Capitolo 3

Analisi e specifica dei requisiti

In questo capitolo presentiamo alcuni linguaggi e metodi usati nella fase di analisi e specifica dei requisiti. I requisiti descrivono ciò che l'utente si aspetta dal sistema, e *specificarli* significa esprimerli in modo chiaro, univoco, consistente e completo. I requisiti si distinguono in *funzionali* e *non funzionali*.

I requisiti funzionali descrivono cosa deve fare il sistema, generalmente in termini di relazioni fra dati di ingresso e dati di uscita, oppure fra *stimoli* (dall'ambiente al sistema) e *risposte* del sistema. Questi requisiti sono generalmente esprimibili in modo formale.

I requisiti non funzionali esprimono dei *vincoli* o delle caratteristiche di qualità. Queste ultime sono piú difficili da esprimere in modo formale, in particolare è difficile esprimerle in modo quantitativo. Fra le caratteristiche di qualità del software ricordiamo le seguenti:

sicurezza (safety): Capacità di funzionare senza arrecare danni a persone o cose (piú precisamente, con un rischio limitato a livelli accettabili). Si usa in questo senso anche il termine *innocuità*.

riservatezza (security): Capacità di impedire accessi non autorizzati ad un sistema, e in particolare alle informazioni in esso contenute. Spesso il termine *sicurezza* viene usato anche in questo senso.

robustezza: Capacità di funzionare in modo accettabile anche in situazioni non previste, come guasti o dati di ingresso errati.

prestazioni: Uso efficiente delle risorse, come il tempo di esecuzione e la memoria centrale.

usabilità: Facilità d'uso.

interoperabilità: Capacità di integrazione con altre applicazioni.

I requisiti di sicurezza, riservatezza e robustezza sono aspetti del piú generale requisito di *affidabilità* (*dependability*), nel senso piú corrente di questo termine. Ricordiamo che le definizioni piú rigorose di questo termine si riferiscono alla probabilità che avvengano malfunzionamenti entro determinati periodi di tempo.

3.1 Classificazioni dei sistemi software

Nell'affrontare l'analisi dei requisiti, è utile individuare certe caratteristiche generali del sistema che dobbiamo sviluppare. A questo scopo possiamo considerare alcuni criteri di classificazione dei sistemi, che vedremo nel resto di questa sezione.

3.1.1 Requisiti temporali

Una prima importante classificazione delle applicazioni può essere fatta in base ai requisiti temporali, rispetto ai quali i sistemi si possono caratterizzare come:

sequenziali: senza vincoli di tempo;
concorrenti: con sincronizzazione fra processi;
in tempo reale: con tempi di risposta prefissati.

Nei sistemi sequenziali un risultato corretto (rispetto alla specifica del sistema in termini di relazioni fra ingresso e uscita) è accettabile qualunque sia il tempo impiegato per ottenerlo. Naturalmente è sempre desiderabile che l'elaborazione avvenga velocemente, ma la tempestività del risultato non è un requisito funzionale, bensí un requisito relativo alle prestazioni o all'usabilità. Inoltre, un sistema è sequenziale quando è visto come un singolo processo, le cui interazioni con l'ambiente (operazioni di ingresso e di uscita) avvengono in una sequenza prefissata.

I sistemi concorrenti, invece, sono visti come insiemi di processi autonomi (eventualmente *distribuiti*, cioè eseguiti da piú di un elaboratore) che in alcuni momenti possono comunicare fra di loro. Le interazioni reciproche dei processi, e fra questi e l'ambiente, sono soggette a vincoli di sincronizzazione ed avvengono in sequenze non determinate a priori. Per *vincoli di sincronizzazione* si intendono delle relazioni di precedenza fra eventi, come, per esempio, "*l'azione* a *del processo* P *deve essere eseguita dopo l'azione*

b *del processo* Q", oppure "*la valvola n. 2 non si deve aprire prima che si sia chiusa la valvola n. 1*". In generale, un insieme di vincoli di sincronizzazione su un insieme di processi interagenti può essere soddisfatto da diverse sequenze di eventi, ma il verificarsi di sequenze che violano tali vincoli è un malfunzionamento.

Nei sistemi concorrenti il tempo impiegato per l'elaborazione, come nei sistemi sequenziali, non è un requisito funzionale. Come esempio molto semplice di sistema concorrente possiamo considerare il comando `cat` del sistema Unix combinato per mezzo di un *pipe* al comando `lpr`:

```
cat swe.txt | lpr
```

Il processo `cat` legge un file e ne scrive il contenuto sull'uscita standard, l'operatore *pipe* collega l'uscita di `cat` all'ingresso del processo `lpr`, che scrive sulla stampante. I due processi lavorano a velocità diverse, ma sono sincronizzati in modo che il processo piú veloce (presumibilmente `cat`) aspetti il piú lento. Il vincolo di sincronizzazione si può esprimere informalmente cosí: "*il processo* `lpr` *deve scrivere i caratteri nello stesso ordine in cui li riceve dal processo* `cat`".

Questo esempio rientra nel modello della coppia produttore/consumatore con controllo di flusso (*flow control*), in cui il produttore si blocca finché il consumatore non è pronto a ricevere nuove informazioni: in questo caso, i due processi hanno un vincolo di sincronizzazione reciproca (il consumatore elabora un dato solo dopo che il produttore lo ha prodotto, il produttore elabora un nuovo dato solo se il consumatore lo può ricevere), ma non esistono limiti prefissati per il tempo di esecuzione.

Un sistema in tempo reale è generalmente un sistema concorrente, e in piú deve fornire i risultati richiesti entro limiti di tempo prefissati: in questi sistemi un risultato, anche se corretto, è inaccettabile se non viene prodotto in tempo utile. Per esempio, una coppia produttore/consumatore *senza* controllo di flusso, in cui il produttore funziona ad un ritmo indipendente da quello del consumatore, è un sistema real-time, poiché in questo caso il produttore impone al consumatore un limite massimo sul tempo di esecuzione. Se il consumatore non rispetta questo limite si perdono delle informazioni. Un esempio di questo tipo di sistema è un sensore che manda informazioni a un elaboratore, dove la frequenza di produzione dei dati dipende solo dal sensore o dall'evoluzione del sistema fisico controllato.

3.1.2 Tipo di elaborazione

Un'altra classificazione dei sistemi si basa sul tipo di elaborazione compiuta prevalentemente. I sistemi si possono quindi caratterizzare come:

orientati ai dati: mantengono e rendono accessibili grandi quantità di informazioni (p.es., banche dati, applicazioni gestionali);

orientati alle funzioni: trasformano informazioni mediante elaborazioni complesse (p.es., compilatori);

orientati al controllo: interagiscono con l'ambiente, modificando il proprio stato in seguito agli stimoli esterni (p.es., sistemi operativi, controllo di processi).

Bisogna però ricordare che ogni applicazione usa dei dati, svolge delle elaborazioni, ed ha uno stato che si evolve, in modo piú o meno semplice. Nello specificare un sistema è quindi necessario, in genere, prendere in considerazione tutti questi tre aspetti.

3.1.3 Software di base o applicativo

Un'altra classificazione si può ottenere considerando se il software da realizzare serve a fornire i servizi base di elaborazione ad altro software (per esempio, se si deve realizzare un sistema operativo o una sua parte), oppure software intermedio fra il software di base e le applicazioni (librerie), oppure software applicativo vero e proprio.

3.2 Linguaggi di specifica

Normalmente i requisiti, sia funzionali che non funzionali, vengono espressi in linguaggio naturale. Questo, però, spesso non basta a specificare i requisiti con sufficiente precisione, chiarezza e concisione. Per questo sono stati introdotti numerosi *linguaggi di specifica* che possano supplire alle mancanze del linguaggio naturale.

3.2.1 Classificazione dei formalismi di specifica

I formalismi usati nella specifica dei sistemi privilegiano in grado diverso gli aspetti dei sistemi considerati piú sopra. Alcuni formalismi sono specializ-

zati per descrivere un aspetto particolare, mentre altri si propongono una maggiore generalità. Alcune metodologie di sviluppo si affidano ad un unico formalismo, mentre altre ne sfruttano piú d'uno.

I formalismi per la specifica, quindi, possono essere suddivisi analogamente ai tipi di sistemi per cui sono concepiti, per cui si avranno formalismi orientati ai dati, alle funzioni, e via dicendo. Inoltre, i formalismi di specifica vengono classificati anche secondo i due criteri del *grado di formalità* e dello *stile di rappresentazione*, che descriviamo di séguito.

Grado di formalità

I linguaggi si possono suddividere in *formali*, *semiformali*, *informali*.

Un linguaggio è *formale* se la sua sintassi e la sua semantica sono definite in modo matematicamente rigoroso; il significato di questa frase sarà meglio definito nella parte dedicata alla logica, dove vedremo come la sintassi e la semantica di un linguaggio si possano esprimere per mezzo di concetti matematici elementari, come insiemi, funzioni e relazioni.

Una specifica espressa in un linguaggio formale è precisa e verificabile, e inoltre lo sforzo di traduzione dei concetti dal linguaggio naturale a un linguaggio formale aiuta la comprensione di tali concetti da parte degli analisti. Questo è dovuto al fatto che, dovendo riformulare in un linguaggio matematico un concetto espresso in linguaggio naturale, si è costretti ad eliminare le ambiguità e ad esplicitare tutte le ipotesi date per scontate nella forma originale.

Il maggiore limite dei linguaggi formali è il fatto che richiedono un certo sforzo di apprendimento e sono generalmente poco comprensibili per chi non abbia una preparazione adeguata. Conviene però ricordare che la sintassi dei linguaggi formali può essere resa piú amichevole, per esempio usando notazioni grafiche.

Per notazioni o linguaggi *semiformali* si intendono quelle che hanno una sintassi (spesso grafica) definita in modo chiaro e non ambiguo ma non definiscono una semantica per mezzo di concetti matematici, per cui il significato dei simboli usati viene espresso in modo informale. Nonostante questo limite, i linguaggi semiformali sono molto usati perché permettono di esprimere i concetti in modo piú conciso e preciso del linguaggio naturale, e sono generalmente piú facili da imparare ed usare dei linguaggi formali.

Fra i linguaggi semiformali possiamo includere il *linguaggio naturale strutturato*, cioè il linguaggio naturale usato con una sintassi semplificata e varie convenzioni che lo rendano piú chiaro e preciso.

I linguaggi *informali* non hanno né una sintassi né una semantica definite rigorosamente. I linguaggi naturali hanno una sintassi codificata dalle rispettive grammatiche, ma non formalizzata matematicamente, ed una semantica troppo ricca e complessa per essere formalizzata. Al di fuori del linguaggio naturale non esistono dei veri e propri linguaggi informali, ma solo delle notazioni grafiche inventate ed usate liberamente per schematizzare qualche aspetto di un sistema, la cui interpretazione viene affidata all'intúito del lettore ed a spiegazioni in linguaggio naturale. I disegni fatti alla lavagna durante le lezioni rientrano in questo tipo di notazioni.

Stile di rappresentazione

Un'altra possibile suddivisione è fra linguaggi *descrittivi* e *operazionali*.

La differenza fra questi linguaggi può essere compresa considerando, per esempio, un semplice sistema costituito da un contenitore di gas con una valvola di scarico. I requisiti di sicurezza esigono che quando la pressione p del gas supera un certo valore di soglia P la valvola si apra, e quando la pressione torna al di sotto del valore di soglia la valvola si chiuda. Possiamo esprimere questo requisito con una formula logica:

$$p < P \Leftrightarrow \textbf{valvola-chiusa}$$

in cui **valvola-chiusa** è una proposizione che è vera quando la valvola è chiusa. In questo caso, il sistema è rappresentato da una relazione logica fra le proprietà delle entità coinvolte, e si ha quindi una rappresentazione descrittiva (o *dichiarativa*).

Lo stesso sistema può avere una rappresentazione operazionale, cioè in termini di una *macchina astratta*, che si può trovare in certo *stato* e passare ad un altro stato (effettuando cioè una *transizione*) quando avvengono certi *eventi*. La figura 3.1 mostra la macchina astratta corrispondente all'esempio, che si può trovare nello stato "aperto" o "chiuso" secondo il valore della pressione. Le espressioni $(p \geq P := T)$ e $(p < P := T)$ denotano gli eventi associati al passaggio della pressione a valori, rispettivamente, maggiori o minori della soglia P (la prima espressione, per esempio, significa "$p \geq P$ diventa vero").

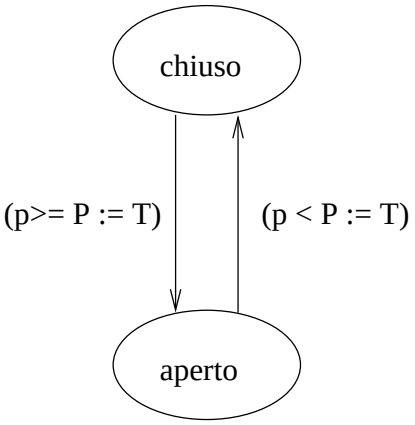

Figura 3.1: Una descrizione operazionale

Quindi i linguaggi descrittivi rappresentano un sistema in termini di entità costituenti, delle loro proprietà, e delle relazioni fra entità, mentre i linguaggi operazionali lo rappresentano in termini di stati e transizioni che ne definiscono il comportamento. Spesso una specifica completa richiede che vengano usate tutte e due le rappresentazioni.

Osserviamo che, nel nostro esempio, la formulazione descrittiva esprime direttamente il requisito di sicurezza, mentre la formulazione operazionale descrive un comportamento del sistema tale che rispetti il requisito, ed è quindi meno astratta.

3.3 Formalismi orientati ai dati

I formalismi orientati ai dati si possono suddividere fra quelli usati per definire la *semantica* dei dati e quelli che ne definiscono la *sintassi*. I primi permettono di descrivere il significato delle informazioni associate ai dati, mentre i secondi descrivono la struttura dei dati, ovvero la forma in cui si presentano.

Fra i formalismi orientati ai dati citiamo soltanto, rimandandone lo studio ad altri corsi o all'iniziativa dello studente, il *modello Entità–Relazioni* (di tipo semantico), le *espressioni regolari*, le *grammatiche non contestuali* e la *Abstract Syntax Notation 1* (di tipo sintattico).

