di regole R γ-safe massimale:

1. Assumi che ci sia l'ordinamento per gli attributi
2. per ogni servizio σ che definisce A_n, crea la regola **se** $wp^{\forall}(\sigma, \gamma)$ **allora invoca** σ
3. Per ogni $i \in [1...n-1]$ fai, in ordine inverso:
   1. siano R_j = **se** α_j **allora invoca** σ_j le regole per quei servizi σ_j che definiscono A_{i+1}
   2. Per ogni servizio σ che definisce A_i crea la regola **se** $wp^{\forall}(\sigma, \gamma) \bigwedge(\alpha_j)$ **allora invoca** σ

Assumiamo che vi sia un dominio infinito D su cui è possibile definire un ordine totale denso; le formule del linguaggio vengono dunque costruite su $D \cup \{=, \leq\}$.
Un'istanza, o interpretazione, dello schema è un mapping che associa ad ogni simbolo di relazione R dello schema una relazione finita su D di arità $a(R)$.
Se $\varphi(\bar{x})$ è una formula FO con variabili libere $\bar{x}$, e $\bar{u}$ è una tupla su D della stessa arità di $\bar{x}$, denotiamo con $\varphi(\bar{x} \leftarrow \bar{u})$ l'enunciato ottenuto sostituendo $\bar{u}$ a $\bar{x}$ in $\varphi(\bar{x})$.
Dal momento che D è infinito, una formula FO $\varphi(\bar{x})$ può essere soddisfatta da infinite tuple $\bar{u}$ su D. Questo ostacolo può essere superato utilizzando la semantica del dominio attivo, in cui si restringe il dominio al solo insieme di elementi presenti nell'istanza. Data un'istanza I, chiamiamo $adom(I)$ il suo dominio attivo.

Classe di Artefatto

Definizione

Una *classe di artefatto* è una coppia $C = (R, S)$ dove R ed S sono due simboli di relazione. Un'istanza di C è una coppia **C** = (**R**,**S**), dove (i) **R**, chiamata relazione di attributo, è un'interpretazione di R contenente esattamente una tupla su D, e (ii) **S**, chiamata relazione di stato, è un'interpretazione finita di S su D.

Schema e Istanze

Definizione

Uno *schema (di artefatti)* è una tupla $A = (C_1, \ldots, C_n, DB)$ dove ogni $C_i = (R_i, S_i)$ è una classe di artefatto, DB è uno schema relazionale, e C_i, C_j, e DB non hanno simboli di relazione in comune per i $\neq$ j.

Definizione

Un'*instanza* di uno schema $A = (C_1, \ldots, C_n, DB)$ è una tupla $\mathbf{A} = (\mathbf{C}_1, \ldots, \mathbf{C}_n, \mathbf{DB})$ dove $\mathbf{C}_i$ è un'istanza di C_i e **DB** è un'istanza di DB su D.

Definizione

Un servizio σ per uno schema A è una tupla $\sigma = (\pi, \psi, Z)$ dove:

- π, chiamata *pre-condizione*, è un enunciato in L_A;
- ψ, chiamata *post-condizione*, è un enunciato in L_A, con variabili libere ($\bar{x}_R | R$ è un attributo di relazione di una classe in A).
- Z è un insieme di *regole di stato* contenente, per ogni relazione S di A, una, nessuna o entrambe delle seguenti regole:
  - $S(\bar{x}) \leftarrow \phi_S^+(\bar{x})$;
  - $\neg S(\bar{x}) \leftarrow \phi_S^-(\bar{x})$;

  dove $\phi_S^+(\bar{x})$ e $\phi_S^-(\bar{x})$ sono formule in L_A con variabili libere $\bar{x}$ t.c. $|\bar{x}| = a(S)$.

Semantica dei Servizi

Definizione

Un *sistema (di artefatti)* è una coppia $\Gamma = \langle A, \Sigma \rangle$, dove A è uno schema e Σ un insieme di servizi non vuoto per A.

Sia $\sigma = (\pi, \psi, Z)$ un servizio per uno schema A. Siano **A** e **A'** due istanze di A. Diciamo che **A'** è un possibile successore di **A** rispetto a σ (scritto **A** $\xrightarrow{\sigma}$ **A'**) se valgono le seguenti condizioni:

Semantica dei Servizi

1. $A \models \pi$;
2. $A'|DB = A|DB$;
3. se $\bar{u}_R$ è il contenuto della relazione di attributo R di A in **A'**, allora **A** soddisfa la post-condizione ψ dove ogni $\bar{x}_R$ è sostituito da $\bar{u}_R$;
4. per ogni relazione di stato S di A e per ogni tupla $\bar{u}$ su $adom(A)$ di arità $a(S)$, **A'** $\models S(\bar{u})$ se e solo se **A** $\models$

$$(\phi_S^+(\bar{u}) \wedge \neg\phi_S^-(\bar{u})) \vee (S(\bar{u}) \wedge \phi_S^+(\bar{u}) \wedge \phi_S^-(\bar{u})) \vee (S(\bar{u}) \wedge \neg\phi_S^+(\bar{u}) \wedge \neg\phi_S^-(\bar{u}))$$

dove $\phi_S^+(\bar{u})$ e $\phi_S^-(\bar{u})$ sono interpretate con la semantica del dominio attivo (inoltre se la rispettiva regola non esiste vengono automaticamente poste a `false`).

