Artifact Centric Business Processes

Seminari di Ingegneria del Software

Autore: Piero Cangialosi
Dipartimento di Informatica e Sistemistica
SAPIENZA - Universitá di Roma

Docente: Giuseppe De Giacomo
Dipartimento di Informatica e Sistemistica
SAPIENZA - Universitá di Roma

16 Novembre 2008

Sommario

In questa tesina viene illustrato il problema di modellare Processi di Business seondo una nuova ottica Artifact-Centric. La tecnica si focalizza sui dati e sull'individuazione degli oggetti chiave che evolvono durante il processo. Il metodo è stato usato con successo in vari contesti, comportando significativi miglioramenti nell'implementazione dei Processi di Business.

Nella prima parte mostriamo un'introduzione all'argomento ed una metodologia di modellazione generale. Nella seconda parte vengono illustrati vari approcci in maniera formale, che portando ad importanti risultati teorici. Il primo approccio riguarda la creazione automatica di processi a partire dai cicli di vita degli oggetti coinvolti. Viene poi illustrato un modello generale riconducibile alla metodologia introduttiva e i primi risultati di complessità. Le ultime due sezioni riguardano rispettivamente la creazione automatica di regole associate ai servizi coinvolti nel processo, e la verifica automatizzata di proprietà del sistema.

Indice

Capitolo 1

Introduzione

1.1 Gli artefatti

La modellazione dei processi di business *artifact-centric* nasce a metà degli anni novanta all'interno di IBM, presentandosi come la risposta al crescente bisogno delle aziende di poter applicare rapidamente i cambiamenti ai propri processi di business e poter così superare i propri competitori sul mercato.

Per usare termini matematici, l'obiettivo principale era quello di poter applicare dei cambiamenti, innovazioni e integrazioni dei propri business in maniera lineare con le dimensioni degli stessi, cosa che, evidentemente, le vecchie metodologie *process-centric* non riuscivano a garantire.

Questa nuova tecnica non nasce dunque da un'esigenza prettamente informatica, ma da una richiesta proveniente direttamente dai business manager.

L'obiettivo ambizioso che ci si propose di raggiungere era infatti quello di creare una rappresentazione pensata per le persone responsabili dei processi di business, ma che avesse al contempo una struttura formale che permettesse un'analisi rigorosa, applicando ad esempio tecniche di ragionamento automatico.

La metodologia che presenteremo si basa sul concetto fondamentale di *artefatto*. I principali vantaggi di ragionare secondo quest'ottica sono:

- maggiore flessibilità nella rapresentazione
- maggiore facilità ad analizzare i cambiamenti
- maggiore capacità di gestire i sistemi di implementazione che supportano il business

1.1.1 Una visione assiomatica

Un artefatto è un pezzo di informazione concreto, identificabile e auto-esplicativo che può essere utilizzato da una persona coinvolta nel processo di business per portare avanti il processo stesso, e di esso rappresenta l'unica informazione esplicita.

Le proprietà formali di un artefatto sono catturate dai seguenti assiomi:

- un artefatto è caratterizzato da un'identità unica, valida all'interno dell'intero enteprise, e da un contenuto auto-esplicativo
- l'identità di un artefatto non può essere cambiata
- il contenuto di un artefatto può essere modificato arbitrariamente

Un artefatto ha anche un ciclo di vita, che riflette ovviamente l'evoluzione di tale artefatto attraverso il processo di business.

Il processamento degli artefatti (che possono essere creati, modificati, archiviati - anche se non ancora completi) viene modellato attraverso altri due concetti: task e repository.

Qui una visione assiomatica:

- l'obiettivo di un task è quello di effettuare un'azione e memorizzare il risultato in uno o più artefatti in suo possesso
- un task viene eseguito mediante l'arrivo di un artefatto oppure tramite un opportuno meccanismo di attivazione
- gli artefatti entrano in un task spontaneamente oppure attraverso una esplicita richiesta
- se un task ha terminato la sua esecuzione tutti gli artefatti in esso contenuti vengono espulsi
- gli artefatti possono essere inseriti in un repository
- un repository può rispondere ad una richiesta per un artefatto in esso contenuto ma non può inviare artefatti spontaneamente

Supponiamo ora di voler rappresentare il processo di business relativo ad un ristorante. L'esempio, incentrato sull'artefatto `Guest Check`, è ripreso da [**?**].

Lo scenario tipico è il seguente: un cliente si reca al ristorante e viene ricevuto da un commesso. Viene creata una scheda che registrerà tutte le informazioni (di interesse per il business) relative alla cena del cliente e dei suoi accompagnatori. Tali informazioni possono includere il nome del cliente principale, il numero di persone sedute al tavolo, il numero del tavolo, la data, i piatti ordinati e il costo totale. Il ristorante offre un menu standard e delle specialità del giorno. Una richiesta dei clienti genera un'ordinazione per la cucina e i piatti scelti vengono memorizzati nel `Guest Check`. Quando la cena è conclusa il conto viene chiuso, il bilancio in cassa modificato di conseguenza, e il `Guest Check` viene archiviato come pagato.

In figura **??** è mostrato il ciclo di vita dell'artefatto `Guest Check`.

Entrano in gioco pertanto 5 artefatti:

- i piatti del menu;
- le specialità del giorno;
- i `Guest Check`;
- gli ordini per la cucina;
- la cassa.

Come si evince dalla figura e dallo scenario descritto, abbiamo bisogno di 4 task per modellare le seguenti operazioni:

- creazione del `Guest Check`;
- chiusura del `Guest Check`;

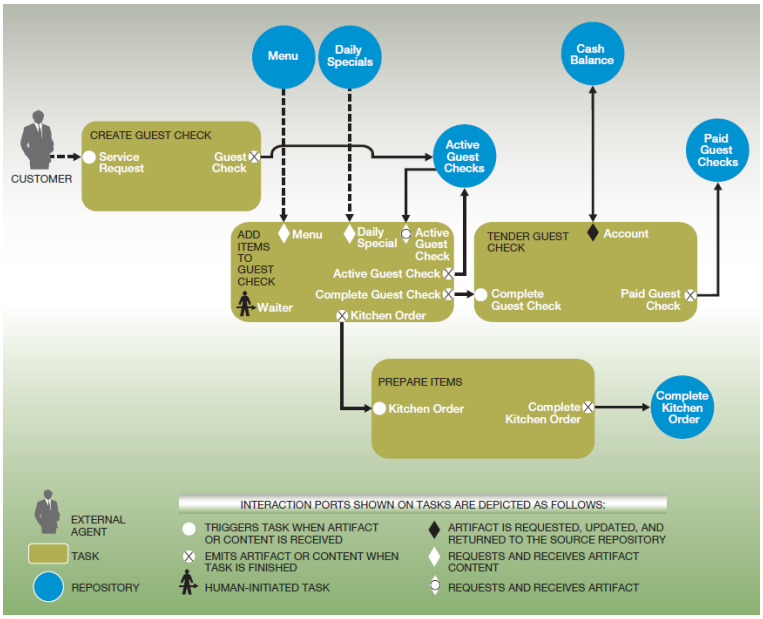

Figura 1.1.1: Ciclo di vita dell'artefatto `Guest Check`

- aggiunta ordinazioni al `Guest Check`;
- creazione ordine per la cucina.

Abbiamo bisogno poi di 6 repository per mantenere:

- i piatti del menu;
- le specialità del giorno;
- i `Guest Check` attivi;
- i `Guest Check` completati;
- gli ordini per la cucina;
- la cassa.

Il task che aggiunge le ordinazioni al `Guest Check` può essere chiamato un numero arbitrario di volte, prelevando l'artefatto dal repository dei `Guest Check` attivi ed aggoirnandolo (si pensi allo scenario in cui si chiede al cameriere di ordinare un nuovo piatto).

Il modello mostrato è ovviamente incompleto, dal momento che manca tutta la parte relativa al ciclo di vita degli altri artefatti.

In generale, la tecnica è quella di individuare gli artefatti chiave assieme ai relativi cicli di vita. Dall'analisi di questi ultimi vengono fuori un'altra serie di artefatti, e così via..

Questo processo a cascata termina quando è stato raggiunto il livello di dettaglio desiderato.

Nella prossima sezione verrà illustrata una metodologia precisa più orientata ai servizi.

1.2 Una Metodologia di Design Data-Centric per i Processi di Business

In questa sezione mostreremo una metodologia di design incentrata sui Business Artifact come illustrata in [**?**].

Il primo passo dell'approccio data-centric consiste nell'individuare gli artefatti chiave assieme ai rispettivi cicli di vita. Successivamente viene creata una specifica logica dettagliata per ogni classe di artefatto, per i servizi che vi opereranno, e per le associazioni tra i task e i servizi (che, come vedremo, assumono la forma di regole di business). L'ultimo passo consiste nel trasformare questa specifica *dichiarativa* in una specifica *procedurale*, che può messere ottimizzata e successivamente mappata in una implementazione fisica.

Ci concentreremo sui primi 2 passi.

1.2.1 La Metodologia in Dettaglio

Il cuore della metodologia è rappresentato da un modello stratificato a 3 livelli. Al livello più alto si trova il Business Operations Model (BOM) che fornisce una *specifica logica* del processo di business secondo una precisa semantica che comprende gli artefatti, i servizi e le regole di attivazione di tali servizi, descritte tramite regole Evento-Condizione-Azione (ECA), secondo lo spirito delle moderne architetture Service Oriented. Al livello medio ti trova un flusso di lavoro *concettuale* che cattura la semantica essenziale del BOM, trascurando però i dettagli implementativi. Al livello più basso troviamo un flusso di lavoro *operazionale* nel quale i servizi manipolano gli artefatti e si scambiano informazioni tramite messaggi.

La metodologia si basa dunque su quattro concetti fondamentali:

- **ARTEFATTI**
  Un artefatto, come accennato nell'introduzione, contiene informazioni pertinenti al processo di business, e in particolare le informazioni necessarie a valutare lo stato di avanzamento del processo in ogni istante.
  Un artefatto ha una propria identità e il suo stato di avanzamento può essere tracciato attraverso l'intero flusso di lavoro.
  Possiede una serie di attributi che possono essere semplici scalari o strutture dati annidate.

- **(MACRO) CICLI DI VITA**
  La scoperta di nuovi artefatti procede mano a mano che si analizza il ciclo di vita degli artefatti stessi.
  Questi cicli di vita sono generalmente definiti come macchine a stati finiti, che incorporano gli stati che rappresentano i momenti fondamentali della vita degli artefatti all'interno del processo, dalla creazione al completamento e all'archviazioni finale.
  In alcuni casi un artefatto ha una vita breve (si pensi agli ordini per la cuicina nell'esempio precedente); in altri casi è quasi permanente (ad esempio i piatti presenti nel menu).

- **SERVIZI**
  Un servizio modifica uno o più artefatti (la modifica si riflette nel valore degli attributi di questi ultimi) e tali modifiche sono transazionali, ovvero un servizio deve avere un controllo esclusivo sugli artefatti che sta modificando. Viene utilizzato il termine *servizio* piuttosto che *task* per evidenziare la stretta corrispondenza tra i

servizi in cui si parla qui e i tipi di servizi che si trovano nelle Services Oriented Architecture (SOA) e nei Web Service in generale.