L'uso dei linguaggi di specifica orientati agli oggetti per la descrizione dei dati verrà trattato nella Sezione 3.6.

3.4 Formalismi orientati al controllo

Questi formalismi descrivono gli aspetti dei sistemi relativi alla loro evoluzione temporale, alle possibili sequenze di eventi o di azioni, alla sincronizzazione fra le attività dei loro sottosistemi, o fra il sistema e l'ambiente in cui opera. Questi aspetti sono particolarmente importanti nei sistemi *reattivi*, che devono reagire a stimoli provenienti dall'ambiente, che si presentano in un ordine generalmente non prevedibile.

I formalismi orientati al controllo sono un campo di studio molto vasto e articolato. In questo corso verranno date solo alcune nozioni elementari relative al formalismo degli *automi a stati finiti* ed alle sue estensioni adottate nel linguaggio UML (Sezione 3.6.7).

Fra i formalismi che non verranno trattati nel corso, citiamo le *reti di Petri*, le *algebre di processo* e le *logiche temporali*.

3.4.1 Automi a stati finiti

Col formalismo degli *automi a stati finiti* (ASF) o *macchine a stati*[1] descriviamo un sistema attraverso gli *stati* in cui si può trovare, le *transizioni*, cioè i passaggi da uno stato all'altro, gli *ingressi* (stimoli esterni) che causano le transizioni, le *uscite* del sistema, che possono essere associate alle transizioni (*macchine di Mealy*) o agli stati (*macchine di Moore*), e lo *stato iniziale* da cui parte l'evoluzione del sistema.

L'ambiente esterno agisce sul sistema rappresentato dall'automa generando una successione di ingressi discreti, e il sistema risponde a ciascun ingresso cambiando il proprio stato (eventualmente restando nello stato corrente) e generando un'uscita. L'automa definisce le regole secondo cui il sistema risponde agli stimoli esterni.

Un ASF è quindi definito da una sestupla

$$< S, I, U, d, t, s_0 >$$

con

- S insieme degli stati

[1] Precisiamo che sono stati descritti diversi tipi di macchine a stati; nel seguito faremo riferimento a uno dei tipi usati piú comunemente, i *trasduttori deterministici*.

- I insieme degli ingressi
- U insieme delle uscite
- $d : S \times I \to S$ funzione di transizione
- $t : S \times I \to U$ funzione di uscita (macchine di Mealy)
- $s_0 \in S$ stato iniziale

Nelle macchine di Moore si ha $t : S \to U$.

Un ASF viene rappresentato con un grafo orientato, i cui nodi (cerchi o rettangoli ovalizzati) sono gli stati, e gli archi orientati, etichettati dagli ingressi e dalle uscite, descrivono la funzione di transizione e la funzione di uscita. Lo stato iniziale viene indicato da un'arco senza stato di origine.

L'automa rappresentato in Fig. 3.2 descrive l'interazione fra un utente ed un centralino che accetta chiamate interne a numeri di due cifre e chiamate esterne a numeri di tre cifre, precedute dallo zero (esempio da [14]). Nel diagramma, le transizioni sono etichettate con espressioni della forma '*ingresso*/*uscita*'; alcune transizioni non producono uscite. Un ingresso della forma '*m*:*n*' rappresenta una cifra fra *m* e *n*. Ogni stato è identificato sia da un nome che da un numero. Al numero si farà riferimento nella rappresentazione tabulare (v. oltre) dell'automa.

Un automa può essere rappresentato anche per mezzo di tabelle. Una rappresentazione possibile si basa su una matrice quadrata di ordine n (numero degli stati): se esiste una transizione dallo stato s_i allo stato s_j, l'elemento della matrice sulla riga i e la colonna j contiene l'ingresso che causa la transizione e la corrispondente uscita. La rappresentazione tabulare dell'ASF di Fig. 3.2 è data dalla tabella 3.4.1.

Componibilità e scalabilità

Nella specifica di sistemi complessi, e in particolare di sistemi concorrenti, è spesso necessario o conveniente rappresentare il sistema complessivo come un aggregato di sottosistemi. Un formalismo di specifica ha la proprietà della *componibilità* se permette di costruire la rappresentazione del sistema complessivo per mezzo di semplici operazioni di composizione sulle rappresentazioni dei sottosistemi. Per *scalabilità* si intende la capacità di rappresentare un sistema in modo tale che la complessità di tale rappresentazione sia dello stesso ordine di grandezza della somma delle complessità dei singoli sottosistemi.

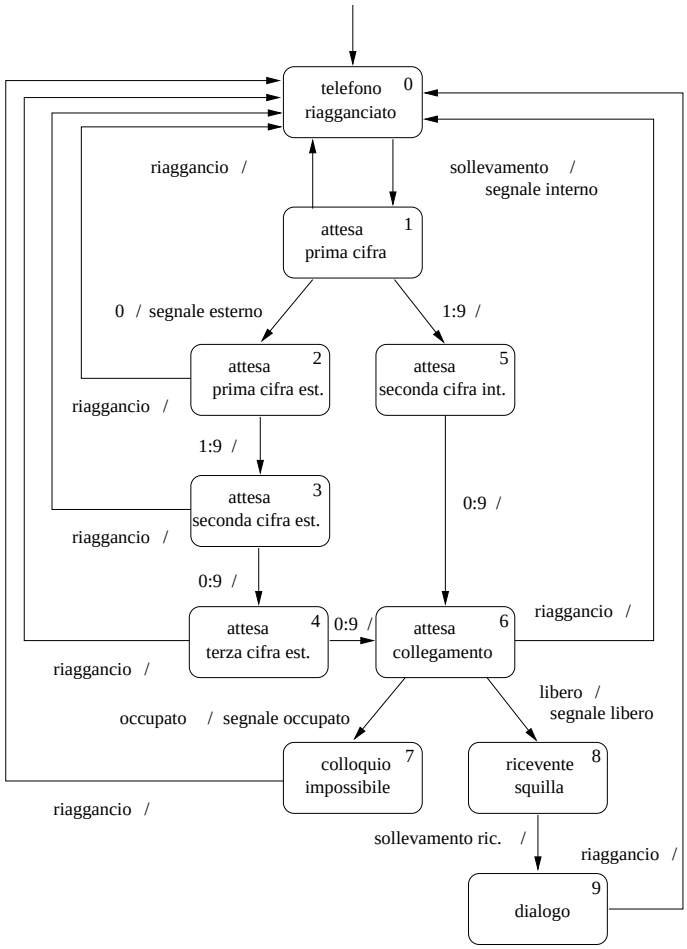

Figura 3.2: Un automa a stati finiti

Nel caso degli ASF la componibilità e la scalabilità sono limitate, come mostrerà l'esempio sviluppato nei paragrafi successivi.

La figura 3.3 mostra tre sottosistemi: un produttore, un magazzino ed un consumatore (esempio da [14]). Il produttore si trova inizialmente nello stato (p_1) in cui è pronto a produrre qualcosa (per esempio, un messaggio o una richiesta di elaborazione), e con la produzione passa allo stato (p_2) in cui attende di depositare il prodotto; col deposito ritorna allo stato iniziale. Il consumatore ha uno stato iniziale (c_1) in cui è pronto a prelevare qualcosa, e col prelievo passa allo stato (c_2) in cui attende che l'oggetto venga consumato. Il magazzino ha tre stati, in cui, rispettivamente, è vuoto (m_1), contiene un oggetto (m_2), e contiene due oggetti (m_3); il magazzino passa da uno stato all'altro in seguito alle operazioni di deposito e di prelievo.

	0	1	2	3	4	5	6	7	8	9
0		S/int.								
1	R/		0/est.			1:9/				
2	R/			1:9/						
3	R/				0:9/					
4	R/						0:9/			
5	R/						0:9/			
6	R/							O/occ.	L/lib.	
7	R/									
8										Sr/
9	R/									

Tabella 3.1: Rappresentazione tabulare

R: riaggancio; **S**: sollevamento del chiamante; **O**: ricevente occupato; **L**: ricevente libero; **Sr**: sollevamento del ricevente

In questo modello, gli ingressi *produzione*, *deposito*, *prelievo* e *deposito* rappresentano il completamento di attività eseguite dal sistema modellato. Questo non corrisponde all'idea intuitiva di "ingresso" di un sistema, ma descrive correttamente un sistema di controllo che riceve segnali dal sistema controllato.

Il sistema complessivo, mostrato in figura 3.4, ha dodici stati, invece dei sette usati per descrivere i sottosistemi separatamente: ciascuno dei dodici stati è una possibile combinazione degli stati dei sottosistemi. In generale, componendo un numero n di sottosistemi in un sistema complessivo, si ha che:

1. l'insieme degli stati del sistema complessivo è il prodotto cartesiano degli insiemi degli stati dei sottosistemi;
2. cioè, ogni stato del sistema complessivo è una n-upla formata da stati dei sottosistemi, per cui viene nascosta la struttura gerarchica del sistema (gli stati dei sottosistemi vengono concentrati nello stato globale);
3. l'evoluzione del sistema viene descritta come se ad ogni passo uno solo dei sottosistemi potesse compiere una transizione, mentre in generale è possibile che transizioni in sottosistemi distinti possano avvenire in modo concorrente;
4. il numero degli stati del sistema totale cresce esponenzialmente col numero dei sottosistemi.

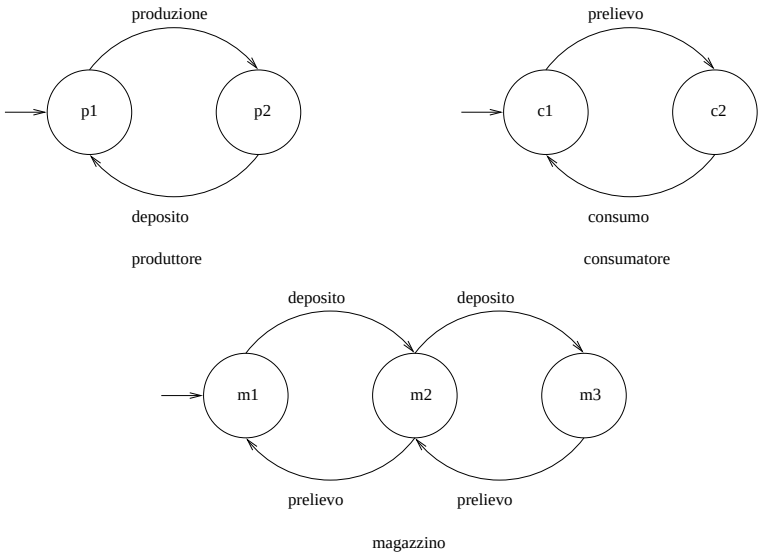

Figura 3.3: Esempio (1)

Mentre i punti 1 e 4 dimostrano la poca scalabilità degli ASF, il punto 3 ne mette in evidenza un'altra caratteristica: non esprimono la concorrenza dei sottosistemi, per cui si prestano solo alla specifica di sistemi sequenziali.

Il problema dell'aumento della complessità quando si compongono sottosistemi dipende dal fatto che, nel modello di ASF qui presentato, lo stato del sistema è *globale*, in quanto in un dato istante l'intero sistema viene modellato da un unico stato, e *atomico*, in quanto lo stato non contiene altra informazione che la propria identità e le funzioni di transizione e di uscita.

Un modo per rendere meno complesse le specifiche di sistemi per mezzo di ASF consiste nell'estendere il concetto di "stato" associandovi delle strutture dati. Nel nostro esempio, i tre stati dell'ASF che rappresenta il magazzino potrebbero essere sostituiti da un unico stato a cui è associata una variabile il cui valore è il numero di elementi immagazzinati. Le uscite associate alle operazioni di prelievo e di deposito sarebbero azioni di decremento e, rispettivamente, incremento della variabile (Fig. 3.5). Il prelievo e il deposito, a loro volta, sarebbero condizionati dal valore della variabile, cosa che comporta l'estensione del concetto di "transizione", a cui si devono aggiungere delle condizioni (*guardie*) che devono essere soddisfatte affinché una transizione possa aver luogo.

Un'altra estensione degli ASF consiste nell'introduzione di stati compositi, cioè descritti, a loro volta, da macchine a stati. In questo modo un sistema complesso può essere descritto ad alto livello da un automa con po-

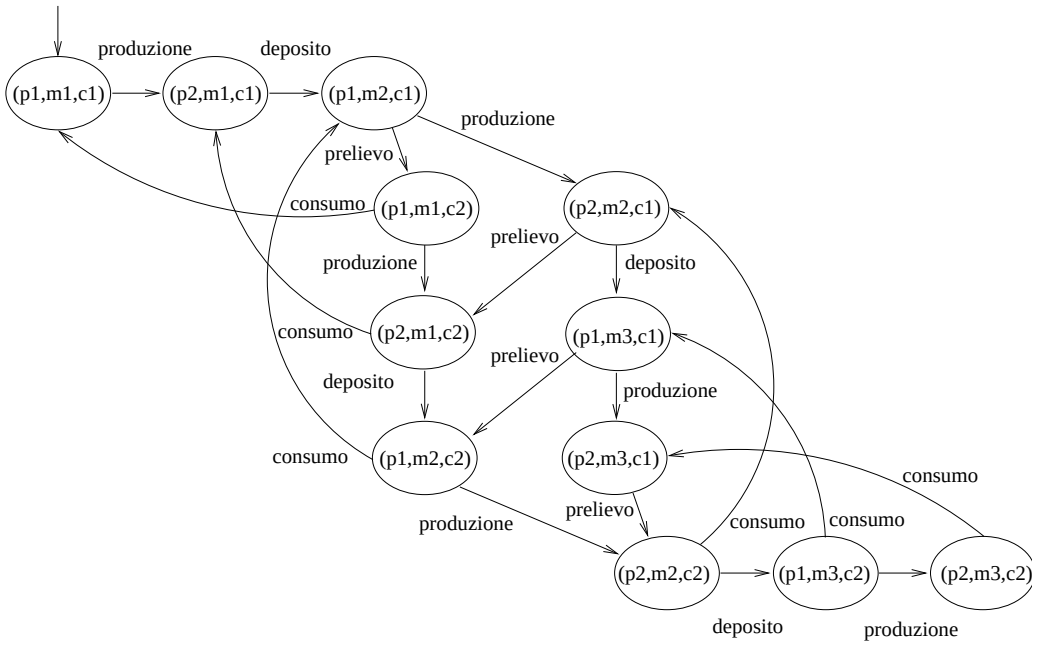

Figura 3.4: Esempio (2)

chi stati, ciascuno di quali può essere decomposto in sottostati quando serve una specifica piú dettagliata. Questo metodo è alla base degli *Statecharts*, un formalismo che esamineremo nel capitolo relativo ai metodi orientati agli oggetti.