Esecuzioni del Sistema

Definizione

Un'esecuzione di un sistema $\Gamma = \langle A, \Sigma \rangle$ è una sequenza infinita $\rho = \{\rho_i\}_{i \geq 0}$ di istanze su A (chiamate anche *configurazioni*) tali che:

- ρ_0 è un'istanza iniziale di Γ;
- per ogni $i \geq 0$, $\rho_i \xrightarrow{\sigma} \rho_{i+1}$ per qualche $\sigma \in \Sigma$.

Una *pre-esecuzione* è una sequenza finita $\{\rho_i\}_{0 \leq i \leq n}$ che soddisfa le condizioni appena definite $i < n$. Una pre-esecuzione è *bloccante* se la sua ultima configurazione non ha nessun possibile successore.

Utilizzo della logica LTL-FOL

Definizione

Il linguaggio LTL-FO è ottenuto chiudendo la logica del primo ordine sotto negazione, disgiunzione, e la seguente regola per la formazione delle formule: se φ e ψ sono formule, allora $\mathbf{X}\varphi$ e $\varphi\mathbf{U}\psi$ sono formule.
La chiusura universale di una formula LTL-FO $\varphi(\bar{x})$ con variabili libere $\bar{x}$ è la formula $\forall\bar{x}\varphi(\bar{x})$. Un enunciato nella logica LTL-FO è la chiusura universale di una formula LTL-FO.

Utilizzo della logica LTL-FOL (2)

L'innovazione consiste nell'utilizzo di LTL-FO per la verifica delle proprietà del sistema.
Sia $\Gamma = \langle A, \Sigma \rangle$ un sistema, e $\forall \bar{x} \varphi(\bar{x})$ un enunciato LTL-FO su A. Il sistema Γ soddisfa $\forall \bar{x} \varphi(\bar{x})$ se e solo se ogni esecuzione di Γ lo soddisfa. Sia $\rho = \{\rho_i\}_{i \geq 0}$ tale esecuzione ($\rho_{\geq j}$ denota $\rho = \{\rho_i\}_{i \geq j}$, per $j \geq 0$). L'esecuzione ρ soddisfa $\forall \bar{x} \varphi(\bar{x})$ se e solo se per ogni valutazione ν di $\bar{x}$ in D, $\{\rho_i\}_{\geq 0}$ soddisfa $\varphi(\nu(\bar{x}))$.

Decidibilità delle verifiche

La verifica di soddisfacibilità di una formula LTL-FO è in generale indecidibile (si veda il teorema di Trakhtenbrot). Per superare tale difficoltà, si introduce una speciale classe di sistemi, chiamati *guarded*.
Tali sistemi rispettano tutte le definizioni formali appena introdotte, ad eccezione del fatto che le regole per la creazione di *formule ben formate* il logica FOL vengono modificate.
Inoltre, le pre-e-post-condizioni dei servizi vengono ristrette al solo uso del quantificatore esistenziale.
Un sistema si dice guarded se e solo se lo sono tutte le formule utilizzate nelle regole di aggiornamento delle relazioni di stato dei suoi servizi, e tutte le pre-e-postcondizioni sono formule $\exists*$FO dove gli atomi di stato sono ground (ovvero non contengono variabili).

Sistemi Guarded

Definizione

Sia $\Gamma = \langle A, \Sigma \rangle$ un sistema. L'insieme di formule FO *guarded* su A è ottenuta sostituendo questa regola:

- Se φ è una formula e x è una variabile, allora $\forall$xA e $\exists$xA sono formule.

con la seguente:

- se φ è una formula, α è un atomo che utilizza una relazione di attributo di qualche artefatto di A, $\bar{x} \subseteq \mathit{free}(\alpha)$, e $\bar{x} \cap \mathit{free}(\beta) = \emptyset$ per ogni atomo di stato β in φ, allora $\exists \bar{x}(\alpha \wedge \varphi)$ e $\forall \bar{x}(\alpha \rightarrow \varphi)$ sono formule.

Theorem

Dato un sistema guarded Γ *ed una formula LTL-FO guarded* φ*, verificare se ogni esecuzione di* Γ *soddisfa* φ *è decidibile. Inoltre, il problema è* PSPACE*-completo per schemi di arità fissata, ed* EXPSPACE *altrimenti.*

Theorem

Dato un sistema guarded Γ*, verificare se tutte le pre-esecuzioni di* Γ *sono bloccanti è decidibile. Inoltre, il problema è* PSPACE*-completo per schemi di arità fissata, ed* EXPSPACE *altrimenti.*

Il rilassamento di una qualsiasi delle proprietà introdotte finora porta ad un condizione di non decidibilità.
Eliminando anche una sola delle restrizioni per sistemi *guarded*, o eliminando la distinzione tra attributi di relazione ed attributi di stato, non è possibile decidere se un sistema $\Gamma \models \varphi$.
Anche una semplice dipendenza funzionale nello schema relazionale porta ad una condizione di indecidibilità:

Theorem

Dato un sistema guarded singleton Γ*, un insieme di dipendenze funzionali* F *su DB, ed un enunciato LTL-FO guarded* φ*, non è possibile decidere se* $\rho \models \varphi$ *per ogni esecuzione* ρ *di* Γ *in un database che soddisfi* F*.*

Sorprendentemente, è indecidibile anche la verifica dell'esistenza di un'esecuzione bloccante.

Theorem

Dato un sistema guarded Γ*, è indecidibile verificare se* Γ *ha qualche esecuzione bloccante.*