- **ASSOCIAZIONI**
  Un servizio applica delle modifiche ad una serie di artefatti in maniera ristretta da una serie di vincoli. Tali vincoli assumono forme differenti a seconda che ci si trovi nei vari layer descritti precedentemente.
  Per la parte che interessa questa sezione, i vincoli verranno espressi sotto la vorma di regole ECA. In seguito verranno mostrati vari esempi.

Riassumendo, i passi da compiere per arrivare ad una specifica finale sono i seguenti:

1. *Individuazione degli Artefatti*
   - Identificare gli Artefatti principali
   - Identificare gli stadi principali del ciclo di vita degli Artefatti tramite l'analisi dello scenario
2. *Design del Business Operation Model (BOM)*
   - Specifica logica dello schema degli Artefatti (ad esempio modello ER)
   - Specifica dei Servizi che fanno avanzare gli Artefatti attraverso i loro cicli di vita
   - Specifica delle regole ECA che abilitano tali servizi
3. *Design del diagramma del flusso di lavoro concettuale*
4. *Implementazione di ogni singolo componente*

La centralità dei dati in questa metodologia si esprime attraverso due principi. Da un lato, si nota come ad ogni step la specifica e il design dei dati preceda quello degli altri componenti. Dall'altro, la specifica dei task e del flusso di lavoro viene creata sulla base del design dei dati ottenuto ad ogni step.

1.2.2 Esempio di Applicazione della Metodologia

Mostriamo ora un esempio di applicazione di questa metodologia sulla base dello scenario accennato nella precedente sezione.

Individuazione di Artefatti e Ciclo di Vita

Ricordiamo che erano stati individuati 5 artefatti: piatto del menu, specialità del giorno, `Guest Check`, ordine per la cucina, cassa.

Supponiamo che il `Guest Check` possa trovarsi nello stato attivo oppure completato. Per tutti gli altri artefatti prevediamo, per semplicità, che il ciclo di di vita comprenda solo la fase di creazione.

In figura è mostrato il ciclo di vita dell'artefatto `Guest Check`.

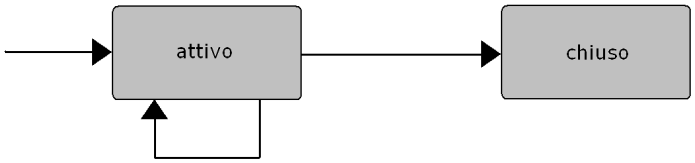

Figura 1.2.1: Ciclo di vita dell'artefatto `Guest Check`

Design del Business Operations Model

In questa parte, come accennato, bisogna creare una specifica logica dettagliata di: (a) schemi degli artefatti, (b) servizi che opereranno su tali artefatti per permetterne l'avanzamento lungo il proprio ciclo di vita, (c) regole ECA che associano i servizi agli artefatti. Scegliamo di modellare lo schema degli artefatti utilizzando il modello Entità-Relazione (la scelta non è comunque obbligata).

Supponiamo che un `Guest Check` contenga informazioni circa: lo stato in cui si trova (se attivo oppure completo), i piatti del menu e le specialità del giorno che si sono ordinate al tavolo, il cliente principale, il numero di clienti, il numero del tavolo, la data, l'entità del conto e, ovviamente, un identificativo. Un elemento `Menu` contiene un identificativo del piatto, un descrizione ed un prezzo, così come un elemento `SpecGiorno`, che in aggiunta contiene una descrizione degli ingredienti. Un `Ordine` per la cucina contiene i piatti del menu e le specialità del giorno da preparare ed un identificativo. Per semplicità la struttura dell'artefatto `Cassa` non viene mostrato.

In figura **??**, **??**, **??**, **??** vengono mostrati gli schemi degli artefatti appena descritti.

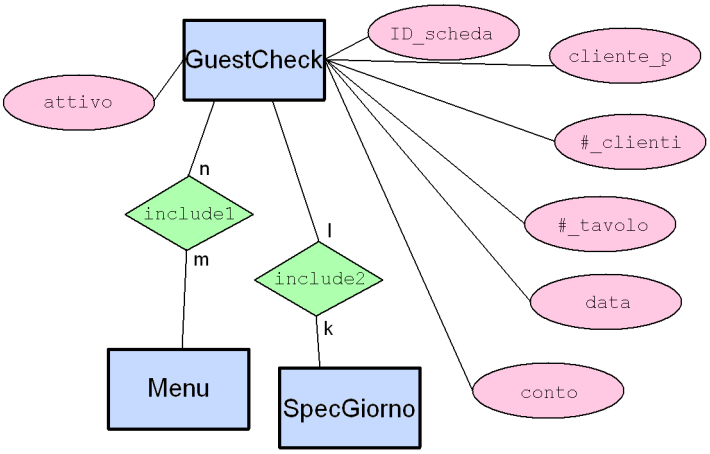

Figura 1.2.2: Schema dell'artefatto `Guest Check`

Quando un artefatto viene creato, molti dei propri attributi sono indefiniti (ad esempio, alla creazione del `Guest Check` sicuramente gli attributi che contengono i piatti ordinati e il conto totale non hanno un valore).

Durante il passaggio nei vari stadi gli attributi possono assumere un valore (diventare cioè definiti) o essere sovrascritti. Nel caso di attributi che contengono insiemi o liste (come i piatti ordinati nell'artefatto `Guest Check`) si possono aggiungere o eliminare valori.

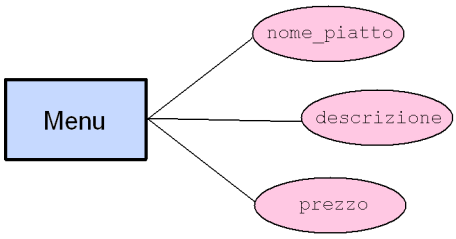

Figura 1.2.3: Schema dell'artefatto `Menu`

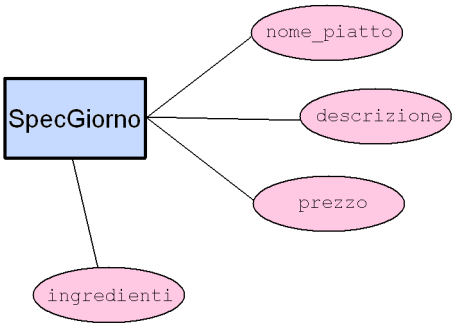

Figura 1.2.4: Schema dell'artefatto `SpecGiorno`

É anche possibile prevedere che un attributo venga *invalidato*, ma non è di interesse ai fini di questa spiegazione approfondire questo caso.

Passiamo ora a descrivere la seconda fase della specifica del Business Operations Model, ovvero l'individuazione dei Servizi.

Ogni Servizio si focalizza su di un singola classe di un Artefatto, ed eventualmente legge o scrive attributi di altri Artefatti.

Mostriamo pertanto un possibile elenco di Servizi relativi all'esempio scelto:

crea_Guest_Check(): Questo servizio ha l'effetto di creare un `Guest Check`

crea_Menu(): Questo servizio ha l'effetto di creare un elemento del `Menu`

crea_SpecGiorno(): Questo servizio ha l'effetto di creare una specialità giornaliera

prepara_piatto(): Questo servizio istanzia un ordine per la cucina

aggiungi_piatto_Guest_Check(`Menu`: m, `SpecGiorno`: s, `Guest Check`: g): Questo servizio aggiunge il piatto del menu m o la specialità giornaliera s a g

chiudi_Guest_Check(`Guest Check`: g, `Cassa`: c): Questo servizio chiude g e aggiorna il saldo in c

Mostreremo ora la specifica completa di due servizi: *aggiungi_piatto_Guest_Check* e *chiudi_Guest_Check*.

Nella metodologia che stiamo illustrando, al livello del Business Operations Model, la specifica sei Servizi è composta dai seguenti elementi (IOPE):

- artefatti e attributi di **I**nput,

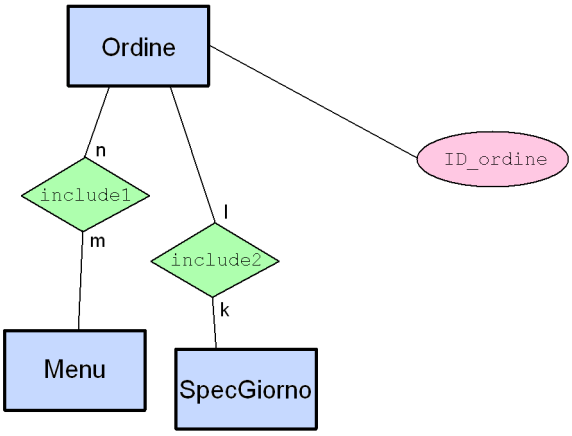

Figura 1.2.5: Schema dell'artefatto `Ordine`

- artefatti e attributi di **O**utput,
- **P**re-condizioni, e
- **E**ffetti (Condizionali).

Nella specifica IOPE dei Servizi gli artefatti e gli attributi di input (risp. output) identificano i valori che verranno letti (risp. aggiornati) dal servizio. Le pre-condizioni devono essere soddisfatte affinché il servizio possa essere eseguito.

Specifica IOPE del Servizio *aggiungi_piatto_Guest_Check*:

- Input:
  - Una lista di elementi `Menu` m da aggiungere alla scheda.
  - Una lista di elementi `SpecGiorno` s da aggiungere alla scheda.
  - L'artefatto `Guest Check` g da modificare.
- Output:
  - Update di g
  - un nuovo ordine per la cucina o
- Pre-condizioni:
  - `Guest Check` è nello stato *attivo*.
- Effetti condizionali:
  - `true` → Le relazioni *include*1 e *include*2 di g vengono aggiornate per includere i nuovi piatti e specialità del giorno.
  - `true` → l'ordine o contiene la lista dei piatti e delle specialità ordinate

Specifica IOPE del Servizio *chiudi_Guest_Check*:

- Input:
  - L'artefatto `Guest Check` g da chiudere.

  - La cassa

- Output:
  - Update del `Guest Check` g e della Cassa.
- Pre-condizioni:
  - `Guest Check` è nello stato *attivo*.
- Effetti condizionali:
  - `true` $\rightarrow$ g è nello stato `completato` e al bilancio cassa si aggiunge $g.conto$.

La semantica degli effetti condizionali del servizio è associata ad una condizione di *circumscription* (vedremo in seguito il significato preciso).

Passiamo ora alla specifica dei (Micro) cicli di di vita degli artefatti, ovvero alle regole Evento-Condizione-Azione che associano l'esecuzione dei servizi agli artefatti.