Infine, nelle *reti di Petri* lo stato del sistema viene modellato in modo diverso, tale che la specifica renda esplicito e visibile il fatto che il sistema totale è composto da sottosistemi. Questo permette di descrivere sistemi concorrenti.

3.5 Logica

La logica serve a formalizzare il ragionamento, e in particolare a decidere in modo rigoroso (e quindi potenzialmente automatizzabile) se certe affermazioni sono vere e se dalla verità di certe affermazioni si può dedurre la verità di altre affermazioni. È evidente che la logica è fondamentale per qualsiasi forma di ragionamento scientifico, anche quando viene condotto per mezzo del linguaggio naturale, e in qualsiasi campo di applicazione, anche al di fuori delle discipline strettamente scientifiche e tecnologiche. In particolare, la

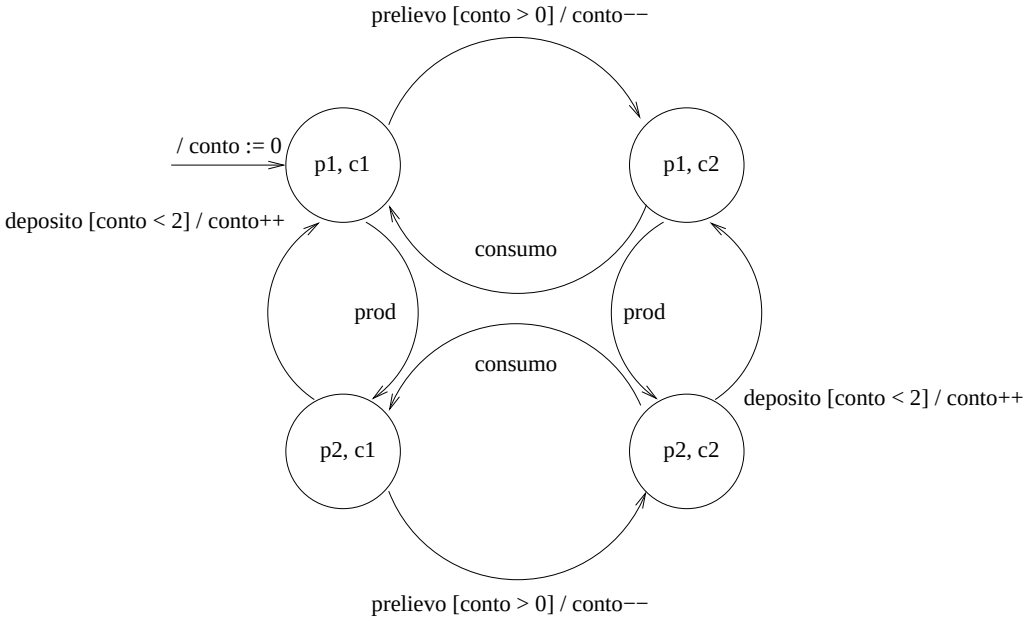

Figura 3.5: Esempio (3)

logica moderna è stata sviluppata per servire da fondamento alle discipline matematiche, fra cui rientra gran parte della scienza dell'informazione.

Al di là del suo carattere fondamentale, la logica può essere usata come linguaggio di specifica. Nell'ingegneria del software, quindi, una logica serve da linguaggio di specifica, quando si descrive un sistema in termini delle sue proprietà. Un sistema specificato per mezzo della logica può essere analizzato rigorosamente, e la specifica stessa può essere trasformata per ottenere descrizioni equivalenti ma piú vicine all'implementazione. Usando opportuni linguaggi (linguaggi di *programmazione logica*, come, per esempio, il Prolog), una specifica logica può essere eseguibile, e quindi fornire un prototipo del sistema specificato.

La logica è ovviamente importante per l'ingegneria del software anche perché, come accennato prima, è alla base di tutti i metodi formali usati nell'informatica. Inoltre la logica, come linguaggio di specifica, si può integrare con altri linguaggi; per esempio, si possono usare delle espressioni logiche come annotazioni formali per chiarire aspetti del sistema lasciati indeterminati da descrizioni informali.

Esistono diversi tipi di logica, come la logica proposizionale e la logica del primo ordine che vedremo fra poco, ognuno dei quali quali si presta a determinati scopi e campi di applicazione. Ciascuno di questi tipi di logica

permette di definire dei *sistemi formali* (o *teorie* formali), ognuno dei quali si basa su un *linguaggio* per mezzo del quale si possono scrivere formule che rappresentano le affermazioni che ci interessano. Un linguaggio viene descritto dalla propria *sintassi*. Il significato che attribuiamo alle formule è dato dalla *semantica* del linguaggio. La semantica associa i simboli del linguaggio alle entità di cui vogliamo parlare; queste entità costituiscono il *dominio* o *universo di discorso* del linguaggio.

Dato un linguaggio e la sua semantica, un insieme di *regole di inferenza* permette di stabilire se una formula può essere derivata da altre formule, ovvero se una formula è un *teorema* di un certo sistema formale, ovvero se esiste una *dimostrazione* di tale formula.

Le regole di inferenza si riferiscono alla sintassi del linguaggio: possiamo applicare una regola di inferenza ad un insieme di formule senza analizzare il loro significato, e in particolare senza sapere se sono vere o false. Un sistema formale è *corretto* (*sound*) se tutte le formule ottenibili dalle regole d'inferenza sono vere rispetto alla semantica del linguaggio, e *completo* se tutte le formule vere rispetto alla semantica sono ottenibili per mezzo delle regole d'inferenza. Naturalmente la correttezza è un requisito indispensabile per un sistema formale.

Un sistema formale è *decidibile* se esiste un algoritmo che può decidere in un numero finito di passi se una formula è vera (la logica del primo ordine non è decidibile).

Un sistema formale è quindi un apparato di definizioni e regole che ci permette di ragionare formalmente su un qualche settore della conoscenza. Precisiamo che spesso un particolare sistema formale viene indicato come una *logica*. Questa sovrapposizione di termini non crea problemi, perché dal contesto si capisce se si parla *della logica* in generale (la scienza del ragionamento formale) o *di una logica* particolare (un determinato sistema formale).

Le definizioni che verranno date nel séguito sono tratte principalmente da [23], con varie semplificazioni e probabilmente qualche imprecisione.

3.5.1 Calcolo proposizionale

Il calcolo proposizionale è la logica piú semplice. Gli elementi fondamentali del suo linguaggio[2] (non ulteriormente scomponibili) sono le *proposizioni*, cioè delle affermazioni che possono essere vere o false. Il fatto che le proposizioni non siano scomponibili, cioè siano *atomiche*, significa che in una frase come "*il tempo è bello*", il linguaggio del calcolo proposizionale non ci permette di individuare il soggetto e il predicato, poiché questo linguaggio non contiene dei simboli che possano nominare oggetti, proprietà o azioni: in un linguaggio proposizionale possiamo nominare soltanto delle frasi intere. Questo comporta anche che non è possibile mettere in evidenza la struttura comune di certi insiemi di frasi, come, per esempio, "*Aldo è bravo*", "*Beppe è bravo*", e "*Carlo è bravo*". Pertanto, qualsiasi proposizione può essere rappresentata semplicemente da una lettera dell'alfabeto[3] , come T per "*il tempo è bello*, A per "*Aldo è bravo*", eccetera.

Le proposizioni vengono combinate per mezzo di alcuni operatori per formare frasi piú complesse: gli operatori (chiamati *connettivi*) sono simili alle congiunzioni del linguaggio naturale, da cui hanno ereditato i nomi. Ai connettivi sono associate delle regole (*tabelle* o *funzioni di verità*) che permettono di stabilire la verità delle frasi complesse a partire dalle proposizioni.

Infine, un calcolo proposizionale ha delle *regole d'inferenza* che permettono di derivare alcune frasi a partire da altre frasi. Le regole d'inferenza sono scelte in modo che le frasi derivate per mezzo di esse siano vere, se sono vere le frasi di partenza. Si fa in modo, cioè, che i sistemi formali di tipo proposizionale siano corretti.

Sintassi

Nel calcolo proposizionale un linguaggio è formato da:

- un insieme numerabile $\mathcal{P}$ di *simboli proposizionali* (A,B,C, ...);
- un insieme finito di *connettivi*; per esempio: $\neg$ (*negazione*), $\wedge$ (*congiunzione*), $\vee$ (*disgiunzione*), $\Rightarrow$ (*implicazione*[4]), $\Leftrightarrow$ (*equivalenza* o *coimpli-*

[2]Piú precisamente, dei linguaggi di tipo proposizionale. Per semplicità, diremo genericamente "*il linguaggio* proposizionale".

[3]O da qualsiasi simbolo o sequenza di caratteri usabile come nome, p.es. '*valvola-chiusa*' nella Sez. 3.2.1.

[4]L'operazione logica associata a questo connettivo si chiama anche *implicazione materiale*, per distinguerla dall'*implicazione logica* che verrà introdotta piú oltre. Ricordiamo anche che a volte viene usato il simbolo $\rightarrow$ per l'implicazione materiale e il simbolo $\Rightarrow$ per l'implicazione logica.

cazione);
- un insieme di *simboli ausiliari* (parentesi e simili);
- un insieme $\mathcal{W}$ di *formule* (dette anche *formule ben formate*, *well-formed formulas*, o *wff*), definito dalle seguenti regole:
  1. un simbolo proposizionale è una formula;
  2. se $\mathcal{A}$ e $\mathcal{B}$ sono formule, allora sono formule anche $(\neg\mathcal{A})$, $(\mathcal{A} \wedge \mathcal{B})$, $(\mathcal{A} \vee \mathcal{B})$, ...
  3. solo le espressioni costruite secondo le due regole precedenti sono formule.

Alle regole sintattiche si aggiungono di solito delle regole ulteriori che permettono di semplificare la scrittura delle formule indicando le priorità dei connettivi, in modo simile a ciò che avviene nella notazione matematica. I connettivi vengono applicati in quest'ordine:

$$\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$$

per cui, ad esempio, la formula

$$A \Leftrightarrow \neg B \vee C \Rightarrow A$$

equivale a

$$A \Leftrightarrow (((\neg B) \vee C) \Rightarrow A)$$

Vediamo quindi che una formula del calcolo proposizionale, analogamente a un'espressione aritmetica, ha una struttura gerarchica: si parte da simboli elementari (i simboli proposizionali) che vengono combinati con operatori unari o binari ottenendo formule che si possono a loro volta combinare, e cosí via.

Semantica

La semantica di un calcolo proposizionale stabilisce le regole che associano un *valore di verità* a ciascuna formula. Il calcolo di questo valore avviene con un metodo di *ricorsione strutturale*: una formula complessa viene scomposta nelle sue formule componenti, fino ai simboli proposizionali; si assegnano i rispettivi valori di verità a ciascuno di essi (per mezzo della *funzione di valutazione*, vedi oltre), sulla base di questi valori si calcolano i valori delle formule di cui fanno parte i rispettivi simboli (per mezzo delle *funzioni di verità*, vedi oltre), e cosí via fino ad ottenere il valore della formula complessiva.

La semantica è quindi data da:

- l'insieme *booleano* $\mathbb{B} = \{\mathbf{T}, \mathbf{F}\}$, contenente i valori *vero* (**T**, *true*) e *falso* (**F**, *false*).
- una *funzione di valutazione* $v : \mathcal{P} \to \mathbb{B}$;
- le *funzioni di verità* di ciascun connettivo

$$\begin{array}{rcl} H_\neg & : & \mathbb{B} \to \mathbb{B} \\ H_\wedge & : & \mathbb{B}^2 \to \mathbb{B} \\ & \ldots & \end{array}$$

- una *funzione di interpretazione* $S_v : \mathcal{W} \to \mathbb{B}$ cosí definita:

$$\begin{array}{rcl} S_v(A) & = & v(A) \\ S_v(\neg\mathcal{A}) & = & H_\neg(S_v(\mathcal{A})) \\ S_v(\mathcal{A} \wedge \mathcal{B}) & = & H_\wedge(S_v(\mathcal{A}), S_v(\mathcal{B})) \\ & \ldots & \end{array}$$

  dove $A \in \mathcal{P}$, $\mathcal{A} \in \mathcal{W}$, $\mathcal{B} \in \mathcal{W}$.

La funzione di valutazione assegna un valore di verità a ciascun simbolo proposizionale. Osserviamo che questa funzione è arbitraria, nel senso che viene scelta da chi si vuole servire di un linguaggio logico per rappresentare un certo dominio, in modo da riflettere ciò che si considera vero in tale dominio.

Per esempio, consideriamo le proposizioni A (*Aldo è bravo*), S (*Aldo passa l'esame di Ingegneria del Software*) e T (*il tempo è bello*). A ciascuna proposizione si possono assegnare valori di verità scelti con i criteri ritenuti piú adatti a rappresentare la situazione, per esempio la valutazione di A può essere fatta in base a un giudizio soggettivo sulle capacità di Aldo, la valutazione di S e T può essere fatta in base all'osservazione sperimentale o anche assegnata arbitrariamente, considerando una situazione ipotetica.

Le funzioni di verità dànno il valore di verità restituito dall'operatore logico rappresentato da ciascun connettivo, in funzione dei valori di verità delle formule a cui viene applicato l'operatore. Di solito queste funzioni vengono espresse per mezzo di tabelle di verità come le seguenti:

x	y	$H_\neg(x)$	$H_\wedge(x,y)$	$H_\vee(x,y)$	$H_\Rightarrow(x,y)$	$H_\Leftrightarrow(x,y)$
F	**F**	**T**	**F**	**F**	**T**	**T**
F	**T**	**T**	**F**	**T**	**T**	**F**
T	**F**	**F**	**F**	**T**	**F**	**F**
T	**T**	**F**	**T**	**T**	**T**	**T**

Le funzioni di verità, a differenza della funzione di valutazione, sono parte integrante del linguaggio: poiché definiscono la semantica dei connettivi adottati dal linguaggio, non si possono modificare.

Un insieme di connettivi si dice *completo* se è sufficiente ad esprimere tutte le funzioni di verità possibili. Alcuni insiemi completi sono:

$$\{\neg, \wedge\}$$
$$\{\neg, \vee\}$$
$$\{\neg, \Rightarrow\}$$

La funzione d'interpretazione, infine, calcola il valore di verità di qualsiasi formula, in base alla funzione di valutazione, alle funzioni di verità, ed alla struttura della formula stessa.

Soddisfacibilità e validità

Una formula, in generale, può essere vera o falsa a seconda della funzione di valutazione, cioè del modo in cui vengono assegnati valori di verità ai simboli proposizionali. Esistono però delle formule il cui valore di verità *non dipende* dalla funzione di valutazione, cioè sono vere (o false) per qualsiasi scelta dei valori di verità dei simboli proposizionali. Ovviamente queste formule sono particolarmente utili perché sono di applicabilità piú generale. Nei paragrafi seguenti esprimeremo piú precisamente la dipendenza del valore di verità di una formula dalla funzione di valutazione.

Se, per una formula $\mathcal{F}$ ed una valutazione v, si ha che $S_v(\mathcal{F}) = \mathbf{T}$, si dice che v *soddisfa* $\mathcal{F}$, e si scrive $v \models \mathcal{F}$.

Osservazione. Il simbolo $\models$ non appartiene al linguaggio del calcolo proposizionale, poiché non serve a costruire delle formule, ma è solo un'abbreviazione della frase "soddisfa", che appartiene al metalinguaggio, cioè al linguaggio di cui ci serviamo per parlare del calcolo proposizionale.

Una formula si dice *soddisfacibile* o *consistente* se esiste almeno una valutazione che la soddisfa. Per esempio:

$$A \Rightarrow \neg A \quad \text{per } v(A) = \mathbf{F}$$

Una formula si dice *insoddisfacibile* o *inconsistente* se non esiste alcuna valutazione che la soddisfi. Si dice anche che la formula è una *contraddizione*.

Per esempio:

$$A \wedge \neg A$$

Una formula soddisfatta da tutte le valutazioni è una *tautologia*, ovvero è *valida*. La verità di una tautologia non dipende quindi dalla verità delle singole proposizioni che vi appaiono, ma unicamente dalla struttura della formula. Esempi di tautologie sono:

$$\begin{aligned} A \vee \neg A \\ A \Rightarrow A \\ \neg\neg A \Rightarrow A \\ \neg(A \wedge \neg A) \\ (A \wedge B) \Rightarrow A \\ A \Rightarrow (A \vee B) \end{aligned}$$

Se $\mathcal{A} \Rightarrow \mathcal{B}$ (ove $\mathcal{A}$ e $\mathcal{B}$ sono formule) è una tautologia, si dice che $\mathcal{A}$ *implica logicamente* $\mathcal{B}$, ovvero che $\mathcal{B}$ è *conseguenza logica* di $\mathcal{A}$.

Conviene osservare la differenza fra *implicazione* (o *implicazione materiale*) e *implicazione logica*. Infatti, la formula $\mathcal{A} \Rightarrow \mathcal{B}$ si legge "$\mathcal{A}$ implica $\mathcal{B}$", e questa frase significa che il valore di verità della formula viene calcolato secondo la funzione di verità dell'operatore di implicazione a partire dai valori di $\mathcal{A}$ e $\mathcal{B}$, che a loro volta dipendono, in generale, dalla struttura di queste ultime formule e dalla particolare valutazione. La frase "$\mathcal{A}$ implica logicamente $\mathcal{B}$", invece, significa che la formula $\mathcal{A} \Rightarrow \mathcal{B}$ è vera per ogni valutazione, ed è quindi una affermazione piú forte.

Analogamente, se $\mathcal{A} \Leftrightarrow \mathcal{B}$ è una tautologia si dice che $\mathcal{A}$ e $\mathcal{B}$ sono *logicamente equivalenti*.

3.5.2 Teorie formali

Nel calcolo proposizionale è sempre possibile accertarsi se una data formula è valida oppure no: basta calcolare il suo valore di verità per tutte le valutazioni possibili, che per ciascuna formula costituiscono un insieme finito, essendo finiti l'insieme dei simboli proposizionali nella formula, delle loro occorrenze, e dei loro possibili valori di verità. Questo metodo diretto di verifica della validità fa riferimento alla semantica del linguaggio usato. I metodi basati sulla

semantica del linguaggio sono indicati comunemente col termine *model checking*, e sono impiegati per molti linguaggi formali di specifica nell'ingegneria del software.

Se il numero di simboli proposizionali in una formula è grande, la verifica diretta può essere impraticabile. Inoltre la verifica diretta è generalmente impossibile nelle logiche piú potenti del calcolo proposizionale (come la logica del primo ordine, che vedremo fra poco), la cui semantica può comprendere insiemi infiniti di valori. Si pone quindi il problema di *dedurre* la verità di una formula non attraverso il calcolo diretto del suo valore di verità, ma attraverso certe relazioni, basate sulla sua sintassi, con altre formule. Informalmente, il meccanismo di deduzione si può descrivere in questo modo:

1. si sceglie un insieme di formule che consideriamo valide *a priori*, senza necessità di dimostrazione, oppure che verifichiamo direttamente per mezzo della semantica;
2. si definiscono delle regole (dette *d'inferenza*) che, date alcune formule con una certa struttura, considerate vere, permettono di scrivere nuove formule;
3. a partire dalle formule introdotte al punto 1, si costruiscono delle catene di formule applicando ripetutamente le regole d'inferenza;
4. ogni formula appartenente a una delle catene cosí costruite si considera dimostrata sulla base delle formule che la precedono nella catena.

Regole di inferenza

Una *regola di inferenza* è una relazione fra formule; per ogni n-upla di formule che soddisfa una regola di inferenza R, una determinata formula della n-upla viene chiamata *conseguenza diretta* delle altre formule della n-upla *per effetto di* R. Per esempio, una regola di inferenza potrebbe essere l'insieme delle triple aventi la forma $\langle \mathcal{B}, \mathcal{A}, \mathcal{A} \Rightarrow \mathcal{B} \rangle$. Questa regola si scrive generalmente in questa forma:

$$\frac{\mathcal{A} \qquad \mathcal{A} \Rightarrow \mathcal{B}}{\mathcal{B}}$$

che si legge "$\mathcal{B}$ *è conseguenza diretta di* $\mathcal{A}$ *e di* $\mathcal{A} \Rightarrow \mathcal{B}$". Meno sinteticamente: "Se $\mathcal{A}$ è vera ed $\mathcal{A}$ implica $\mathcal{B}$, allora possiamo dedurre che $\mathcal{B}$ è vera".

Dimostrazioni e teoremi

Possiamo ora introdurre il concetto di *teoria* (o *sistema*) *formale*, nell'àmbito del quale si definiscono i concetti di *dimostrazione* e di *teorema*.

Una teoria formale è data da
1. un linguaggio $\mathcal{L}$;
2. un insieme $\mathbf{A}$ (eventualmente infinito) di formule di $\mathcal{L}$ chiamate *assiomi*;
3. un insieme finito di regole di inferenza fra formule di $\mathcal{L}$.

Se Γ è un insieme di formule, dette *ipotesi* o *premesse*, $\mathcal{F}$ è una formula da dimostrare, e Δ è una sequenza di formule, allora si dice che Δ è una *dimostrazione* (o *deduzione*) *di* $\mathcal{F}$ *da* Γ se l'ultima formula di Δ è $\mathcal{F}$ e ciascun'altra: o (i) è un assioma, o (ii) è una premessa, o (iii) è conseguenza diretta di formule che la precedono nella sequenza Δ.

Si dice quindi che $\mathcal{F}$ *segue da* Γ, o è *conseguenza* di Γ, e si scrive

$$\Gamma \vdash \mathcal{F}$$

Se l'insieme Γ delle premesse è vuoto, allora la sequenza Δ è una **dimostrazione di** $\mathcal{F}$, ed $\mathcal{F}$ è un *teorema*, e si scrive

$$\vdash \mathcal{F}$$

Quindi, in una teoria formale, una dimostrazione è una sequenza di formule tali che ciascuna di esse o è un assioma o è conseguenza diretta di alcune formule precedenti, e un teorema è una formula che si può dimostrare ricorrendo solo agli assiomi, senza ipotesi aggiuntive.

Osservazione. Notare la differenza fra *dimostrazione di* $\mathcal{F}$ e *dimostrazione di* $\mathcal{F}$ *da* Γ.

Una teoria formale per il calcolo proposizionale

Una semplice teoria formale per il calcolo proposizionale può essere definita come segue [23]:

- Il linguaggio $\mathcal{L}$ è costituito dalle formule ottenute a partire dai simboli proposizionali, dai connettivi $\neg$ e $\Rightarrow$, e dalle parentesi.
- Gli assiomi sono tutte le espressioni date dai seguenti *schemi*:

$$\mathcal{A} \Rightarrow (\mathcal{B} \Rightarrow \mathcal{A}) \tag{3.1}$$

$$(\mathcal{A} \Rightarrow (\mathcal{B} \Rightarrow \mathcal{C})) \Rightarrow ((\mathcal{A} \Rightarrow \mathcal{B}) \Rightarrow (\mathcal{A} \Rightarrow \mathcal{C})) \tag{3.2}$$

$$(\neg\mathcal{B} \Rightarrow \neg\mathcal{A}) \Rightarrow ((\neg\mathcal{B} \Rightarrow \mathcal{A}) \Rightarrow \mathcal{B}) \tag{3.3}$$

  Questa teoria formale ha quindi un insieme infinito di assiomi; osserviamo anche che qualsiasi assioma ottenuto da questi schemi è una tautologia.

- L'unica regola d'inferenza è il *modus ponens* (MP): una formula $\mathcal{B}$ è conseguenza diretta di $\mathcal{A}$ e $\mathcal{A} \Rightarrow \mathcal{B}$. Si scrive anche

$$\frac{\mathcal{A} \qquad \mathcal{A} \Rightarrow \mathcal{B}}{\mathcal{B}}$$

Dimostriamo, per esempio, che $F \Rightarrow F$, per ogni formula F[5]:

$$\begin{array}{rcll}
(F \Rightarrow ((F \Rightarrow F) \Rightarrow F)) & \Rightarrow & ((F \Rightarrow (F \Rightarrow F)) \Rightarrow (F \Rightarrow F)) & \\
 & & \text{da 3.2, } \mathcal{A} = \mathcal{C} = F,\ \mathcal{B} = F \Rightarrow F & (3.4) \\
F \Rightarrow ((F \Rightarrow F) \Rightarrow F) & & \text{da 3.1, } \mathcal{A} = F,\ \mathcal{B} = F \Rightarrow F & (3.5) \\
(F \Rightarrow (F \Rightarrow F)) \Rightarrow (F \Rightarrow F) & & \text{da 3.4 e 3.5 per MP,} & \\
 & & \mathcal{A} = (F \Rightarrow ((F \Rightarrow F) \Rightarrow F)), & \\
 & & \mathcal{B} = (F \Rightarrow (F \Rightarrow F)) \Rightarrow (F \Rightarrow F) & (3.6) \\
F \Rightarrow (F \Rightarrow F) & & \text{da 3.1, } \mathcal{A} = F,\ \mathcal{B} = F & (3.7) \\
F \Rightarrow F & & \text{da 3.6 e 3.7 per MP,} & \\
 & & \mathcal{A} = F \Rightarrow (F \Rightarrow F), & \\
 & & \mathcal{B} = F \Rightarrow F & (3.8)
\end{array}$$

Un utile risultato dimostrabile su questa teoria formale è il *Teorema della deduzione*: *se* Γ *è un insieme di wff,* $\mathcal{A}$ *e* $\mathcal{B}$ *sono wff, e* $\Gamma \cup \{\mathcal{A}\} \vdash \mathcal{B}$*, allora* $\Gamma \vdash \mathcal{A} \Rightarrow \mathcal{B}$. In particolare, se $\{\mathcal{A}\} \vdash \mathcal{B}$, allora $\vdash \mathcal{A} \Rightarrow \mathcal{B}$. Cioè, si può affermare che $\mathcal{A}$ implica logicamente[6] $\mathcal{B}$ se $\mathcal{B}$ è dimostrabile da $\mathcal{A}$, coerentemente col comune modo di dimostrare i teoremi in matematica.

Osservazione. Il Teorema della deduzione è, propriamente, un metateorema, perché afferma una proprietà di un sistema formale, mentre un teorema è una formula dimostrabile nell'àmbito del sistema stesso.

Infine, si può dimostrare che questa teoria formale è corretta e completa, cioè che ogni teorema di questa teoria è una tautologia, e viceversa.

3.5.3 Logica del primo ordine

La logica del primo ordine (anche *FOL*, *first order logic*) permette di formare delle frasi in cui è possibile riferirsi a entità individuali (*oggetti* o *individui*),

[5] Scriviamo F invece di $\mathcal{F}$ per evidenziare graficamente la formula

[6] Se $\mathcal{A} \Rightarrow \mathcal{B}$ è un teorema, è una tautologia.

sia specificamente, per mezzo di simboli (*costanti* e *funzioni*) che denotano particolari entità, sia genericamente, per mezzo di simboli (*variabili*) che si riferiscono a individui non specificati (analogamente ai pronomi indefiniti nel linguaggio naturale). Le frasi del linguaggio, inoltre, possono essere *quantificate* rispetto alle variabili che vi compaiono, cioè si può indicare se determinate proprietà valgono per tutti i valori di tali variabili o solo per alcuni.

Le frasi piú semplici che possiamo esprimere nel linguaggio della logica del primo ordine sono del tipo "gli oggetti a, b, c, ... sono legati dalla relazione p" (o, come caso particolare, "l'oggetto a gode della proprietà p"). Queste formule vengono combinate per costruire frasi piú complesse, usando i connettivi ed i quantificatori.