I componenti fondamentali di una regola ECA sono i seguenti:

- Eventi:
  - Un attributo appena assegnato
  - Un passaggio di stato per un artefatto
  - Il lancio o il completamento di un servizio
  - Una esplicita richiesta da parte di un esecutore
- Condizioni
  - Formule scritte in un sottoinsieme della Logica del Primo Ordine o in Algebra Relazionale
- Azioni
  - Invocazione di un Servizio
  - Cambio di stato per un Artefatto

Mostriamo due esempi di regole:

R1: *Aggiungi Piatto*

evento richiesta di aggiungere una lista di elementi `Menu` m e una lista di elementi `SpecGiorno` s al `Guest Check` g.

condizione g è nello stato `attivo`

azione invoca *aggiungi_piatto_Guest_Check*(m,s,g)

R2: *Chiudi Conto*

evento richiesta di chiudere il conto associato al `Guest Check` g.

condizione g è nello stato `attivo`

azione invoca *chiudi_Guest_Check*(g)

La semantica dell regole ECA è basata sui concetti seguenti:

- **Non-determinismo:** Se più di una regola è candidata ad essere attivata da uno stesso evento, il sistema ne sceglie una non deterministicamente.
- **No starvation:** Una regola non può attendere di essere attivata indefinitamente.
- **Serializzabilità:** L'esecuzione delle regole può essere interleaved e/o parallela, ma l'effetto deve essere equivalente a quello di un qualche ordine seriale.

Utilizzando una sematica basata sulle regole ECA, vi sono varie cose che si possono verificare. Al livello logico, ad esempio:

- **Raggiungibilità:** Dato un insieme di regole, esiste un predicato raggiungibile tramite un'esecuzione di tali regole?
- **Deadlock:** Il sistema può raggiungere un deadlock? In caso affermativo, la situazione di deadlock può essere evitata?
- **Terminazione:** Supponendo di avere artefatti con un ciclo di vita finito, ogni possibile esecuzione delle regole garantisce che gli artefatti si trovino nello stato finale dopo un numero finito di passi?

Dall'implementazione delle regole ECA nascono poi altri due questi fondamentali (c'è una distizione essenziale tra implementazioni semplicistiche delle regole ed implementazioni altamente ottimizzate):

- **Conformità:** Si può dimostrare che l'implementazione soddisfa tutti i requisiti definiti nella semantica (ad esempio assenza di starvation e serializzabilità)?
- **Ottimizzazione:** Come può essere implementato il sistema di regole affinchè non si testino inutilmente condizioni di attivazione di regole?

Una volta ottenuto il Business Operation Model bisogna eseguire gli ultimi due step al fine di ottenere un'implementazione per ogni singolo componente. Questa parte della metodologia verrà per brevità omessa.

É facile però intuire come la stratificazione introdotta garantisca un'indipendenza logica e di processo che permette cambiamenti ai requisiti senza grosso impatto su gli altri componenti.

I miglioramenti introdotti con la visione Data-Centric sono in un certo senso assimilabili a quelli dei linguaggi ad oggetti rispetto ai predecessori.

In questa sezione abbiamo brevemente introdotto una possibile metodologia di design (lo sviluppo non è stato approfondito) Artifact/Data-Centric. É essenziale precisare che tale metodologia non è assolutamente l'unica. La ricerca sull'argomento, al momento in cui questa tesina viene scritta, è ben lungi dall'essere conclusa. In verità, i primi risultati teorici sono recentissimi.

Nella prossime sezioni abbandoneremo il modello generale appena introdotto (verrano poste restrizioni, ad esempio, su come possono essere scritte le regole e i servizi) utilizzando una semantica e una sintassi estremamente precisa.

Mostreremo come, partendo da vari approcci, si potranno stabilire importanti risultati teorici che rispondono in parte alle questioni pocanzi introdotte.

Capitolo 2

Varianti formali

2.1 Generazione Automatica di Modelli per Processi di Business

Presentiamo ora una tecnica di generazione automatica di modelli per processi di business a partire dal ciclo di vita definito per i vari oggetti, basandoci sul lavoro di [**?**].

Anche se nell'articolo non emerge chiaramente il concetto di artefatto o di servizio, è utile comprendere varie nozioni necessarie a costruire il concetto di *compliance* di un modello di processo rispetto ad una serie di cicli di vita di oggetti.

2.1.1 Modelli per Processi di Business e Cicli di Vita degli Oggetti

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

Come accennato nel precedente capitolo, l'utilizzo di macchine a stati finiti non deterministiche è una tecnica utile ed efficace per rappresentare il ciclo di vita di un oggetto (il concetto di stato di un oggetto è assimilabile a quello definito per gli artefatti).

Definizione 2.1.1. 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* $\delta : S \times \Sigma \to P(s)$. Dato $s_j \in \delta(s_i, e)$, scriviamo $s_i \xrightarrow{e} s_j$.

In figura **??** è mostrato il ciclo di vita per due tipi di oggetti di esempio `Claim` e `Payment`.

Un modello di processo di business mostra come vengono organizzati e coordinati i singoli task (assimilabili ai servizi introdotti nella precedente sezione) e come questi ultimi si scambiano gli oggetti.

Definizione 2.1.2. Un modello di processo $P = (N, E, O, S, \beta, I, O, D)$ è costituito da:

- un insieme finito di N nodi partizionato come segue:
  - N_A *nodi azione*;
  - N_C *nodi di controllo* partizionati come segue: N_D *nodi di decisione*, N_M nodi di merge, N_F *nodi di fork*, N_J *nodi di join*, N_S *nodi di partenza* e N_{FF} *nodi di arrivo*;
  - N_O *nodi oggetto* partizionati come segue: N_{IP} *pin di input*, N_{OP} *pin di output* e N_{DS} datastore;

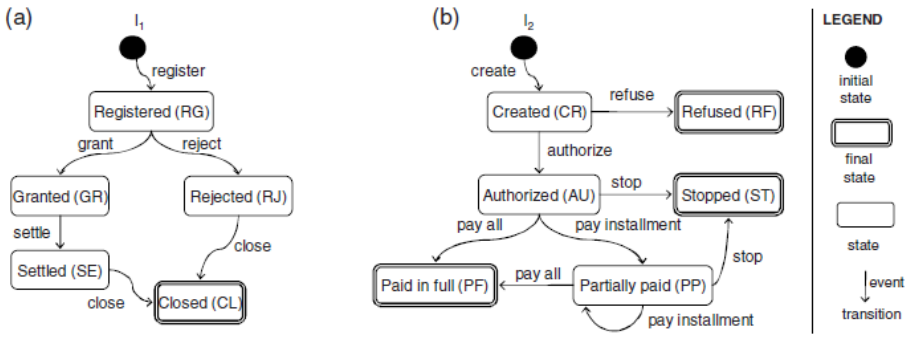

Figura 2.1.1: Ciclo di vita du due oggetti: (a) `Claim` (b) `Payment`

- una relazione $E : (N_A \cup N_C) \times (N_A \cup N_C)$ che rappresenta il *flusso di controllo*;
- un insieme finito O di *tipi di oggetti*;
- un insieme finito S di *stati di oggetti*, partizionati in una serie di insiemi, uno per ogni tipo di oggetto, ovvero $S = S_{o_1} \cup \ldots \cup S_{o_n}$ se $O = (o_1, \ldots, o_n)$;
- una funzione che rappresenta la *condizione di branch* $\beta : N_D \times N \times O \to P(S)$;
- una famiglia di funzioni $I = (instate_o : N_A \to P(S_o))_{o \in O}$, dove $instate_o(a)$ è l'insieme *input state* di a per o;
- una famiglia di funzioni $O = (outstate_o : N_A \to P(S_o))_{o \in O}$, dove $outstate_o(a)$ è l'insieme *output state* di a per o;
- una famiglia di funzioni $D = (dep_o : N_A \times S_o \to P(S_o))_{o \in O}$, dove $dep_o(a, s)$ è l'insieme *dependency state* di uno stato $s \in outstate_o(a)$.

La figura **??** mostra un modello di processo semplificato per la gestione dei due oggetti d'esempio introdotti precedentemente, che chiamiamo `Claim handling`.

La notazione è mutuata dagli Activity Diagram di UML2.

La relazione E rappresenta gli archi di collegamento tra i task; la funzione β rappresenta il comportamento dinamico del sistema; $instate_o(a)$ è l'insieme di stati dell'oggetto o che l'azione a accetta in input; $outstate_o(a)$ è l'insieme di stati in cui l'azione a può lasciare l'oggetto o dopo la sua esecuzione. Tutti gli elementi della definizione hanno una rappresentazione grafica, eccetto l'insieme $dep_o(a, s)$.

Quest'ultimo definisce la relazione ingresso-uscita dell'azione a sugli oggetti di tipo o. La sua controparte nell'esempio della precedente sezione è la definizione del servizio. Mentre però in quel caso il cambiamento operato dai servizi si rifletteva anche in una modifica degli attributi, qui tale concetto scompare e prendiamo in consideraizone solo cambiamenti di stato.

Se volessimo ad esempio modellare il fatto che il task `notify refusal` non cambia lo stato degli oggetti di tipo `Claim` dovremmo asserire: dep_C(`notify refusal`,RJ) = {RJ} e dep_C(`notify refusal`,CA) = {CA}.

dep_C(`close claim`,CL) = {$\emptyset$} indica invece che `close claim` lascia sempre gli oggetti di tipo `Claim` nello stato CL.

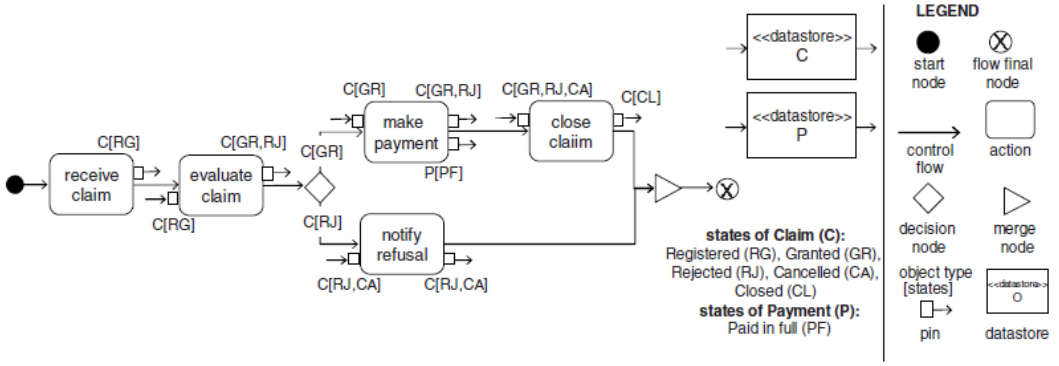

Figura 2.1.2: Esempio di modello di processo di business `Claim handling`

2.1.2 Object Life Cycle Compliance

Introduciamo ora delle definizioni necessarie a catturare il concetto di Object Life Cycle Compliance.

Un modello di processo, come vedremo, è *compliant* rispetto all'insieme dei cicli di vita degli oggetti sul quale è definito, se è corretto e completo.

Dato un modello di processo $P = (N, E, O, S, \beta, I, O, D)$, un *cammino* è definito come una sequenza finita $n_1, \ldots, n_k$ dove $n_i \in N$ e $(n_i, n_{i+1}) \in E$ per $1 \leq i < k$. Il concetto è perfettamente assimilabile a quello di cammino in un grafo, dove i nodi sono rappresentati dall'insieme N e gli archi sono definiti dalla relazione E.

Definizione 2.1.3. Siano dati un modello di processo $P = (N, E, O, S, \beta, I, O, D)$, un tipo di oggetto $o \in O$, uno stato $s \in S_o$ e due azioni $a_1, a_2 \in N_A$, tali che $s \in outstate_o(a_1) \cap instate_o(a_2)$. 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.