Per esempio, consideriamo queste frasi:

1. "*Aldo è bravo*", "*Beppe è bravo*", "*Carlo è bravo*".
2. "*Aldo passa Ingegneria del Software*", "*Beppe passa Sistemi Operativi e Reti*", "*Carlo passa Epistemologia Generale*".

Le frasi del gruppo 1 dicono che certi individui godono di una certa proprietà, ovvero che Aldo, Beppe e Carlo appartengono all'insieme degli studenti bravi. Le frasi del gruppo 2 dicono che esiste una certa relazione fra certi individui e certi altri individui (non si fa distinzione fra entità animate e inanimate), ovvero che le coppie (Aldo, Ingegneria del Software), (Beppe, Sistemi Operativi e Reti), e (Carlo, Epistemologia Generale) appartengono all'insieme di coppie ordinate tali che l'individuo nominato dal primo elemento della coppia abbia superato l'esame nominato dal secondo elemento.

Sia le frasi del primo gruppo che quelle del secondo affermano che certi *predicati*, cioè proprietà o relazioni, valgono per certe entità individuali, e si possono scrivere sinteticamente come:

$$b(A),\ b(B),\ b(C),\ p(A, I),\ p(B, S),\ p(C, E)$$

dove il *simbolo di predicato* b sta per "è bravo", p per "passa l'esame di", il *simbolo di costante* A per "Aldo", e cosí via.

Queste frasi sono istanze particolari delle formule $b(x)$ e $p(x, y)$, dove le variabili x e y sono dei simboli segnaposto che devono essere sostituiti da nomi di individui per ottenere delle frasi cui si possa assegnare un valore di verità. È bene sottolineare che la formula $b(x)$ non significa "*qualcuno*

è bravo": la formula non significa niente, non è né vera né falsa, finché il simbolo x non viene sostituito da un'espressione che denoti un particolare individuo, per esempio "Aldo" o anche "il padre di Aldo". Dopo questa sostituzione, la formula può essere vera o falsa, ma ha comunque un valore definito.

Se vogliamo esprimere nella logica del primo ordine la frase "*qualcuno è bravo*", dobbiamo usare un nuovo tipo di simbolo, il quantificatore esistenziale: $\exists x b(x)$, cioè "esiste qualcuno che è bravo". Questa formula ha significato cosí com'è, perché afferma una proprietà dell'insieme degli studenti bravi (la proprietà di non essere vuoto).

Una formula come $b(x)$, dove, cioè, appaiono variabili non quantificate, si dice *aperta*, mentre una formula come $\exists x b(x)$, con le variabili quantificate, si dice *chiusa*.

Sintassi

Quanto sopra viene espresso piú formalmente dicendo che un linguaggio del primo ordine è costituito da:

- un insieme numerabile $\mathcal{V}$ di *variabili* (x, y, z, ...);
- un insieme numerabile $\mathcal{F}$ di *simboli di funzione* (f, g, h, ...); a ciascun simbolo di funzione è associato il numero di argomenti (*arietà*) della funzione: $\mathcal{F} = \mathcal{F}_0 \cup \mathcal{F}_1 \cup \mathcal{F}_2 \cup \ldots$, dove $\mathcal{F}_n$ è l'insieme dei simboli n-ari di funzione; le funzioni di arietà zero sono chiamate *costanti* (a, b, c, ...);
- un insieme $\mathcal{T}$ di *termini* (t_1, t_2, t_3, ...), definito dalle seguenti regole:
  1. una variabile è un termine;
  2. se f è un simbolo n-ario di funzione e $t_1, \ldots, t_n$ sono termini, allora $f(t_1, \ldots, t_n)$ è un termine; in particolare, un simbolo c di arietà nulla è un termine (*costante*);
  3. solo le espressioni costruite secondo le due regole precedenti sono termini.
- un insieme numerabile $\mathcal{P}$ di *simboli di predicato* (p, q, r, ...); a ciascun simbolo di predicato è associato il numero di argomenti (*arietà*) del predicato: $\mathcal{P} = \mathcal{P}_0 \cup \mathcal{P}_1 \cup \mathcal{P}_2 \cup \ldots$, dove $\mathcal{P}_n$ è l'insieme dei simboli n-ari di predicato; i predicati di arietà zero corrispondono ai simboli proposizionali del calcolo proposizionale;

- un insieme finito di *connettivi*; per esempio:

$$\neg, \wedge, \vee, \Rightarrow, \ldots$$

- un insieme finito di *quantificatori*; per esempio:

$$\forall, \exists$$

- un insieme di *simboli ausiliari* (parentesi e simili);
- un insieme $\mathcal{W}$ di *formule*, definito dalle seguenti regole:
  1. se p è un simbolo n-ario di predicato e $t_1, \ldots, t_n$ sono termini, allora $p(t_1, \ldots, t_n)$ è una formula (detta *atomica*, in questo caso);
  2. se $\mathcal{A}$ e $\mathcal{B}$ sono formule, allora sono formule anche $(\neg\mathcal{A})$, $(\mathcal{A} \wedge \mathcal{B})$, $(\mathcal{A} \vee \mathcal{B})$, ...
  3. se $\mathcal{A}$ è una formula e x è una variabile, allora sono formule anche $(\forall x\mathcal{A})$ e $(\exists x\mathcal{A})$. Le formule $(\forall x\mathcal{A})$ e $(\exists x\mathcal{A})$ sono dette *formule quantificate* ed $\mathcal{A}$ è il *campo* del rispettivo quantificatore.
  4. solo le espressioni costruite secondo le regole precedenti sono formule.

Le priorità nell'ordine di applicazione dei connettivi è la stessa vista per il calcolo proposizionale, salvo che la priorità dei quantificatori è intermedia fra quella di $\vee$ e quella di $\Rightarrow$.

Semantica

La semantica di una logica del primo ordine è data da:

- l'insieme $\mathbb{B} = \{\mathbf{T}, \mathbf{F}\}$;
- le *funzioni di verità* di ciascun connettivo;
- un insieme non vuoto $\mathcal{D}$, detto *dominio* dell'interpretazione;
- una *funzione di interpretazione delle funzioni* $\Phi : \mathcal{F} \to \mathcal{F}_\mathcal{D}$, dove $\mathcal{F}_\mathcal{D}$ è l'insieme delle funzioni su $\mathcal{D}$. Φ assegna a ciascun simbolo n-ario di funzione una funzione $\mathcal{D}^n \to \mathcal{D}$.
- una *funzione di interpretazione dei predicati* $\Pi : \mathcal{P} \to \mathcal{R}_\mathcal{D}$, dove $\mathcal{R}_\mathcal{D}$ è l'insieme delle relazioni su $\mathcal{D}$. Π assegna a ciascun simbolo n-ario di predicato una funzione $\mathcal{D}^n \to \mathbb{B}$.
- un *assegnamento di variabili* $\xi : \mathcal{V} \to \mathcal{D}$;

- un *assegnamento di termini* $\Xi : \mathcal{T} \to \mathcal{D}$ cosí definito:

$$\begin{aligned} \Xi(x) &= \xi(x) \\ \Xi(f(t_1, \ldots, t_n)) &= \Phi(f)(\ \Xi(t_1), \ldots, \Xi(t_n)\ ) \end{aligned}$$

  dove $x \in \mathcal{V}$, $t_i \in \mathcal{T}$, $f \in \mathcal{F}$;
- una *funzione di interpretazione* $S_{I,\xi} : \mathcal{W} \to \mathbb{B}$ cosí definita:

$$\begin{aligned} S_{I,\xi}(p(t_1, \ldots, t_n)) &= \Pi(p)(\ \Xi(t_1), \ldots, \Xi(t_n)\ ) \\ S_{I,\xi}(\neg\mathcal{A}) &= H_\neg(S_{I,\xi}(\mathcal{A})) \\ S_{I,\xi}(\mathcal{A} \wedge \mathcal{B}) &= H_\wedge(S_{I,\xi}(\mathcal{A}), S_{I,\xi}(\mathcal{B})) \\ &\ldots \\ S_{I,\xi}(\exists x\mathcal{A}) &= \mathbf{T} \\ &\quad \text{se e solo se esiste un } d \in \mathcal{D} \text{ tale che} \\ &\quad [S_{I,\xi}]_{x/d}(\mathcal{A}) = \mathbf{T} \\ S_{I,\xi}(\forall x\mathcal{A}) &= \mathbf{T} \\ &\quad \text{se e solo se per ogni } d \in \mathcal{D} \text{ si ha} \\ &\quad [S_{I,\xi}]_{x/d}(\mathcal{A}) = \mathbf{T} \end{aligned}$$

  dove $p \in \mathcal{P}$, $t_i \in \mathcal{T}$, $\mathcal{A}, \mathcal{B} \in \mathcal{W}$, $I = (\mathcal{D}, \Phi, \Pi)$ è detta *interpretazione* del linguaggio, $x \in \mathcal{V}$, e $[S_{I,\xi}]_{x/d}$ è la funzione di interpretazione uguale a $S_{I,\xi}$ eccetto che assegna alla variabile x il valore d.

Il dominio è l'insieme degli oggetti di cui vogliamo parlare. Per esempio, se volessimo parlare dell'aritmetica, il dominio sarebbe l'insieme dei numeri interi.

La funzione d'interpretazione delle funzioni stabilisce il significato dei simboli che usiamo nel nostro linguaggio per denotare le funzioni. Nell'esempio dell'aritmetica, una funzione di interpretazione delle funzioni potrebbe associare la funzione aritmetica *somma* ($somma \in \mathcal{F}_\mathcal{D}$) al simbolo di funzione f ($f \in \mathcal{F}$), cioè $\Phi(f) = somma$. In particolare, Φ stabilisce il significato delle costanti, per esempio possiamo associare al simbolo a il numero *zero*, al simbolo b il numero *uno*, e cosí via. Di solito, quando si usa la logica per parlare di un argomento ove esiste una notazione tradizionale, si cerca di usare quella notazione, a meno che non si voglia sottolineare la distinzione fra linguaggio e dominio (come stiamo facendo qui). Quindi in genere è possibile usare il simbolo '+' per la somma, il simbolo '1' per il numero *uno*, eccetera.

La funzione d'interpretazione dei predicati stabilisce il significato dei simboli che denotano le relazioni. Per esempio, una funzione di interpretazione dei predicati può associare la relazione *minore o uguale* al simbolo di predicato p.

simbolo	funzione	interpretazione	simbolo comune
f	Φ	somma	$+$
b	Φ	uno	1
x	ξ	tre	3
p	Π	minore o uguale	$\leq$

Tabella 3.2: Esempio di interpretazione.

L'assegnamento di variabili stabilisce il significato delle variabili. Per esempio, possiamo assegnare il valore *tre* alla variabile x.

L'assegnamento di termini stabilisce il significato dei termini. Per esempio, dato il termine $f(b, x)$, e supponendo che $\Phi(f) = somma$, $\Phi(b) = uno$, $\xi(x) = tre$, allora $\Xi(f(b, x)) = quattro$.

Infine, la funzione d'interpretazione (delle formule) stabilisce il significato delle formule di qualsivoglia complessità. Come nel calcolo proposizionale, la funzione di interpretazione ha una definizione ricorsiva e si applica analizzando ciascuna formula nelle sue componenti. Le formule piú semplici (formule atomiche) sono costituite da simboli di predicato applicati ad n-uple di termini, quindi il valore di verità di una formula atomica (fornito dalla funzione Π) dipende dall'assegnamento dei termini.

La definizione della funzione di interpretazione per le formule quantificate rispecchia la comune nozione di quantificazione esistenziale ed universale, a dispetto della notazione un po' oscura qui adottata. L'espressione $[S_{I,\xi}]_{x/d}$, come abbiamo visto, è definita come "la funzione di interpretazione uguale a $S_{I,\xi}$ eccetto che assegna alla variabile x il valore d". Questo significa che, per vedere se una formula quantificata sulla variabile x è vera, non ci interessa *il* valore attribuito a x dal particolare assegnamento ξ , ma *l'insieme* dei possibili valori di x, indipendentemente dall'assegnamento: per la quantificazione esistenziale vediamo se almeno uno dei valori possibili soddisfa la formula compresa nel campo del quantificatore, per la quantificazione universale vediamo se tutti i valori possibili la soddisfano. In ambedue i casi, per le altre variabili si considerano i valori assegnati da ξ.

Come esempio di interpretazione, data la formula $p(f(b, x), x) \wedge p(b, x)$, si ha $S_{I,\xi}(p(f(b, x), x) \wedge p(b, x)) = \mathbf{F}$, se l'interpretazione I e l'assegnamento di variabili ξ sono compatibili con le interpretazioni ed assegnamenti visti sopra. Si verifica facilmente che I e ξ trasformano la formula logica considerata nell'espressione $1 + 3 \leq 3 \wedge 1 \leq 3$. La Tab. 3.2 riassume le interpretazioni dei vari simboli.

Soddisfacibilità e validità

Nel calcolo proposizionale l'interpretazione di una formula dipende solo dalla sua struttura e, in generale, dalla funzione di valutazione. Nella logica dei predicati le formule sono piú complesse e il loro valore di verità dipende, oltre che dalla struttura della formula, anche dal dominio di interpretazione, dalle funzioni d'interpretazione dei simboli di funzione e di predicato, e dalla funzione di assegnamento di variabili. Il dominio e le funzioni di interpretazione costituiscono, come abbiamo visto, l'interpretazione del linguaggio e ne definiscono l'aspetto strutturale, invariabile. Anche nella logica dei predicati ci interessa studiare la dipendenza del valore di verità delle formule dall'interpretazione del linguaggio e dall'assegnamento di variabili. Di séguito le definizioni relative.

Una formula $\mathcal{A}$ è *soddisfacibile in un'interpretazione* I se e solo se esiste un assegnamento di variabili ξ tale che $S_{I,\xi}(\mathcal{A}) = \mathbf{T}$. Si dice allora che l'interpretazione I *soddisfa* $\mathcal{A}$ con assegnamento di variabili ξ, e si scrive $I \overset{\xi}{\models} \mathcal{A}$.

Una formula $\mathcal{A}$ è *soddisfacibile* (tout-court) se e solo se esiste un'interpretazione I in cui $\mathcal{A}$ è soddisfacibile.

Una formula $\mathcal{A}$ è *valida in un'interpretazione* (o *vera in un'interpretazione*) I se e solo se $S_{I,\xi}(\mathcal{A}) = \mathbf{T}$ per ogni un assegnamento di variabili ξ. Si dice quindi che I è un *modello* di $\mathcal{A}$, e si scrive $I \models \mathcal{A}$.

Osserviamo che una formula *aperta* $\mathcal{A}$, cioè contenente variabili non quantificate, è valida in un'interpretazione I se e soltanto se è valida la sua *chiusura universale*, cioè la formula ottenuta da $\mathcal{A}$ quantificando universalmente le sue variabili libere.

Una formula $\mathcal{A}$ è (*logicamente*) *valida* se e solo se è valida per ogni interpretazione I, e si scrive $\models \mathcal{A}$.

I concetti di validità e soddisfacibilità si estendono a insiemi di formule: un insieme di formule è equivalente alla loro congiunzione.

Una formula $\mathcal{A}$ *implica logicamente* $\mathcal{B}$, ovvero $\mathcal{B}$ è *conseguenza logica* di $\mathcal{A}$ se e solo se $\mathcal{A} \Rightarrow \mathcal{B}$ è valida.

Una formula $\mathcal{A}$ *è logicamente equivalente* a $\mathcal{B}$, se e solo se $\mathcal{A} \Leftrightarrow \mathcal{B}$ è valida.

Una teoria formale per la FOL

Una semplice teoria formale per la FOL può essere definita come segue [23]:

- il linguaggio $\mathcal{L}$ è un linguaggio del primo ordine che usa i connettivi $\neg$ e $\Rightarrow$, il quantificatore $\forall$, e le parentesi[7];
- gli assiomi sono divisi in *assiomi logici* ed *assiomi propri* (o *non logici*); gli assiomi logici sono tutte le espressioni date dai seguenti *schemi*:

$$\mathcal{A} \Rightarrow (\mathcal{B} \Rightarrow \mathcal{A}) \tag{3.9}$$
$$(\mathcal{A} \Rightarrow (\mathcal{B} \Rightarrow \mathcal{C})) \Rightarrow ((\mathcal{A} \Rightarrow \mathcal{B}) \Rightarrow (\mathcal{A} \Rightarrow \mathcal{C})) \tag{3.10}$$
$$(\neg\mathcal{B} \Rightarrow \neg\mathcal{A}) \Rightarrow ((\neg\mathcal{B} \Rightarrow \mathcal{A}) \Rightarrow \mathcal{B}) \tag{3.11}$$
$$\forall x \mathcal{A}(x) \Rightarrow \mathcal{A}(t) \tag{3.12}$$
$$\forall x(\mathcal{A} \Rightarrow \mathcal{B}) \Rightarrow (\mathcal{A} \Rightarrow \forall x \mathcal{B}) \tag{3.13}$$

- le regole d'inferenza sono:

  modus ponens come nel calcolo proposizionale;
  generalizzazione

$$\frac{\mathcal{A}}{\forall x \mathcal{A}}$$

Negli assiomi, $\mathcal{A}(x)$ rappresenta una formula contenente la variabile x (ed eventualmente altre variabili), ed $\mathcal{A}(t)$ è la formula ottenuta da $\mathcal{A}(x)$ sostituendo x col termine t.

Allo schema di assioma 3.12 bisogna aggiungere la restrizione che t sia *libero per x in $\mathcal{A}$*, cioè che non contenga variabili che in $\mathcal{A}$ abbiano un quantificatore universale il cui campo d'azione includa occorrenze di x; questo vincolo impedisce che variabili libere nel termine t diventino quantificate quando t viene sostituito a x. Supponiamo, per esempio, di voler dimostrare la formula

$$\forall x(\neg\forall y(x = y)) \Rightarrow \neg(\forall y(y = y))$$

nel dominio dei numeri interi.

Questa formula significa che "*se per ogni x esiste un y diverso da x, allora non è vero che ogni y è uguale a se stesso*", ed è falsa, date le note proprietà della relazione di uguaglianza. Però la formula ha la stessa struttura dello

[7] La quantificazione esistenziale si può esprimere usando il quantificatore universale, se poniamo $\exists x \mathcal{A}$ equivalente a $\neg(\forall x(\neg\mathcal{A}))$.

schema 3.12, come si verifica sostituendo $\neg\forall y(x = y)$ ad $\mathcal{A}(x)$ e $\neg(\forall y(y = y))$ al posto di $\mathcal{A}(t)$. Ma lo schema non è applicabile, perché richiede che la metavariabile t venga sostituita dal termine y, che non è libero per x nella formula $\neg\forall y(x = y)$; infatti y è quantificata nella sottoformula $\neg\forall y(x = y)$ (che sostituisce $\mathcal{A}(x)$), e il campo d'azione del suo quantificatore comprende una occorrenza di x.

Allo schema di assioma 3.13 bisogna aggiungere la restrizione che $\mathcal{A}$ non contenga occorrenze libere di x. Per esempio, se $\mathcal{A}$ e $\mathcal{B}$ corrispondono tutte e due alla formula $p(x)$, dove $p(x)$ viene interpretato come "*x è pari*", l'applicazione dell'assioma 3.13 porterebbe alla formula $\forall x(p(x) \Rightarrow p(x)) \Rightarrow (p(x) \Rightarrow \forall x\ p(x))$, che è falsa: l'antecedente dell'implicazione principale $(\forall x(p(x) \Rightarrow p(x)))$ significa che "*per ogni x la parità implica la parità*", e il conseguente $((p(x) \Rightarrow \forall x\ p(x)))$ significa che "*la parità di un numero implica la parità di tutti i numeri*".

Gli assiomi propri sono specifici di ciascuna particolare teoria (p.es., teoria dei gruppi, dell'aritmetica, ...) e possono mancare. Un sistema formale privo di assiomi propri si dice *calcolo dei predicati*.

In ogni calcolo dei predicati, tutti i teoremi sono formule logicamente valide e viceversa (Teorema di completezza di Gödel).

Il sistema formale qui esposto ha un numero relativamente grande di schemi di assiomi e poche regole di inferenza: i sistemi di questo tipo vengono spesso detti *alla Hilbert*. Altri sistemi formali, come la *deduzione naturale* o il *calcolo dei sequenti* (v. oltre), hanno pochi assiomi e un maggior numero di regole d'inferenza.

3.5.4 Esempio di specifica e verifica formale

Consideriamo il problema dell'ordinamento di un vettore (esempio tratto da [14]). Vogliamo specificare la relazione fra un vettore arbitrario x di $N > 2$ elementi ed il vettore y ottenuto ordinando x in ordine crescente, supponendo che gli elementi del vettore abbiano valori distinti. Possiamo esprimere questa relazione cosí:

$$\begin{aligned} \mathit{ord}(x, y) &\Leftrightarrow \mathit{permutazione}(x, y) \wedge \mathit{ordinato}(y) \\ \mathit{permutazione}(x, y) &\Leftrightarrow \forall k((1 \leq k \wedge k \leq N) \Rightarrow \\ &\qquad \exists i(1 \leq i \wedge i \leq N \wedge y_i = x_k) \wedge \end{aligned}$$

$$\exists j(1 \leq j \wedge j \leq N \wedge x_j = y_k))$$
$$ordinato(x) \Leftrightarrow \forall k(1 \leq k \wedge k < N \Rightarrow x_k \leq x_{k+1})$$

Possiamo verificare la correttezza di un programma che ordina un vettore di N elementi rispetto alla specifica. Trascureremo la parte di specifica relativa alla permutazione di un vettore. L'ordinamento di un vettore `v` può essere ottenuto col seguente frammento di programma, che implementa l'algoritmo bubblesort, dove $M = N - 1$ (ricordiamo che in C++ gli indici di un array di N elementi vanno da 0 a $N - 1$):

```
for (i = 0; i < M; i++) {          // 0 ≤ i < M
    for (j = 0; j < M-i; j++) {    // 0 ≤ j < M − i
        if (v[j] > v[j+1]) {
            t = v[j];
            v[j] = v[j+1];
            v[j+1] = t;
        } // v_j ≤ v_{j+1}                                   (i)
    } // ∀k(M − i − 1 ≤ k < M ⇒ v_k ≤ v_{k+1})               (ii)
} // ∀k(0 ≤ k < M ⇒ v_k ≤ v_{k+1})                          (iii)
```

L'asserzione (i) vale dopo l'esecuzione dell'istruzione `if`, per il valore corrente di `j`. Le asserzioni (ii) e (iii) valgono, rispettivamente, all'uscita del loop interno (quando sono stati ordinati gli ultimi $i + 2$ elementi) e del loop esterno (quando sono stati ordinati tutti gli elementi). Le altre due asserzioni esprimono gli intervalli dei valori assunti da `i` e `j`. La forma $A \leq B < C$ è un'abbreviazione di $A \leq B \wedge B < C$.

L'asserzione (i) può essere verificata informalmente considerando le operazioni svolte nel corpo dell'istruzione `if` (una verifica formale richiede una definizione della semantica dell'assegnamento).

Possiamo verificare la formula (ii) per induzione sul valore di i, che controlla il ciclo esterno.

Per $i = 0$, la (ii) diventa

$$\forall k(M - 1 \leq k < M \Rightarrow v_k \leq v_{k+1})$$

La variabile k assume solo il valore $M - 1$, e j varia fra 0 ed $M - 1$, pertanto all'uscita del ciclo si ha dalla (i) che $v_{M-1} \leq v_M$, e la (ii) è verificata per $i = 0$.

Per un $\bar{\imath}$ arbitrario purché minore di $M-1$, la (ii) (che è vera per l'ipotesi induttiva) assicura che gli ultimi $\bar{\imath}+2$ elementi del vettore sono ordinati. Sono state eseguite $M-\bar{\imath}$ iterazioni del ciclo interno, e (dalla (i)) $v_{M-\bar{\imath}-1} < v_{M-\bar{\imath}}$. Per $i = \bar{\imath}+1$, il ciclo interno viene eseguito $M-\bar{\imath}-1$ volte, e all'uscita del ciclo $v_{M-\bar{\imath}-2} < v_{M-\bar{\imath}-1}$. Quindi gli ultimi $i+2$ elementi sono ordinati. La (ii) è quindi dimostrata.

La formula (iii) si ottiene dalla (ii) ponendovi $i = M-1$, essendo $M-1$ l'ultimo valore assunto da i. Risulta quindi che all'uscita del ciclo esterno il vettore è ordinato, q.e.d.

Osserviamo che sono state sviluppate delle logiche destinate espressamente alla verifica del software, fra le quali la piú nota è la logica di Floyd e Hoare [12, 17].

Un'altra teoria formale per la FOL

Il *calcolo dei sequenti* è un sistema formale il cui linguaggio ha come espressioni principali i *sequenti*, formule aventi questa struttura:

$$A_1, A_2, \ldots, A_n \vdash B_1, B_2, \ldots, B_m \ ,$$

ove gli A_i sono detti *antecedenti* (o, collettivamente, *l'antecedente*) ed i B_i *conseguenti* (*il conseguente*). Il simbolo '$\vdash$', che qui chiameremo *simbolo di sequente*, si può leggere "*comporta*" (*yields*). Ciascun A_i e B_i, a sua volta, è una formula qualsiasi del linguaggio visto nelle sezioni precedenti (o anche di altri linguaggi), purché non contenga il simbolo di sequente.

Informalmente, un sequente può essere considerato come una notazione alternativa per questa espressione:

$$A_1 \wedge A_2 \wedge \ldots \wedge A_n \Rightarrow B_1 \vee B_2 \vee \ldots \vee B_m \ ,$$

per cui un sequente corrisponderebbe all'implicazione fra la congiunzione degli antecedenti e la disgiunzione dei conseguenti. *Questa corrispondenza non è formalmente corretta*, ma può aiutare a ricordare certe regole del calcolo dei sequenti.

Un sequente è vero se vale almeno una di queste condizioni:

- almeno un antecedente è falso;
- almeno un conseguente è vero;

$$\frac{}{\Gamma,A\vdash A,\Delta}\text{axm} \qquad \frac{\Gamma\vdash\Delta,A \;\; A,\Gamma\vdash\Delta}{\Gamma\vdash\Delta}\text{ cut} \qquad \frac{A,A,\Gamma\vdash\Delta}{A,\Gamma\vdash\Delta}\text{ctr L} \qquad \frac{\Gamma\vdash\Delta,A,A}{\Gamma\vdash\Delta,A}\text{ctr R}$$

$$\frac{\Gamma\vdash\Delta,A}{\neg A,\Gamma\vdash\Delta}\neg L \qquad \frac{A,\Gamma\vdash\Delta}{\Gamma\vdash\Delta,\neg A}\neg R \qquad \frac{A,B,\Gamma\vdash\Delta}{A\wedge B,\Gamma\vdash\Delta}\wedge L \qquad \frac{\Gamma\vdash\Delta,A \;\; \Gamma\vdash\Delta,B}{\Gamma\vdash\Delta,A\wedge B}\wedge R$$

$$\frac{A,\Gamma\vdash\Delta \;\; B,\Gamma\vdash\Delta}{A\vee B,\Gamma\vdash\Delta}\vee L \qquad \frac{\Gamma\vdash\Delta,A,B}{\Gamma\vdash\Delta,A\vee B}\vee R \qquad \frac{\Gamma\vdash\Delta,A \;\; B,\Gamma\vdash\Delta}{A\Rightarrow B,\Gamma\vdash\Delta}\Rightarrow L \qquad \frac{A,\Gamma\vdash\Delta,B}{\Gamma\vdash A\Rightarrow B,\Delta}\Rightarrow R$$

$$\frac{A[x\leftarrow t],\Gamma\vdash\Delta}{\forall x.A,\Gamma\vdash\Delta}\forall L \qquad \frac{\Gamma\vdash\Delta,A[x\leftarrow y]}{\Gamma\vdash\forall x.A,\Delta}\forall R \qquad \frac{A[x\leftarrow y],\Gamma\vdash\Delta}{\exists x.A,\Gamma\vdash\Delta}\exists L \qquad \frac{\Gamma\vdash\Delta,A[x\leftarrow t]}{\Gamma\vdash\exists x.A,\Delta}\exists R$$

Tabella 3.3: Le regole d'inferenza nel calcolo dei sequenti.