Ad esempio, in figura **??**, `evaluate claim` è un object provider per `make payment` e `notify refusal` rispetto agli oggetti di tipo `Claim` e agli stati GR e RJ.

Definizione 2.1.4. Siano dati un modello di processo $P = (N, E, O, S, \beta, I, O, D)$ e un tipo di oggetto $o \in O$. 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 $outstate_o^{eff} : N_A \to P(S_o)$ 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 $s' \in S_o)\}$.

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.

Analizziamo la definizione: le prime due asserzioni limitano s_{tgt} e s_{src} ad essere tutte le coppie di stati, per ogni azione, tali che (1) uno è in ingresso all'azione, (2) l'altro è in uscita, e (3) il passaggio da uno all'altro è ammesso dalla formalizzazione del modello. La terza asserzione stabilisce che deve esistere un'azione a' che faccia da object provider di a per o nello stato s_{src} e $s_{src} \in dep_o(a, s_{tgt})$, ovvero a' sta creando l'oggetto oppure sta compiendo anch'essa una transizione effettiva.

Ad esempio, in figura **??**, (`close claim`, GR, CL) è una transizione indotta per `Claim`.

Definizione 2.1.5. Siano dati un modello di processo $P = (N, E, O, S, \beta, I, O, D)$ e un tipo di oggetto $o \in O$.

- 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 $s_{last} \in \beta(d, n, o)$.

Analizziamo la prima asserzione della definizione: se un azione a può lasciare un oggetto o uno stato s, ma non ha input pin di tipo o, ne deriva che quell'oggetto è stato creato da a, ed s è uno stato iniziale.

Analizziamo la seconda asserzione: scegliamo un'azione a, un tipo di oggetto o e uno stato $s \in outstate_o^{eff}(a)$. Scegliamo poi un cammino da a ad un nodo di arrivo. Se il cammino è conforme alla condizione di branch e non ci sono altre azioni che leggono oggetti di tipo o, s è ovviamente uno stato finale per o nel processo.

In figura **??** RG è lo stato iniziale e CL ed RJ sono gli stati finali per gli oggetti di tipo `Claim`.

A questo punto, si detengono tutti gli strumenti formali per definire la *conformità* e la *copertura* che, come accennato, rappresentano la correttezza e la completezza di un modello di processo di business rispetto ai cilci di vita.

Definizione 2.1.6. 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 *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)*.

Il processo in figura **??**, in riferimento a `Claim`, soddisfa la conformità allo stato inziale ma non quella allo stato iniziale e alle transizioni.

Definizione 2.1.7. 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)*.

Il processo in figura **??**, in riferimento a `Claim`, soddisfa la copertura dello stato inziale e finale ma non quella delle transizioni.

2.1.3 Il Processo di Generazione

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.

Il caso di due o più tipi di oggetti, invece, è più complesso. Bisogna infatti capire come combinare i cicli di vita dei vari oggetti per ottenere una macchina a stati finiti unica, sulla quale si potrà poi agire in maniera automatica come nel caso di cui sopra.

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.

Di seguito (figura **??**) è presentato l'algoritmo:

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 2.1.3: Generazione del pattern per le azioni

Analizziamo la procedura: nel primo blocco dell'algoritmo viene analizzata la transizione. A seconda dello stato iniziale, per ogni oggetto del vettore, possiamo capire se si tratta di una creazione, di una transizione o di una non-transizione. Viene così creato un pattern.

Nel secondo blocco dell'algoritmo si cerca un'azione che abbia un pattern uguale. Se esiste, è sufficiente aggiungere i nuovi stati; altrimenti è necessario creare una nuova azione.

Un esempio è mostrato in figura **??**:

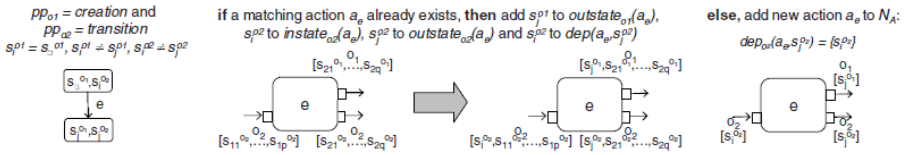

Figura 2.1.4: Esempio di generazione di un'azione

step 2 Una volta creato tutto il set di azioni, deve essere creata una relazione di precedenza tra di esse. Utilizziamo a tal fine la seguente definizione:

Definizione 2.1.8. 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 $o \in \{o_{11}, \ldots, o_{1k}\} \cap \{o_{21}, \ldots, o_{2m}\}$, $outstate_o(a_1) \cap intstate_o(a_2) \neq \emptyset$.

La definizione è piuttosto semplice: prese due azioni a_1 e a_2, a_1 precede a_2 se e solo se, per tutti i pin in comune tra l'input di a_2 una e l'output di a_1, c'è almeno uno stato in comune.

step 3 A questo punto vengono generati tutti i frammenti del processo iterando le regole (mutuamente esclusive) 3.1-3.4 mostrate in figura **??**.

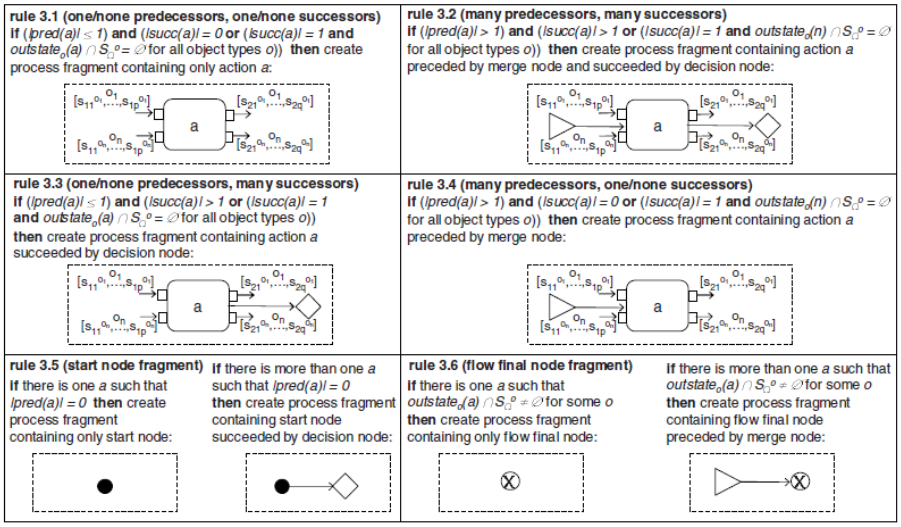

Figura 2.1.5: Generazione dei frammenti del processo

step 4 Infine, i frammenti creati vengono connessi utilizzando le regole mostrate in figura **??** e in figura **??**.

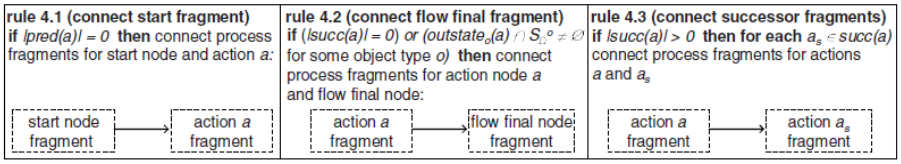

Figura 2.1.6: Connessione dei frammenti del processo

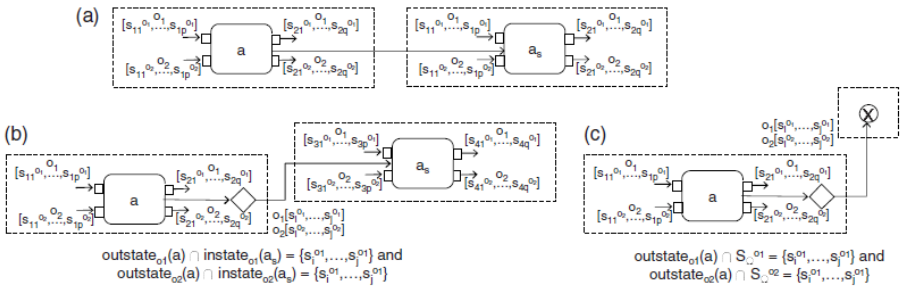

Figura 2.1.7: Connessione dei frammenti del processo

Il seguente teorema mostra che la conformità e la copertura sono soddisfatte utilizzando il procedimento appena descritto.

Teorema 2.1.1. *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.
Lemma: *Per tutte le azioni* $a \in N_A$ *e i tipi di oggetti* $o \in O$, $s \in outstate_o(a)$ *implica* $s \in outstate_o^{eff}(a)$.

Sincronizzazione e Composizione di Cicli di Vita di Oggetti

Resta ora da definire il meccanismo col quale è possibile comporre cicli di vita da oggetti diversi e crearne uno unico.

La tecnica si basa sul concetto fondamentale di *evento di sincronizzazione*.

Definizione 2.1.9. 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*.

In generale, la composizione di clicli di vita si calcola utilizzando la seguente definizione:

Definizione 2.1.10. 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:

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

Per capire l'utilità del concetto di evento di sincronizzazione è utile astrarsi per un istante dal modello matematico appena introdotto e concentrarsi sull'obiettivo principale della nostra modellazione.

Tipicamente il processo di business riguarda una serie di oggetti il cui ciclo di vita viene costruito indipendentemente l'uno dall'altro, ma che concorrono parallelamente all'esecuzione del processo.

Utilizzare la definizione di composizione appena data senza individuare gli eventi di sincronizzazione significherebbe giungere ad una situazione in cui sarebbero ammessi stati compositi in realtà non validi, e ad una evidente esplosione di stati nel grafo del ciclo di vita.

Ad 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 $\texttt{settle}^C|\texttt{pay all}^C$).

La figura **??** mostra i ciclo di vita composto utilizzando tali eventi di sincronizzazione.

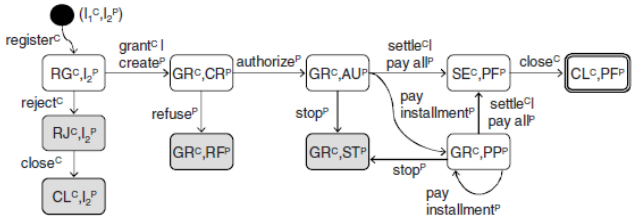

Figura 2.1.8: Composizione dei cicli di vita di `Claim` e `Payment`

Ci sono quindi due stati composti, come CL^C,RF^P e CL^C,ST^P, che non sono raggiungibili nel grafo composto appena creato; vi sono inoltre stati composti evidenziati in grigio che non portano ad uno stato finale (sono essenzialmente stati di deadlock). La scelta se includere o no tali stati nel grafo è lasciata al designer.

La figura **??** (a) mostra il processo generato sulla base del solo tipo di oggetto `Claim`. La figura **??** (b) mostra il processo generato sulla base del ciclo di vita derivante dalla composizione di `Claim` and `Payment`.