- almeno una formula compare sia come antecedente che come conseguente (segue dalle due condizioni precedenti).

Il calcolo dei sequenti ha un solo assioma:

$$\Gamma, A \vdash A, \Delta \ ,$$

dove Γ e Δ sono multiinsiemi di formule.

Le regole d'inferenza sono mostrate nella Tabella 3.3. Nella tabella, i simboli immediatamente a destra di ciascuna regola d'inferenza servono a identificare sinteticamente la regola. Per esempio, il simbolo '$\neg L$' si può leggere come "*inserimento a sinistra* (L, *left*) *della negazione*". Infatti la regola corrispondente dice che da un sequente della forma $\Gamma \vdash \Delta, A$ si può dedurre un sequente della forma $\neg A, \Gamma \vdash \Delta$; analogamente per la regola etichettata dal simbolo '$\neg R$' (R sta per *right*). In base a queste due regole, qualsiasi formula si può spostare da un lato all'altro del simbolo di sequente, negandola (analogamente ai termini additivi di un'equazione algebrica).

Le etichette '*axm*', '*cut*' e '*ctr*' si possono leggere come '*assioma*', '*taglio*' e '*contrazione*'. L'unico assioma del sistema formale viene visto come una regola di inferenza con un insieme vuoto di premesse.

Nel calcolo dei sequenti, la dimostrazione di una formula F si costruisce all'indietro, partendo da un sequente della forma $\vdash F$. Le regole d'inferenza vengono applicate all'indietro: dato un sequente, si cerca una regola la cui conseguenza abbia le stessa struttura del sequente, e questo viene sostituito dalle premesse. Poiché le regole d'inferenza possono avere due premesse, questo procedimento costruisce un albero di sequenti, chiamato *albero di*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A, B \vdash A}\,\text{axm}}{\neg A, A, B \vdash}\,\neg L \quad \dfrac{\dfrac{}{A, B \vdash B}\,\text{axm}}{\neg B, A, B \vdash}\,\neg L}{(\neg A \vee \neg B), A, B \vdash}\,\vee L}{(\neg A \vee \neg B), (A \wedge B) \vdash}\,\wedge L}{(\neg A \vee \neg B) \vdash \neg(A \wedge B)}\,\neg R}{\vdash (\neg A \vee \neg B) \Rightarrow \neg(A \wedge B)}\,\Rightarrow R$$

Figura 3.6: Una dimostrazione nel calcolo dei sequenti.

dimostrazione (*proof tree*), avente per radice il sequente iniziale. Applicando le varie regole, ogni ramo dell'abero può crescere dando origine a nuovi rami, ma quando un sequente ha la stessa struttura dell'assioma ($\Gamma, A \vdash A, \Delta$) la regola *axm* produce un sequente vuoto. La dimostrazione termina con successo se e quando tutti i rami sono chiusi con l'assioma. La Figura 3.6 mostra come esempio la dimostrazione della formula $\neg A \vee \neg B \Rightarrow \neg(A \wedge B)$.

3.5.5 Logiche tipate

Nelle logiche tipate, il dominio è ripartito in *tipi* (*types*, *sorts*). Esiste un quantificatore per ciascun tipo, per ogni predicato viene fissato il tipo di ciascun argomento, e per ogni funzione vengono fissati i tipi degli argomenti e del risultato. Le logiche tipate sono equivalenti alle logiche non tipate, nel senso che qualsiasi espressione di una logica tipata può essere tradotta in un'espressione non tipata equivalente, ma permettono di esprimere le specifiche in modo piú naturale, e soprattutto permettono di verificare, anche automaticamente, la correttezza delle espressioni, analogamente a quanto avviene con i linguaggi di programmazione tipati.

3.5.6 Logiche di ordine superiore

Nella logica del primo ordine le variabili possono rappresentare solo entità individuali, non funzioni, relazioni o insiemi. Quindi nella logica del primo ordine si possono esprimere delle frasi come "*per ogni x reale, $x^2 = x \cdot x$*", mentre non si possono esprimere delle frasi come "*per ogni funzione f di un numero reale, $f^2(x) = f(x) \cdot f(x)$*".

Nelle logiche di ordine superiore, le variabili possono rappresentare anche funzioni e relazioni, permettendo cosí di eprimere frasi come "*se x e y sono*

numeri reali e $x = y$, allora per ogni P tale che P sia un predicato unario si ha $P(x) = P(y)$". Generalmente le logiche di ordine superiore sono tipate.

3.5.7 Il *Prototype verification system*

Il *Prototype verification system* (PVS) è un dimostratore interattivo di teoremi sviluppato al Computer Science Laboratory dell'SRI International [27]. Il suo sistema formale si basa su un linguaggio tipato di ordine superiore e sul calcolo dei sequenti. È stato applicato in molti campi, fra cui la verifica formale di sistemi digitali, algoritmi, e sistemi safety-critical.

La verifica delle proprietà di un sistema si svolge come segue:

- si descrive il sistema per mezzo di una teoria, che comprende definizioni di tipi, variabili e funzioni, e gli assiomi richiesti;
- si scrivono le formule che rappresentano le proprietà da dimostrare;
- si seleziona una di tali formule e si entra nell'ambiente di dimostrazione interattiva;
- in tale ambiente si usano dei comandi che applicano le regole di inferenza del calcolo dei sequenti, trasformando il sequente iniziale fino ad ottenere la dimostrazione (se possibile).

La dimostrazione, quindi, non viene eseguita automaticamente dallo strumento, ma viene guidata dall'utente, che ad ogni passo sceglie il comando da applicare. Ciascun comando, però, può applicare una combinazione di piú regole di inferenza o applicarle ripetutamente, per cui una dimostrazione complessa si può spesso risolvere in pochi passi.

Il seguente esempio mostra una semplice teoria sulla struttura algebrica dei gruppi, in cui si vuole dimostrare una proprietà della funzione *inverso*:

```
group : THEORY
BEGIN
  G : TYPE+      % insieme non interpretato, non vuoto
  e : G          % elemento neutro
  i : [G -> G]   % inverso
  * : [G,G -> G] % operazione binaria da G x G a G
  x,y,z : VAR G
  associative : AXIOM
    (x * y) * z = x * (y * z)
```

```
  id_left : AXIOM
    e * x = x
  inverse_left : AXIOM
    i(x) * x = e
  inverse_associative : THEOREM
    i(x) * (x * y) = y
END group
```

Questa teoria descrive una struttura algebrica formata dall'insieme G e dall'operazione binaria '$*$' (che chiameremo, per semplicità, "prodotto") che gode delle proprietà di chiusura, associatività, esistenza dell'elemento neutro e invertibilità.

La prima dichiarazione dice che G è un tipo *non interpretato*, cioè non definito in termini di altri tipi, e non vuoto (indicato dal simbolo '+'). Seguono le dichiarazioni dell'operazione di inversione i e dell'operazione binaria caratteristica del gruppo.

I simboli x, y e z sono variabili sull'insieme G che vengono usate nelle definizioni successive.

I tre assiomi definiscono la proprietà di associatività, l'elemento neutro (o identico) e la proprietà di invertibilità. Le variabili che compaiono in queste formule hanno implicitamente il quantificatore universale.

Infine, un teorema da dimostrare: *per ogni x e y appartenenti a G, il prodotto dell'inverso di x per il prodotto di x ed y è uguale a y.*

Per dimostrare il teorema, si seleziona lo stesso col cursore e si dà il comando che attiva l'ambiente di simulazione (*dimostratore*, *prover*). Viene mostrato il sequente iniziale, o *goal*:

```
inverse_associative :

  |-------
{1}   FORALL (x, y: G): i(x) * (x * y) = y

Rule?
```

ovvero $\vdash \forall x \forall y (i(x) * (x * y)) = y$.

Si inseriscono gli assiomi nell'antecedente:

```
Rule? (lemma "associative")

{-1}  FORALL (x, y, z: G): (x * y) * z = x * (y * z)
  |-------
[1]   FORALL (x, y: G): i(x) * (x * y) = y

Rule? (lemma "inverse_left")
   ...
Rule? (lemma "id_left")
   ...
```

Nel conseguente, si inseriscono delle costanti arbitrarie (dette *di Skolem*) per eliminare il quantificatore:

```
Rule? (skosimp*)

[-1]  FORALL (x: G): e * x = x
[-2]  FORALL (x: G): i(x) * x = e
[-3]  FORALL (x, y, z: G): (x * y) * z = x * (y * z)
  |-------
{1}   i(x!1) * (x!1 * y!1) = y!1
```

Si sostituiscono le costanti di Skolem nelle formule dell'antecedente:

```
Rule? (inst -3 "i(x!1)" "x!1" "y!1")
   ...
Rule? (inst -2 "x!1")
   ...
Rule? (inst -1 "x!1")

{-1}  e * x!1 = x!1
[-2]  i(x!1) * x!1 = e
[-3]  (i(x!1) * x!1) * y!1 = i(x!1) * (x!1 * y!1)
  |-------
[1]   i(x!1) * (x!1 * y!1) = y!1
```

Si applicano delle semplificazioni secondo l'algebra booleana:

```
Rule? (grind)
```

```
[-1]  e * x!1 = x!1
[-2]  i(x!1) * x!1 = e
{-3}  e * y!1 = i(x!1) * (x!1 * y!1)
  |-------
[1]   i(x!1) * (x!1 * y!1) = y!1
```

Si usa la formula (-3) dell'antecedente per semplificare il conseguente:

```
Rule? (replace -3 :dir RL)

[-1]  e * x!1 = x!1
[-2]  i(x!1) * x!1 = e
[-3]  e * y!1 = i(x!1) * (x!1 * y!1)
  |-------
{1}   e * y!1 = y!1
```

Si reinserisce l'assioma dell'identità:

```
Rule? (lemma "id_left")

{-1}  FORALL (x: G): e * x = x
[-2]  e * x!1 = x!1
[-3]  i(x!1) * x!1 = e
[-4]  e * y!1 = i(x!1) * (x!1 * y!1)
  |-------
[1]   e * y!1 = y!1
```

E infine si sostituisce la variabile di Skolem del conseguente nell'assioma dell'identità, concludendo la dimostrazione:

```
Rule? (inst -1 "y!1")

Q.E.D.
```

Un esempio di applicazione

Consideriamo un semplice caso per mostrare come un sistema tecnico, specificamente un circuito logico, si possa modellare in PVS.

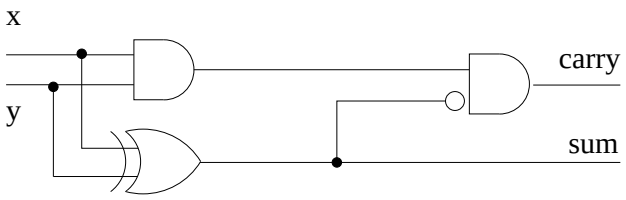

Figura 3.7: Un'implementazione del semiaddizionatore.

Un semiaddizionatore (*half adder*, *HA*) è un circuito digitale i cui due ingressi rappresentano la codifica su una cifra binaria di due numeri naturali, e la cui uscita è la codifica binaria su due cifre della loro somma. La cifra meno significativa viene chiamata *sum* (somma) e la piú significativa *carry* (riporto).

La Fig. 3.7 mostra un'implementazione, di cui vogliamo verificare la correttezza.

La seguente teoria contiene il modello del circuito e il teorema di correttezza da dimostrare:

```
HA : THEORY
BEGIN
  x,y : VAR bool

  HA(x,y) : [bool, bool] =
     ((x AND y) AND (NOT (x XOR y)),      % carry
      (x XOR y))                          % sum

  % convert Boolean to natural
  b2n(x) : nat = IF x THEN 1 ELSE 0 ENDIF

  HA_corr : THEOREM                       % correttezza
    LET (carry, sum) = HA(x, y) IN
      2*b2n(carry) + b2n(sum) = b2n(x) + b2n(y)
END HA
```

La funzione *HA* restituisce una coppia ordinata di valori booleani corrispondenti al riporto e alla somma, calcolati per mezzo delle espressioni booleane implementate dal circuito. La funzione *b2n* converte un valore booleano in un numero naturale.

Nel teorema di correttezza, l'espressione *let* pone la coppia di variabili $(carry, sum)$ uguale al risultato della funzione HA. Il primo membro dell'equazione successiva calcola il numero naturale rappresentato da *carry* e *sum*, ed il secondo è la somma dei numeri naturali codificati da x e y.

In modo analogo si può dimostrare la correttezza di un'implementazione alternativa, o l'equivalenza di due implementazioni.

3.5.8 Logiche modali e temporali

Le logiche modali arricchiscono il linguaggio della logica permettendo di esprimere aspetti del ragionamento che nel linguaggio naturale possono essere espressi dai modi grammaticali (indicativo, congiuntivo, condizionale). Per esempio, la frase "se piovesse non uscirei" suggerisce che non sta piovendo nel momento in cui viene pronunciata, *oltre* ad esprimere il fatto che la pioggia e l'azione di uscire sono legate da un'implicazione; questa sovrapposizione di significati non si ha nella frase "se piove non esco". In particolare, una logica modale può distinguere una verità necessaria (p.es., "tutti i corpi sono soggetti alla legge di gravità", nel mondo fisico in cui viviamo) da una verità contingente ("una particolare mela sta cadendo"). Le logiche temporali sono una classe di logiche modali, e servono ad esprimere il fatto che certe formule sono vere o false a seconda dell'intervallo di tempo in cui si valutano.

Le logiche modali presuppongono che le loro formule vengano valutate rispetto a piú "mondi possibili" (universi di discorso), a differenza della logica non modale che si riferisce ad un solo mondo. Il fatto che una formula sia necessariamente vera corrisponde, per la semantica delle logiche modali, al fatto che la formula è vera in tutti i mondi considerati possibili in una particolare logica.

La sintassi delle logiche modali è quella dello logiche non modali, a cui si aggiungono *operatori modali*:

- operatore di necessità $\Box$;
- operatore di possibilità $\Diamond$;

L'insieme delle formule ben formate viene quindi esteso con questa regola:

- se $\mathcal{F}$ è una formula, allora sono formule anche $\Box\mathcal{F}$ e $\Diamond\mathcal{F}$.