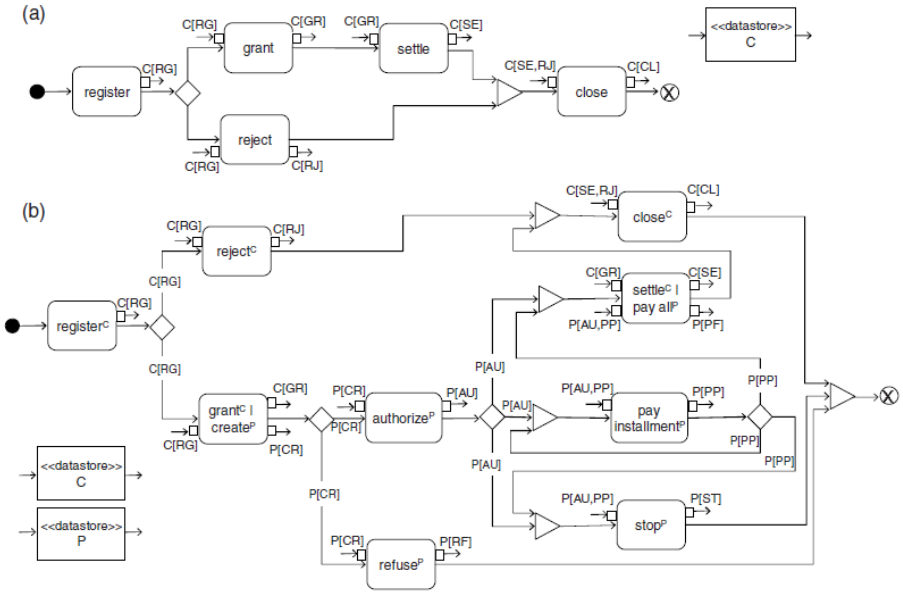

Figura 2.1.9: Esempio di modello di processo per `Claim handling`

2.2 Analisi Formale di Modelli di Processi di Business Artifact-Centric

In questa sezione, verrà dato il dovuto supporto formale al modello introdotto nella prima parte della tesina, sulla base di [**?**]. Il modello è piuttosto generale e potrebbe essere applicato alla metodologia descritta in [**?**], con le dovute restrizioni su come possono essere scritte le precondizioni dei servizi e le regole di attivazione deli stessi.

Sussessivamente, si restringerà ulteriormente il modello, prevedendo sistemi monotoni. Importanti risultati di complessità verranno dimostrati in caso di tali classi di sistemi.

2.2.1 Schemi ed Artefatti

Per cominciare, si assume l'esistenza dei seguenti insiemi (a due a due distinti, infiniti e numerabili): insieme Γ_p di *tipi primitivi*, $\mathbf{C}$ di *nomi di classi (di artefatti)*, $\mathbf{A}$ di *nomi di attributi*, `STATES` di *stati (di artefatti)*, e $\mathbf{ID}_C$ di *identificatori (di artefatti)* per ogni classe $C \in \mathbf{C}$. Un tipo è un elemento nell'unione $\Gamma = \Gamma_p \cup C$.

Il *dominio* di ogni tipo t in Γ, scritto $\mathbf{DOM}(t)$, è definito come segue: (1) se $t \in \Gamma_p$ è un tipo primitivo, il dominio $\mathbf{DOM}(t)$ è qualche insieme standard di *valori* (interi, stringhe, etc.); (2) se $t \in C$ è un tipo artefatto, $\mathbf{DOM}(t) = \mathbf{ID}_t$.

Definizione 2.2.1. Una *classe di artefatto* è una tupla $(C, \mathbf{A}, \tau, Q, s, F)$ dove $C \in C$ è un nome di classe, $\mathbf{A} \subseteq A$ è un insieme finito di attributi, $\tau : \mathbf{A} \to \Gamma$ è una fuinzione totale, $Q \subseteq$ `STATES` è un insieme finito di stati, e $s \in Q$, $F \in Q$ sono gli stati *iniziale* e *finale*.

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 $q \in F$.

Il modello fin qui è piuttosto semplice. Una classe di artefatto è simile come idea ad una classe in un linguaggio di modellazione UML o ad una classe Java (tralasciamo per ora la possibilità di attributi multivalore, cioè che contengono liste). Una classe è dunque composta da una serie di attributi, che possono essere semplici o puntare ad un oggetto di un'altra classe. Ogni oggetto di una classe ha un identificatore unico. Un attributo può anche avere il valore `null`.

Aggiungiamo il concetto di stato in cui si può trovare un oggetto. Esso è iniziale, ovvero è stato appena creato, se si trova nello stato iniziale e tutti i suoi attributi sono `null`, ed è finale se si trova nello stato finale.

Mostriamo ora un esempio adattato da quello introduttivo, semplificando lo scenario e supponendo che un `Guest Check` contenga un solo piatto ed una sola specialità del giorno ordinata. Di seguito l'esempio:

- 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
```

Una classe C_2 è *referenziata da* un'altra classe C_1 se un attributo di C_1 ha come tipo C_2. In maniera similare, un identificatore o è *referenziato in* un artefatto **o** se o è il valore di qualche attributo di **o**.

Un artefatto **o'** estende un altro artefatto **o** se (1) hanno lo stesso identificatore (2) la funzione parziale di **o'** estende quella di **o** (**o** ed **o'** sono identici ad eccezione del fatto che qualche attributo, che in **o** è indefinito, in **o'** ha un valore).

Definizione 2.2.2. 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.

Sia S l'insieme di tali classi in Γ. S è *valido* se gli oggetti in S hanno identificatori distinti; S è *completo* se ogni identificatore di tipo C referenziato in qualche oggetto di S è l'identificatore di qualche altro oggetto in S.

Definizione 2.2.3. 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).

Ancora una volta risulta evidente il collegamento con i linguaggio OO ed UML. Un'istanza dello schema è un insieme di oggetti tali che (1) i loro identificatori sono distinti e (2) non è possibile puntare ad un altro eggetto se quest'ultimo non esiste.

2.2.2 I Servizi

Un servizio è descritto dalle variabili input, una precondizione e un insieme di effetti condizionali. Sempre seguendo la strada tracciata in [**?**], l'output dei servizi corrisponde alle modifiche che essi effettuano sugli artefatti. Come anche in OWL-S, il non-determinismo caratterizza la semantica del modello, nel senso che l'esecuzione di un servizio comporta uno dei possibili effetti condizionali definiti per esso.

Forniamo ora le prime definizioni.

Definizione 2.2.4. L'insieme di *termini* per uno schema Γ include:

- Variabili di una classe C in Γ , e
- $x.A$, dove x è un termine di qualche classe C (in Γ) e A un attributo in C.

Definizione 2.2.5. 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), dove t è un termine della classe C e A un attributo in C,
- NEW(t, A), dove t è un termine della classe C e A un attributo di tipo artefatto in C, e
- $s(t)$ (*atomo di stato*), dove t è un termine della classe C ed s uno stato di C.

Un *atomo negato* prende la forma $\neg c$ dove c è un atomo. Una *condizione* su Γ è una congiunzione di atomi (postivi o negati). Una condizione è *stateless* se non contiene atomi di stato.

Ovviamente, quando definiamo una formula dobbiamo avere anche un'assegnazione per le variabili (un mapping tale che a variabili di classe C è associato un ID in $\mathbf{ID}_C$.

DEFINED(t, A) è vero, banalmente, 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 (che dunque corrisponde ad un oggetto appena creato).

Sia V un insieme di variabili nello schema Γ. Un *effetto (condizionale)* su V è un insieme finito $E = \{\psi_1, \ldots, \psi_q\}$ di condizioni stateless su V. Ogni ψ_q è chiamato un *effetto potenziale* di E. Intuitivamente, se un servizio s con effetto condizionale E è applicato ad un'istanza I, allora l'istanza risultante soddisferà ψ_q per qualche $p \in [1..q]$. Vedremo che, come in [**?**], nella semantica del servizio viene incorporata una condizione basata sul concetto di circumscription.

Descriviamo ora la sintassi e la semantica dei servizi, assumendo l'esistenza di un insieme infinito di elementi distinti Θ contenente *nomi di servizio*.

Definizione 2.2.6. 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.

Mostriamo qui di seguito un esempio di servizio che aggiunge un piatto del menu ed una specialità al `Guest Check`:

```
---------------------------------
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 2.2.7. Sia $\sigma = (n, V_r, V_w, P, E)$ un servizio per lo schema Γ. La *semantica* di σ è un insieme $[[\sigma]] \subseteq V \times inst(\Gamma) \times inst(\Gamma)$ tale che per ogni $I \in inst(\Gamma)$ e assegnazione ν per I su $V_r \cup V_w$,

- C'è almeno un J con $(\nu, I, J) \in [[\nu]]$ se e solo se $I \models P[\nu]$

- se $(\nu, I, J) \in [[\sigma]]$ allora c'è qualche effetto pontenziale $\psi \in E$ per il quale esistono due insiemi K_{prev} e K_{new} di artefatto per lo schema Γ aventi insiemi disgiunti di identificatori di artefatti tali che:
  - NEW(t, A), dove t è un termine della classe C e A un attributo di tipo artefatto in C, e
  - $K = K_{prev} \cup K_{new}$ è un istanza di Γ.
  - Gli artefatti di I e K_{prev} hanno lo stesso insieme di indentificatori.
  - Per ogni identificatore di artefatto o che occore in I, la classe e lo stato di o in K_{prev} sono identici a quelli di o in I.
  - Per ogni atomo ψ della forma NEW(t, A) c'è un identificatore di artefatto distinto o in K_{new}, tale che $\nu(t.A) = o$. Inoltre, ogni identificatore di artefatto o in K_{new} corrisponde a qualche atomo della forma NEW(t, A) che si trova ψ.
  - Per ogni artefatto $o \in \mathbf{ID}_C$ in K_{new}, o è nello stato iniziale per C.
  - $(K_{prev} \cup K_{new}) \models \psi[\nu]$.
  - *(Circumscription)* Supponiamo che $o \in \mathbf{ID}_C$ occorra in K, e sia A un attributo di C. Supponiamo inoltre che $o \neq t[\nu]$ per ogni termine t che occorre in atomi di ψ della forma:
    - $t.A = t'.B$ o $t'.B = t.A$ per qualche attributo B;
    - DEFINED(t, A) occorre in ψ;
    - NEW(t, A)

    Allora vale il seguente:
    - Se o occorre in I, allora $o.A$ è definito in K se e solo se lo è in I.
    - Se o occorre in K_{new}, allora $o.A$ è indefinito in K.

Questa definizione stabilisce come si trova il sistema dopo l'invocazione di un servizio. Intuitivamente, K_{new} rappresenta l'insieme degli identificatori relativi ai nuovi insiemi creati. Tutti i nuovi oggetti devono essere nello stato iniziale. Questi più i vecchi oggetti (che sono eventualmente stati modificati) soddisfano uno dei possibili effetti condizionali. Se c'è un predicato NEW(t, A) in ψ allora ad esso corrisponde in nuovo identificativo in K_{new}, e viceversa. Infine, la condizione di circupscription cattura il fatto che non cambia nulla nell'istanza iniziale ad eccezione di ciò che è necessario per soddisfare ψ.

É utile considerare, al fine di sviluppare risultati positivi circa la complessità della verifica sul sistema, servizi che siano *monotoni*. Ciò significa che un attributo può essere scritto una sola volta, senza essere sovrascritto, riassegnato ad un valore precedente o reso idefinito.

Definizione 2.2.8. Siano I e J due istanze dello schema Γ. Allora J *estende* I se per ogni ID o in I, o occorre in J e $J(o)$ estende $I(o)$.

Definizione 2.2.9. Un servizio σ è *monotono* se J estende I per ogni $(\nu, I, J) \in [[\nu]]$.

Restringiamo ulteriormente il modello prevedendo che un servizio possa modificare un solo attributo.

Definizione 2.2.10. Il servizio $\sigma = (n, V_r, V_w, P, E)$ è *atomico* se è monotono, $V_w = \{x\}$ è un insieme costituito da un solo elemento, e per ogni $(\nu, I, J) \in [[\nu]]$, I e J differisono solo nei seguenti modi:

(a) Al massimo per un ID o e un attributo A di o, $I(o).A$ è indefinito e $J(o).A$ è definito.

(b) Se in (a) il tipo di A è una classe C, allora J ha un ID che I non ha, ovvero $J(o).A$.

(c) L'insieme di identificatori in I è contenuto in quello di J.

2.2.3 Regole di Business

Definizione 2.2.11. 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 2.2.1. Illustriamo ora due regole in base all'esempio che abbiamo portanto avanti. Supponiamo che ci sia una regola che lancia il servizio che aggiunge il piatto al `Guest Check`. Supponiamo inoltre che, una volta ordinati i piatti, il conto venga chiuso.

- **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)

Date due istanze I,J di uno schema Γ, e un'assegnazione $\nu \in V$, ed una regola di business r, diciamo che I *deriva* J *usando* r *e* ν, scritto $I \overset{r,\nu}{\rightarrow} J$, se valgono le seguenti condizioni:

- $I \models \varphi[\nu]$ e $(\nu, I, J) \in [[\nu]]$, se r è la regola **se** φ **invoca** $\sigma(x_1, \ldots, x_l; y_1, \ldots, y_k)$
- $I \models \varphi[\nu]$ e J è identico ad I eccetto per il fatto che ogni $J(\nu(x_i))$ ha lo stato in accordo con ψ, se r è la regola **se** φ **cambia stato in** ψ.

Vediamo dunque che le regole si occupano di attivare i servizi e sono le uniche a poter agire in base agli stati ed effettuare cambiamenti ad essi. Inoltre non possiamo asserire un cambiamento di stato in maniera negativa (non possiamo dire cioè che una formula φ implica che qualche oggetto **non** si trova in un certo stato).

2.2.4 Il Sistema e la sua Semantica

Un modello di business è catturato dal concetto di *sistema (di artefatti)*.

Definizione 2.2.12. Un *sistema (di artefatti)* è una tripla $W = (\Gamma, \Theta, R)$ dove Γ è uno schema, Θ è una famiglia di servizi su Γ, ed R è una famiglia di regole di business rispetto a Γ e Θ.

Definizione 2.2.13. Sia $W = (\Gamma, \Theta, R)$ uno schema e C una classe in Γ. Un *cammino* in W è una sequenza finita $\pi = I_0, I_1, \ldots, I_n$ di istanze di Γ. Il cammino è *valido* se per ogni $j \in [1..n]$, I_j è il risultato dell'applicazione di qualche regola di business r di R ad I_{j-1}.

Concludiamo questa parte introducendo, informalmente, le ultime due definizioni che ci permetteranno di illustrare i risultati di complessità.

Un cammino si dice *o-focused* se:

- o viene creato al primo passo
- o compare in ogni altro passo
- o ad ogni passo cambia stato o cambia il valore di qualche suo attributo
- ogni altro oggetto rimane immutato in tutto il cammino

Il cammino può terminare con o in uno stato finale. Se il cammino non può essere esteso per terminare in uno stato finale si dice *dead-end*.

Sia invece $\rho_{C.A}(W)$ un sistema derivato da W ma con l'attributo A di classe C completamente rimosso.

Definizione 2.2.14. un attributo A di classe C è ridondante se per ogni oggetto e per ogni cammino o-focused in W, il cammino in $\rho_{C.A}(W)$ è equivalente.

2.2.5 Risultati Teorici

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?

Teorema 2.2.1. *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 o-focused.*

Dal momento che il modello più generale è spesso indecidibile, cerchiamo alcune restrizioni che permettano di giungere a risultati di trattabilità od NP-completezza. Ci focalizziamo dunque su sistemi i cui servizi siano monotoni.

Introduciamo un nuovo tipo di predicato di stato: Un atomo *previous-or-current-state* assume la forma $[prev\_curr]s(t)$. Sia $I_0, I_1, \ldots, I_n$ un cammino. Allora [prev_curr]$s(t)$ è vero se t si trova nello stato s ora o precedentemente.

Teorema 2.2.2. *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).*

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.*

Rilassando anche una delle precedenti condizioni si ottiene un risultato di NP-completezza per **Q1**.

Teorema 2.2.3. *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.

Teorema 2.2.4. *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.*

Teorema 2.2.5. *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.*

2.3 Costruzione Automatica di Semplici Workflow Artifact-based

In questa sezione presentiamo il lavoro che si trova in [**?**].

L'articolo 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.

Il modello è meno generale di quello utilizzato nella sezione precedente, dal momento che il concetto di stato di un artefatto non viene modellato. In seguito, ci si focalizzerà inoltre su una singola classe di artefatti (mentre come abbiamo visto il modello più generale ammette diverse classi) e su una singola istanza di tale classe (si suppone che oggetti diversi di tale classe evolvano indipendentemente).

I servizi sono modellati sempre come non-deterministici.

2.3.1 Un primo esempio

Gli artefatti vengono modellati in maniera piuttosto semplice. Prevediamo che siano essenzialmente insiemi di coppie attributo-valore (dunque una serie di attributi di tipi semplici, come stringhe o interi). Un attributo può avere un valore od essere indefinito. Abbiamo inoltre a disposizione una serie di attributi che sono sempre inizializzati.

Esempio 2.3.1. Un esempio di classe (di artefatto), chiamata `PurchaseOrder`, è mostrata in figura **??**.

```
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 2.3.1: La classe `PurchaseOrder`

In figura **??** è mostrata una famiglia di servizi che operano su PurchaseOrder. I servizi sono definiti nella maniera classica. Possono essere eseguiti solo se la precondizione PRE è soddisfatta, e sono caratterizati da una serie di effetti condizionali EFF. Il predicato DEF è definito intuitivamente, ovvero assume il valore `true` se l'attributo a cui si riferisce contiene un valore.

La base su cui andremo a costruire il nostro insieme di regole è un (*pre-schema*), ovvero una coppia (A, S), dove A è una famiglia di classi di artefatti ed S una famiglia di servizi operanti su di essi. Chiamiamo `Purchasing` lo schema d'esempio.

Dato un pre-schema P, un cammino per P è una sequenza $\vec{s} = s_1, \ldots, s_n$ di *snapshot*. Lo stato dell'intero sistema corrisponde dunque ad una *fotografia* del sistema stesso, ovvero la combinazione di valori che detengono tutti gli attributi in gioco.

Dato uno stato s_i, s_{i+1} corrisponde al nuovo stato del sistema dopo l'applicazione di un qualche servizio.

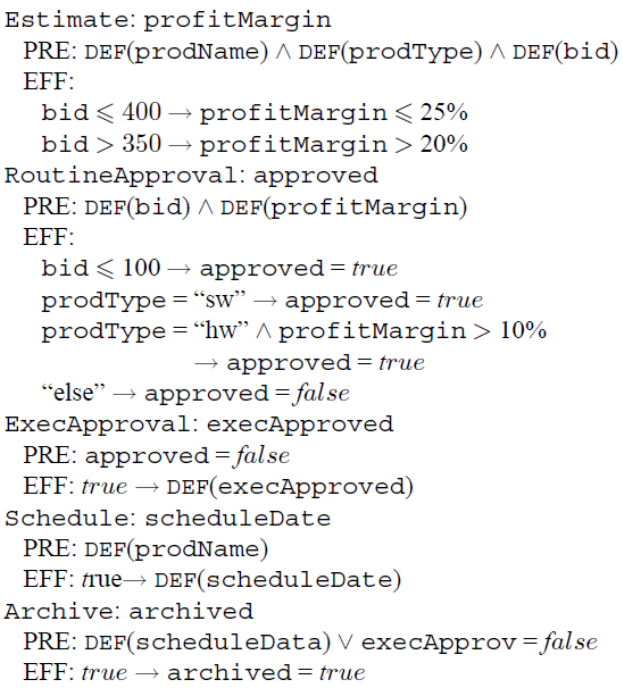

Figura 2.3.2: Famiglia di servizi associati a `PurchaseOrder`

Si nota dunque che, il modello è del tutto simile a quello introdotto in [], ad eccezione del fatto che gli stati degli artefatti non sono presi in considerazione e neanche la creazione degli stessi.

Nel cammino $\vec{o} = o_1, \ldots, o_n$ per un singolo artefatto o, per ogni i, l'istanza o_{i+1} corrisponde all'applicazione di qualche servizio o_i.

Come accennato, dunque, dato un pre-schema $P = (A, S)$ siamo interessati a costruire uno schema completo W, ovvero identificare anche un insieme di regole, tale che ogni sua esecuzione soddisfi un determinato goal, che assume in questo caso la forma di una formula γ.

La prima questione che si pone è la seguente:

Quesito 2.3.1 (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?

Il quesito si chiede intuitivamente se ha senso cercare di costruire un insieme di regole a partire da un dato pre-schema.

Esempio 2.3.2. 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`.

Come accennato pocanzi, dobbiamo ora individuare un insieme di regole R conformi al goal γ. La forma con la quale sono scritte le regole ricalca quella di [**?**], eccezion fatta per la mancanza del concetto di stato.

Mostriamo dunque un esempio di inseme di regole.

Esempio 2.3.3. La figura **??** fornisce una rappresentazione diagrammatica di un insieme di regole per `Purchasing`. 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`

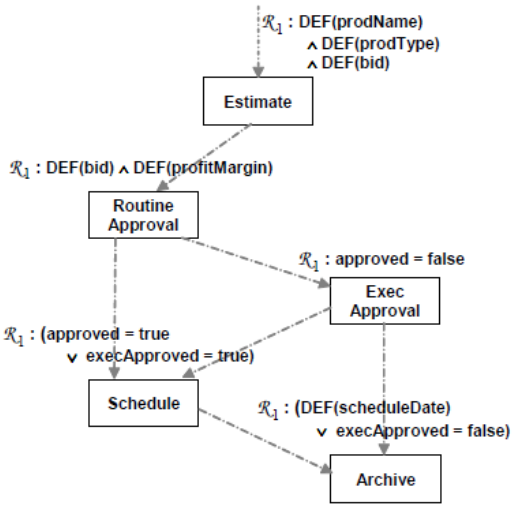

Figura 2.3.3: Insieme di regole γ_1-safe

Dato un pre-schema $P = (A, S)$ e un insieme di regole R per P, diciamo che $W = (A, S, R)$ è uno schema (n.b.: la verità di un antecedente di una regola non implica necessariamente che il corrispondente servizio venga lanciato).

Introduciamo ora il concetto di sicurezza di uno schema. Dato uno schema $W = (A, S, R)$, esso è γ_1-*safe* se ogni esecuzione (non-estendibile) di W termina in uno stato che soddisfa γ_1.

In riferimento al precedente esempio, `Purchasing`, esteso da R_1 è γ_1-safe.

Introdotto ora anche il concetto di sicurezza di uno schema, presentiamo il secondo quesito:

Quesito 2.3.2 (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 (una definizione più rigorosa verrà fornita nel paragrafo successivo).

In riferimento all'esempio precedente, si può dimostrare che R_1 è γ_1-safe mssimale per `Purchasing`.

Esempio 2.3.4. 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*.

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

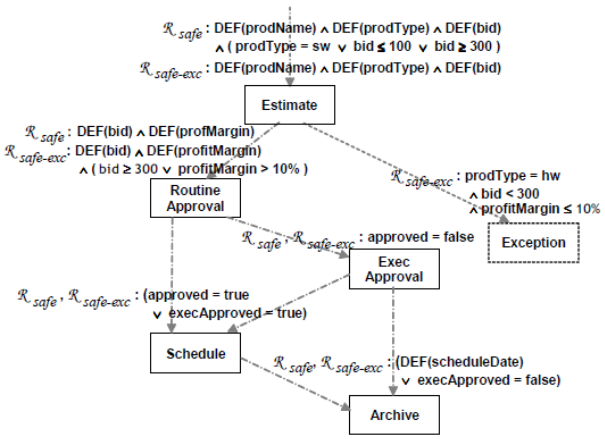

Figura 2.3.4: Famiglia di regole alternativa per γ_2

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}, corrispondente alla coppia $(A^{\texttt{exc}}, S^{\texttt{exc}} \cup \{EXC\})$ dove

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

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

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

Ovviamente dobbiamo avere a disposizione un insieme di regole che rilevino la condizione di non validità.

Un insieme di regole è dunque γ-safe con eccezioni per P se (i) è γ-safe per P^{EXC}, (ii) ogni esecuzione attraverso P che soddisfa γ è un'esecuzione attraverso R, e (iii) se in un'esecuzione $\vec{s}$ gli attributi già definiti di un artefatto o implicano che non ci siano $\vec{s}$ che soddisfano γ, allora in R questa esecuzione è immediatamente inviata al servizio EXC.

Il terzo quesito si basa su una riformulazione del secondo per esetnderlo agli schemi con eccezioni.

Quesito 2.3.3 (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?

Esempio 2.3.5. L'insieme $R_{safe-exc}$ di figura **??** è γ_2-safe massimale con eccezioni per `Purchasing`.

2.3.2 Un Modello Formale

Presentiamo ora le intuizioni mostrate nel paragrafo precedente in maniera più formale, così da poter mostrare i risultati teorici.

Supponiamo per semplicità che vi sia un solo artefatto ed assumiamo l'esistenza dei seguenti insiemi infiniti numerabili:

- ATT = $(A, B, \ldots)$ di *attributi*, e
- SERV = $(\sigma, \sigma_1, \ldots)$ of *nomi di servizi*.

Definizione 2.3.1. Una *classe di artefatto* **A** è un insieme finito di attributi in ATT. Un'istanza A è un mapping o da **A** a $U \cup \{\perp\}$.

Se o è un artefatto, o' denota l'istanza derivante dall'applicazione di un servizio ad o.

Definizione 2.3.2. Sia **A** una classe. Un servizio **A** è una tupla $(\sigma, R, W, \pi, \rho)$, dove $\sigma \in$ SERV è un nome di servizio, R,W sono insiemi finiti (di lettura e scrittura rispettivamente) di attributi in **A**, π e ρ sono formule senza quantificatori che menzionano solo attributi rappresentanti rispettivamente precondizioni e postcondizioni.

La semantica di un servizio è definita come segue: se tutti gli attributi in R sono definiti in o e o soddisfa la precondizione π, il servizio σ può essere invocato. Se invocato, esso modifica o in o' in maniera tale che (1) $A = A'$ per ogni attributo $\in$ **A** - W, e (2) o' soddisfa ρ. In tal caso scriviamo $o \xrightarrow{\sigma} o'$.

Definizione 2.3.3. Un *pre-schema* è una coppia **P** = (**A**, **S**) dove **A** è una classe di artefatto ed **S** è un insieme finito di servizi per **A**. Per ogni coppia di istanze di artefatto o_1, o_2 di **A**, o_1 deriva o_2 in **P**, scritto $o_1 \rightarrow_P o_2$, se $o_1 \xrightarrow{\sigma} o_2$ per qualche servizio $\sigma \in$ **S**.

Fin qui il modello ricalca perfettamente quello definito in [**?**], con le dovute restrizioni.

Per quanto riguarda le regole di attivazione dei servizi, utilizziamo sottoinsiemi L' di L (logica del primo ordine - assumiamo una struttura indicata con **M**).

Definizione 2.3.4. Una *L'-condizione* di una classe di artefatto **A** è una formula φ in L' che ha libere le sole variabili che rappresentano attributi. Una L'-condizione φ è *testabile* se è possibile determinare la sua soddisfacibilità.

Definizione 2.3.5. Dato un pre-schema **P**, una L'-regola è una espressione della forma **if** φ **invoke** σ, dove φ è una L'-condizione testabile, e σ è un servizio in **P**. Un artefatto o_2 è derivato da un altro artefatto o_1 *utilizzando una regola* r, scritto $o_1 \xrightarrow{r} o_2$, se $o_1 \models \varphi$ e $o_1 \xrightarrow{\sigma} o_2$. Un insieme di regole è un insieme finito di L'-regole.

Definizione 2.3.6. Uno schema è una tripla **W** = (**A**,**S**,**R**) dove **P** = (**A**, **S**) è un pre-schema ed **R** un insieme finito di regole per **P**. **W** estende **P**. Un artefatto o_1 deriva un altro artefatto o_2 in **W**, scritto $o_1 \xrightarrow{W} o_2$, se $o_1 \xrightarrow{r} o_2$ per qualche regola r in **R**.

Sia $\mathbf{P} = (\mathbf{A}, \mathbf{S})$ un pre-schema. Un *insieme di input* per $\mathbf{P}$ è un sottoinsieme $A_{init} \subseteq A$. Un *goal* per $\mathbf{P}$ è una L'-condition soddisfacibile su $\mathbf{A}$.

Un *cammino* per un pre-schema $\mathbf{P} = (\mathbf{A}, \mathbf{S})$ con insieme di input A_{init} è una sequenza di istanze $\vec{o} = o_1, \ldots, o_n$ dove

1. $o_1 \models \mathrm{DEF}(A)$ se e solo se $A \in A_{init}$ per ogni $A \in \mathbf{A}$,
2. per ogni $1 \leq i < n$, $o_i \xrightarrow{P} o_{i+1}$.

Sia γ un goal. Il cammino $\vec{o}$ è γ-*safe* se $o_n \models \gamma$.

Sia $\mathbf{W} = (\mathbf{A},\mathbf{S},\mathbf{R})$ e A_{init} un insieme di input per $(\mathbf{A}, \mathbf{S})$. Un cammino $o_1, \ldots, o_n$ per $(\mathbf{A}, \mathbf{S})$ con insieme di input A_{init} è un'*execuzione* di R se per ogni i ∈ [1..n-1], $o_i \xrightarrow{r} o_{i+1}$ per qualche regola r in $\mathbf{R}$.

Definizione 2.3.7. Sia $\mathbf{P}$ un pre-schema, A_{init} un insieme di input, $\mathbf{R}$ ed $\mathbf{R'}$ due insiemi di regole (in L') per $\mathbf{P}$. Allora, $\mathbf{R}$ sussume $\mathbf{R'}$, scritto $\mathbf{R} \succeq \mathbf{R'}$, se ogni esecuzione di $\mathbf{R'}$ in $\mathbf{P}$ e A_{init} è anche un esecuzione di $\mathbf{R}$.

Sia γ un goal. Un cammino (risp. un'esecuzione) è γ-dead-end se non è il prefisso di nessun'altro cammino γ-safe (risp. esecuzione).

Definizione 2.3.8. Sia $\mathbf{P} = (\mathbf{A}, \mathbf{S})$ un pre-schema, A_{init} un insieme di input, e γ un goal per $\mathbf{P}$. Un isieme di regole $\mathbf{R}$ per $\mathbf{P}$ è γ-*safe* se (1) ogni esecuzione di $\mathbf{W}$ è γ-safe e (2) $\mathbf{W} = (\mathbf{A},\mathbf{S},\mathbf{R})$ non ha dead-end.

Un insieme di regole $\mathbf{R}$ per $\mathbf{P}$ è γ-safe *massimale* se è γ-safe e, per ogni insieme di regole $\mathbf{R'}$ per $\mathbf{P}$ anch'esso γ-safe, allora $\mathbf{R} \succeq \mathbf{R'}$.

2.3.3 Costruzione dello schema completo

Siamo ora in possesso di tutte le nozioni che ci permettono di definire insiemi di regole sicuri e massimali.

In seguito 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 (ricordiamo che R e W sono gli insiemi di attributi di lettura e scrittura del servizio).

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:
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

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

.

Intuitivamente, dato un servizio σ e un goal γ, una $\forall$-precondition è una condizione **sufficiente** affinche 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.

Il concetto di *weakest precondition* è stato formulato da Dijkstra [**?**] nel contesto dei linguaggi di programmazione. Qui le weakest precondition assumono la forma di $\forall$-precondition.

In tale articolo veniva analizzata la derivazione formale di programmi *guarded*, ovvero caratterizzati da una speciale sintassi. L'attivazione di uno statement del linguaggio corrisponde all'esecuzione di un nostro servizio.

Una weakest precondition assume dunque il significato di condizione **necessaria e sufficiente** affinche l'attivazione di uno statement (ad esempio il comando $x = E$) porti il programma a terminare soddisfando un determinato goal.

Il calcolo della weakest precondition è anche usato nel Situation Calculus per il calcolo della *regressione*.

Riformuliamo pertanto il **Quesito 2.3.2** indicando un metodo di massima per risolverlo utilizzando le weakest $\forall$-precondition.

Quesito 2.3.4. Esiste un insieme di regole $\mathbf{R}$ γ-safe massimale per $\mathbf{P}$ e A_{init}? In caso affermativo, esiste un algoritmo che lo costruisce?

In un approccio semplicistico, il **Quesito 2.3.4** 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.

Per quanto riguarda i sistemi che permettono eccezioni, possiamo utilizzare le $\exists$-precondition per calcolare, dato un pre-schema P, un insieme di regole R. Sicuramente ogni cammino γ-safe di P è un'esecuzione di R, ma ci potrebbero essere esecuzioni dead-end.

Ciò che serve dunque è un meccanismo per creare le regole che attivano il servizio EXC. Supponiamo che σ sia un servizio e α_σ la disgiunzione di tutte le condizioni delle regole in R che attivano σ. Caratterizziamo quindi, con una formula δ, tutti quegli artefatti che possono derivare da dall'invocazione di σ su un artefatto che soddisfa α_σ (ovvero $\{s' | s \models \alpha \wedge s \xrightarrow{\sigma} s'\}$, e chiamiamo tale formula β) ma che non possono essere estesi ad esecuzioni γ-safe. Dunque, $\delta = \beta \wedge \neg(\bigvee\{\alpha_{\sigma'} | \sigma' \neq \sigma\})$. Per ogni formula δ creiamo una regola **if** δ **invoke** EXC.

Mostriamo ora una serie di risultati teorici riguardanti la soluzione dei nostri questiti in vari sottoinsiemi della logica del primo ordine.

Teorema 2.3.1. *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.

Teorema 2.3.2. *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.*

Teorema 2.3.3. *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 $\leq, +, \times$).*

2.3.4 Complessità di un Caso Speciale

In quest'ultima parte forniamo un'algoritmo più preciso per il calcolo dell'insieme di regole γ-safe massimale. 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 $W^{QF,<}$.

Teorema 2.3.4. *Dato un pre-schema $P = (A, S)$ in $W^{QF,<}$, un insieme fissato di attributi inizializzati, ed un goal γ, il problema di decidere se esiste un insieme di regole R γ-safe che permetta a tutti gli artefatti di input di entrare in (A, S, R) è PSPACE-completo.*

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:
   (a) siano $R_j =$ **se** α_j **allora invoca** σ_j le regole per quei servizi σ_j che definiscono A_{i+1}
   (b) Per ogni servizio σ che definisce A_i crea la regola **se** $wp^{\forall}(\sigma, \gamma) \bigwedge(\alpha_j)$ **allora invoca** σ

Intuitivamente, l'utilizzo delle *wp* fa sì che l'insieme sia γ-safe. L'ordinamento degli attributi e il porre in **and** le *wp* fa sì che l'insieme sia massimale.

2.4 Verfica Automatica di Processi di Business Data-Centric

In quest'ultima sezione presentiamo il lavoro in [**?**].

L'approccio di modellazione è leggermente differente da quelli introdotti precedentemente. Il concetto di identità di artefatto non è esplicitamente modellato e dunque si assume che ogni classe di artefatto (che è sostanzialmente uno schema relazionale) ammetta una sola istanza in ogni istante. L'approccio si presta naturalmente ad essere implementato in un database relazionale.

L'innovazione che introduce questo articolo consiste nell'utilizzo di un potente strumento logico per effetuare la verifica sul sistema, ovvero un'estensione della logica LTL al primo ordine, che chiameremo LTL-FO.

Come è noto, la verifica di formule in tali sistemi è indecidibile. Daremo come prima cosa una definizione formale del nostro sistema, poi verrà illustrato un esempio adattato dall'introduzione e infine introdurremo la definizione di sistema *guarded* che permette di superare l'ostacolo dell'indecidibilità e giungere ad importanti risultati teorici.

2.4.1 Framework

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.

Definizione 2.4.1. 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 $\mathbf{C} = (\mathbf{R},\mathbf{S})$, dove (i) $\mathbf{R}$, chiamata relazione di attributo, è un'interpretazione di R contenente esattamente una tupla su D, e (ii) $\mathbf{S}$, chiamata relazione di stato, è un'interpretazione finita di S su D.

Notiamo dunque che, a differenza dei precedenti modelli, lo stato è rappresentato da una relazione. In seguito si noterà che questa particolare struttura permette di arrivare ad importanti risultati teorici. Possiamo permettere inoltre più d'una relazione di stato; si può dimostrare infatti che tale estensione preserva la decidibilità. Gli stati booleani vengono modellati con relazioni nullarie $\mathbf{S()}$ che indichiamo semplicemente con $\mathbf{S}$.

Definizione 2.4.2. 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 2.4.3. 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 $\mathbf{DB}$ è un'istanza di DB su D.

Riepiloghiamo: abbiamo una serie di classi che rappresentano il livello intensionale degli artefatti. Ogni classe composta da una serie di attributi (coppie nome valore) e di relazioni di stato. L'unione delle classi e del DB rappresenta lo schema. $\mathbf{A}$ è un'istanza dello schema. $\mathbf{DB}$ rappresenta un database fissato e immutabile.

Passiamo ora a definire sintassi e semantica dei servizi.

Definizione 2.4.4. 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. $|\bar{x}| = a(S)$.

Dunque, ψ rappresenta la precondizione del servizio che può essere construita su stati ed attributi. La post-condizione ψ modifica le relazioni di attributo. Le regole di stato modificano le relazioni di stato.

Definizione 2.4.5. 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.

Definizione 2.4.6. 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 $\mathbf{A} \xrightarrow{\sigma} \mathbf{A'}$) se valgono le seguenti condizioni:

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`).

La semantica del servizio segue lo spirito di quella definita per le altre sezioni. Il database **DB**, come segue dalla definizione, non cambia.

Un'*istanza iniziale* di Γ è un'istanza di A le cui relazioni di stato sono vuote.

Definizione 2.4.7. 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.

Definiamo ora l'estensione della logica LTL per supportare asserzioni FOL.

Definizione 2.4.8. 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.

Sia A uno schema. Un enunciato LTL-FO su A è caratterizzato dal fatto che ogni sua componente è su DB_A. 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 $\varphi(\nu(\bar{x}))$. Il seguito è definito per induzione strutturale sulla formula. Il resto della semantica è standard.

Oltre a rispondere ai problemi sollevati nelle precedenti sezioni, possiamo verificare la consistenza del sistema e verificare proprietà arbitrarie circa l'evoluzione globale del sistema stesso.

2.4.2 Un esempio

Costruiamo un esempio per questo modello sulla base dello scenario introdotto nella prima sezione. Supponiamo di avere l'artefatto `GuestCheck`, che memorizza le informazioni del tavolo (cliente, numero tavolo, conto, etc..) ed una lista dei piatti ordinati. L'ordine avviene tramite l'artefatto `ORDINE`, che memorizza il piatto e la specialità del giorno ordinata. Quando l'ordine viene completato questi dati vengono trasferiti nel `GUESTCHECK`.

Il corrispondente sistema $\Gamma_{ex} = \langle A, \Sigma\rangle$ è qui parzialmente descritto:
Lo schema $A = \langle GUESTCHECK, ORDINE, DB\rangle$ è composto da:
$DB = \langle$`MENU`, `SPEC`$\rangle$ è lo schema del DB, dove:

- `MENU`(*nome_piatto*, *descrizione*, *prezzo*) è la relazione che memorizza i piatti standard del menu.
- `SPEC`(*nome_piatto*, *descrizione*, *prezzo*, *ingredienti*) è la relazione che memorizza le specialità giornaliere.

$GUESTCHECK = \langle G_O,$ `piatti`, `special`, `pasto`, `archiviato`$\rangle$ è la classe che memorizza le informazioni relative al `Guest Check`, dove:

- G_O(*#tavolo*, *conto*, *#clienti*, *cliente_principale*, *data*) è la relazione di attributo.
- `piatti`(*nome_piatto*) è una relazione di stato che memorizza i piatti del menu ordinati.
- `special`(*nome_spec*) è una relazione di stato che memorizza le specialità giornaliere ordinate.
- `pasto` e `archiviato` sono relazioni di stato nullarie che tengono traccia dello stadio in cui si trova l'artefatto.

Quando i clienti sono al tavolo, possono ordinare un numero arbitrario di piatti (o specialità). Finito il pasto il Guest Check viene archiviato.

$ORDINE = \langle R_O,$ `in_processamento`, `completato`$\rangle$
è la classe che memorizza le informazioni relative agli ordini per la cucina, dove:

- $R_O(piatto, specialità)$ è la relazione di attributo.
- `in_processamento` e `completato` sono relazioni di stato nullarie che tengono traccia dello stadio in cui si trova l'artefatto.

Un esempio di servizio potrebbe essere $istanzia\_ordine = \langle \pi^{rn}, \psi^{rn}, S^{rn} \rangle$, che invia un'ordinazione alla cucina, descritto come segue:

- $\pi^{rn} = \neg$ `in_processamento`,
  ovvero non c'è un'altra ordinazione gestita al momento
- La post-condizione ψ^{rn} è data da

  $$R_O(p, s) := p \neq \omega \wedge s \neq \omega \wedge \exists d, c, d', c', i(MENU(p, d, c) \wedge SPEC(s, d', c', i))$$

- S^{rn} contiene la seguente regola

  `in_processamento` $\leftarrow$ `true`,
  ovvero ora l'ordine si trova nello stato di processamento

Esempio 2.4.1. Un esempio di proprietà di consistenza da effettuare sul sistema è la seguente:

$$\mathbf{G}(\neg(\texttt{in\_processamento} \wedge \texttt{completato}))$$

2.4.3 Risultati teorici

Come accennato, 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* in logica FOL vengono modificate. Inoltre, le pre-e-post-condizioni dei servizi vengono ristrette al solo uso del quantificatore esistenziale.

Vediamo in dettaglio:

Definizione 2.4.9. 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, $\bar{x} \subseteq free(\alpha)$, e $\bar{x} \cap free(\beta) = \emptyset$ per ogni atomo di stato β in φ, allora $\exists \bar{x}(\alpha \wedge \varphi)$ e $\forall \bar{x}(\alpha \rightarrow \varphi)$ sono formule.

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). Un enucniato LTL-FO su A è guarded se e solo se lo sono tutte le sue componenti FO.

Diamo ora i due risultati teorici principali riguardanti questo tipo di sistemi.

Teorema 2.4.1. *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.*

Corollario 2.4.2. *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.*

2.4.4 Limiti di Verificabilità

In questa ultima parte vedremo come il rilassamento di una qualsiasi delle proprietà introdotte finora porti ad un condizione di non decidibilità. Siamo dunque in presenza del modello più generale possibile che peremtta verifiche decidibili.

Possiamo dimostrare infatti che 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:

Teorema 2.4.3. *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.

Corollario 2.4.4. *Dato un sistema guarded* Γ*, è indecidibile verificare se* Γ *ha qualche esecuzione bloccante.*

Bibliografia

[1] A. Nigam, N. S. Caswell. *Business artifacts: An Approach to operational specification.* IBM SYSTEMS JOURNAL, VOL 42, NO 3, 2003.

[2] K. Bhattacharya, N. S. Caswell, S. Kumaran, A. Nigam, F. Y. Wu. *Artifact-centered operational modeling: Lessons from customer engagements.* IBM SYSTEMS JOURNAL, VOL 46, NO 4, 2007.

[3] K. Bhattacharya, R. Hull, J. Su. *A Data-Centric Design Methodology for Business Processes.*

[4] J. M. Kuster, K. Ryndina, H. Gall. *Generation of Business Process Models for Object Life Cycle Compliance.*

[5] K. Bhattacharya, C. Gerede, R. Hull, R. Liu, J. Su. *Toward Formal Analysis of Artifact-centric Business Processes.*

[6] C. Fritz, R. Hull, J. Su. *Automatic Construction of Simple Artifact-based Workflows.*

[7] A. Deutsch, R. Hull, F. Patrizi, V. Vianu. *Automatic Verification of Data-Centric Business Processes.*

[8] E. Dijkstra. *Guarded Commands, Nondeterminacy and Formal Derivation of Programs.* Communication of the ACM, August 1975, Volume 18, Number 8.