La semantica di una logica modale è data da una *terna di Kripke* $\langle \mathbf{W}, \mathcal{R}, V \rangle$, dove:

- l'insieme $\mathbf{W}$ è l'insieme dei mondi (o interpretazioni);
- $\mathcal{R}$ è la *funzione di accessibilità* (o *visibilità*) : $\mathcal{R} : \mathbf{W} \to \mathbf{W}$;
- V è la *funzione di valutazione* $V : \mathcal{W} \times \mathbf{W} \to \mathbb{B}$;

La funzione di accessibilità assegna una struttura all'insieme dei mondi possibili, determinando quali mondi sono considerati possibili a partire dal "mondo attuale": p.es., in una logica temporale, i mondi possibili in un dato istante sono quelli associati agli istanti successivi. Si suppone che la funzione di accessibilità sia riflessiva e transitiva.

La funzione di valutazione è tale che:

- $V(\Box\mathcal{F}, w) = \mathbf{T}$ se e solo se, per ogni $v \in \mathbf{W}$ tale che $\mathcal{R}(w, v)$, si ha $V(\mathcal{F}, v) = \mathbf{T}$;
- $V(\Diamond\mathcal{F}, w) = \mathbf{T}$ se e solo se esiste un $v \in \mathbf{W}$ tale che $\mathcal{R}(w, v)$ e $V(\mathcal{F}, v) = \mathbf{T}$;

Seguono alcuni assiomi per la logica modale:

$$\Box\neg\mathcal{P} \Leftrightarrow \neg\Diamond\mathcal{P} \tag{3.14}$$

$$\Box(\mathcal{P} \Rightarrow \mathcal{Q}) \Rightarrow (\Box\mathcal{P} \Rightarrow \Box\mathcal{Q}) \tag{3.15}$$

$$\Box\mathcal{P} \Rightarrow \mathcal{P} \tag{3.16}$$

I tre schemi di assiomi si possono leggere, rispettivamente, cosí:

- (3.14): $\mathcal{P}$ è necessariamente falso se e solo se $\mathcal{P}$ non può essere vero.
- (3.15): se $\mathcal{P}$ implica necessariamente $\mathcal{Q}$, allora la necessità di $\mathcal{P}$ implica la necessità di $\mathcal{Q}$.
- (3.15): la necessità di $\mathcal{P}$ implica $\mathcal{P}$.

Logica temporale

Nella logica temporale, i mondi possibili rappresentano *stati* del sistema su cui vogliamo ragionare, corrispondenti a diversi istanti del tempo. La relazione di accessibilità descrive la struttura del tempo: per esempio, se ammettiamo che ad uno stato possano seguire due o piú stati alternativi, abbiamo un tempo ramificato. Si possono quindi considerare dei tempi lineari, o ramificati, o circolari (sistemi periodici), o ancora piú complessi. Inoltre il tempo può essere continuo o discreto. Nel seguito ci riferiremo ad un tempo lineare e discreto. L'insieme dei mondi possibili è quindi una successione di stati s_i, ove la relazione di accessibilità è tale che $\mathcal{R}(s_i, s_j)$ se e solo se $i \leq j$.

Nelle logiche temporali i due operatori modali principali assumono questi significati:

- $\Box$ *henceforth*, d'ora in poi;
- $\Diamond$ *eventually*, prima o poi;

I due operatori sono legati dalle relazioni

$$\Box\mathcal{F} \Leftrightarrow \neg\Diamond\neg\mathcal{F}$$
$$\Diamond\mathcal{F} \Leftrightarrow \neg\Box\neg\mathcal{F}$$

Da questi operatori se ne possono definire altri:

- $\circ$ *next*, prossimo istante;
- **U** *until*, finché ($p\mathbf{U}q$ se e solo se esiste k tale che q è vero in s_k e p è vero per tutti gli s_i con $i \leq k$);
- $\blacksquare$ *always in the past*, sempre in passato;
- $\blacklozenge$ *sometime in the past*, qualche volta in passato;

Nell'esempio seguente usiamo una logica temporale per modellare un semplice sistema a scambio di messaggi (potrebbe far parte della specifica di un protocollo di comunicazione):

$$\Box(\text{send(m)} \Rightarrow \text{state} = \textbf{connected})$$
$$\Box(\text{send\_ack(m)} \Rightarrow \blacklozenge\ \text{receive(m)})$$
$$\Box(\text{B.receive(m)} \Rightarrow \blacklozenge\ \text{A.send(m)})$$

$$\Box(\text{A.send(m)} \Rightarrow \Diamond\text{B.receive(m)})$$
$$\Box(\Box\Diamond\text{A.send(m)} \Rightarrow \Diamond\text{B.receive(m)})$$
$$\Box(\text{B.receive(m)} \Rightarrow \Diamond\text{B.send\_ack(m)})$$

Le prime tre formule esprimono proprietà di *sicurezza* (non succede niente di brutto):

- se viene spedito un messaggio m, la connessione è attiva;
- se viene mandato un acknowledgement per un messaggio, il messaggio è già stato ricevuto;
- se il processo B riceve un messaggio, il messaggio è già stato spedito;

Le formule successive esprimono proprietà di *vitalità* (prima o poi succede qualcosa di buono):

- se viene spedito un messaggio m, prima o poi viene ricevuto;
- se un messaggio m viene spedito piú volte, prima o poi viene ricevuto;
- se il processo B riceve un messaggio, prima o poi restituisce un acknowledgement.

3.6 Linguaggi orientati agli oggetti

In questa sezione vengono esposti i concetti fondamentali dell'analisi e specifica orientata agli oggetti, facendo riferimento ad un particolare linguaggio di specifica, l'UML (*Unified Modeling Language*).

Nell'analisi orientata agli oggetti, la descrizione un sistema parte dall'identificazione di *oggetti* (cioè elementi costitutivi) e di *relazioni* fra oggetti.

Un oggetto viene descritto sia dai propri attributi, cioè da un'insieme di proprietà che lo caratterizzano, che dalle operazioni che l'oggetto può compiere interagendo con altri oggetti; queste operazioni possono avere dei parametri di ingresso forniti dall'oggetto che *invoca* un'operazione, e restituire risultati in funzione dei parametri d'ingresso e degli attributi dell'oggetto che *esegue* l'operazione. Gli attributi, oltre a rappresentare proprietà degli oggetti, possono rappresentarne lo *stato*, e le operazioni possono modificare lo stato. Per chiarire la differenza fra *proprietà* e *stato*, possiamo pensare

di rappresentare un'automobile con due attributi, vel_max (velocità massima) e marcia: il primo rappresenta una proprietà statica, mentre il secondo rappresenta i diversi stati di funzionamento della trasmissione (potrebbe assumere i valori *retromarcia*, *folle*, *prima*...). Possiamo completare la descrizione dell'automobile con le operazioni imposta_vel_max(), marcia_alta() e marcia_bassa(): la prima permette di modificare una proprietà (per esempio dopo una modifica meccanica), le altre a cambiare lo stato di funzionamento. Quindi il modello orientato agli oggetti unifica i tre punti di vista dei linguaggi di specifica: la descrizione dei dati, delle funzioni, e del controllo.

Generalmente, in un sistema esistono piú oggetti simili, cioè delle entità distinte che hanno gli stessi attributi (con valori eventualmente, ma non necessariamente, diversi) e le stesse operazioni. Una *classe* è una descrizione della struttura e del comportamento comuni a piú oggetti. Una specifica orientata agli oggetti consiste essenzialmente in un insieme di classi e di relazioni fra classi.

Un'altra caratteristica del modello orientato agli oggetti è il ruolo che ha in esso il concetto di *generalizzazione*. Questo concetto permette di rappresentare il fatto che alcune classi hanno in comune una parte della loro struttura o del loro comportamento.

I concetti fondamentali del modello orientato agli oggetti si possono cosí riassumere:

oggetti: Un oggetto corrisponde ad un'entità individuale del sistema che vogliamo modellare. Gli oggetti hanno *attributi*, che ne definiscono le proprietà o lo stato, e *operazioni*, che ne definiscono il comportamento. Inoltre, ogni oggetto ha una *identità* che permette di distinguerlo dagli altri oggetti.

legami: Oggetti diversi possono essere in qualche relazione fra loro: tali relazioni fra oggetti sono dette *legami* (*links*).

classi: Gli oggetti che hanno la stessa struttura e comportamento sono raggruppati in classi, ed ogni oggetto è un'*istanza* di qualche classe. Una classe quindi descrive un insieme di oggetti che hanno la stessa struttura (cioè lo stesso insieme di attributi) e lo stesso comportamento, ma sono distinguibili l'uno dall'altro, e in generale, ma non necessariamente, hanno diversi valori degli attributi.

associazioni: Un'associazione sta ad un legame come una classe sta ad un oggetto: un'associazione è un insieme di link simili per struttura e significato.

altre relazioni: Si possono rappresentare altri tipi di relazioni, fra cui la generalizzazione. Osserviamo che questa è una relazione fra classi (corrispondente all'inclusione nella teoria degli insiemi), non fra oggetti.

3.6.1 L'UML

Lo UML [28] si basa su una notazione grafica orientata agli oggetti, applicabile dalla fase di analisi e specifica dei requisiti alla fase di codifica, anche se non vincola quest'ultima fase all'uso di linguaggi orientati agli oggetti: un sistema progettato in UML può essere implementato con linguaggi non orientati agli oggetti, sebbene questi ultimi, ovviamente, non siano la scelta piú naturale. Per il momento studieremo alcuni aspetti dell'UML relativi alla fase di analisi e specifica.

Il linguaggio UML, la cui standardizzazione è curata dall'OMG (Object Management Group)[8], è passato attraverso una serie di revisioni. La versione attuale (2012) è la 2.4.1. In queste dispense si vuol dare soltanto un'introduzione ai concetti fondamentali di questo linguaggio, rinunciando sia alla completezza che al rigore, per cui la maggior parte delle nozioni qui esposte è applicabile a qualsiasi versione dell'UML. Ove sia necessario mettere in evidenza qualche differenza di concetti, di notazioni o di terminologia, si userà il termine "UML2" per la versione attuale, e "UML1" per le versioni precedenti.

In UML, un sistema viene descritto da varî punti di vista, o *viste* (*views*), e le rappresentazioni corrispondenti a tali punti di vista sono realizzate da diversi tipi di diagrammi, ciascuno rivolto a determinati aspetti del sistema. Ogni diagramma è formato da *elementi di modello*, o meglio dalle loro rappresentazioni grafiche: un elemento di modello è un'insieme di informazioni che descrive una particolare caratteristica del sistema, e queste informazioni vengono rappresentate graficamente nei diagrammi. Gli elementi di modello costituiscono il "vocabolario" di cui si serve lo sviluppatore per definire un modello.

Per il momento ci occuperemo dei punti di vista piú rilevanti nella fase di analisi: il punto di vista dei *casi d'uso* (*use cases*), il punto di vista *statico*, e quello *dinamico*. Il punto di vista fondamentale è quello statico[9], cioè la descrizione delle classi e delle relazioni nel sistema modellato.

[8] v. `www.omg.org`

[9] In alcune metodologie, però, si dà un ruolo centrale al punto di vista dei casi d'uso.

La notazione UML permette di rappresentare gli elementi di modello a varî livelli di dettaglio, in modo che si possa specificare un sistema con diversi gradi di astrazione o di approssimazione. Molti elementi hanno una forma minima (per esempio, un semplice simbolo) e delle forme estese, piú ricche di informazioni. Per esempio, una classe può essere rappresentata da un rettangolo contenente solo il nome della classe, oppure da un rettangolo diviso in tre scompartimenti, contenenti nome, attributi ed operazioni, che a loro volta possono essere specificati in modo piú o meno dettagliato.

Infine, i diagrammi possono contenere delle *note*, cioè dei commenti, che si presentano come una raffigurazione stilizzata di un biglietto con un "orecchio" ripiegato, e possono essere collegate ad un elemento di modello con una linea tratteggiata.

Meccanismi di estensione

In UML esistono tre meccanismi di estensione che permettono di adattare il linguaggio a esigenze specifiche: *vincoli*, *valori etichettati* e *stereotipi*.

I *vincoli* sono annotazioni che descrivono determinate condizioni imposte al sistema specificato, come, per esempio, l'insieme di valori ammissibili per un attributo, o il fatto che certe relazioni fra oggetti siano mutuamente esclusive. I vincoli possono venire espressi in linguaggio naturale, oppure nell'*Object Constraint Language* (OCL) dell'UML, o in qualsiasi linguaggio appropriato. La sintassi UML richiede che i vincoli vengano scritti fra parentesi graffe.

I *valori etichettati* (*tagged values*) sono delle proprietà, espresse da un nome e un valore, che si possono associare ad elementi di modello. Per esempio, ad un elemento si può associare la proprietà date il cui valore è la data piú recente in cui quell'elemento è stato modificato: {date = 1 apr 1955}. Se una proprietà ha un valore logico, si scrive solo il nome della proprietà nei casi in cui è vera, altrimenti si omette: {approvato}. Osserviamo che un valore etichettato è una proprietà dell'elemento di modello, non dell'entità reale modellata. Anche i valori etichettati si scrivono fra parentesi graffe.

Gli *stereotipi* sono dei nuovi elementi di modello ottenuti da elementi già esistenti (per esempio, dall'elemento *classe* o dall'elemento *associazione*) aggiungendo informazioni di vario tipo alla loro semantica, per mezzo di vincoli e valori etichettati. Per esempio, se in un certo campo di applicazione si vogliono distinguere le classi destinate all'interfacciamento con l'utente dalle classi che rappresentano i server e da quelle che definiscono la logica

dell'applicazione, si possono definire, rispettivamente, gli stereotipi *boundary*, *server* e *controller*. A quest'ultimo si possono associare informazioni rilevanti per tutti i tipi di server, per esempio il massimo numero di richieste che possono essere messe in attesa, espresso da un valore etichettato (max_reqs). Questi stereotipi si possono definire per mezzo di diagrammi come quello mostrato in Fig. 3.8, in cui la freccia con la punta triangolare nera rappresenta la relazione di *estensione* fra uno stereotipo ed un altro elemento di modello, in questo caso la *metaclasse*[10] **Class**.

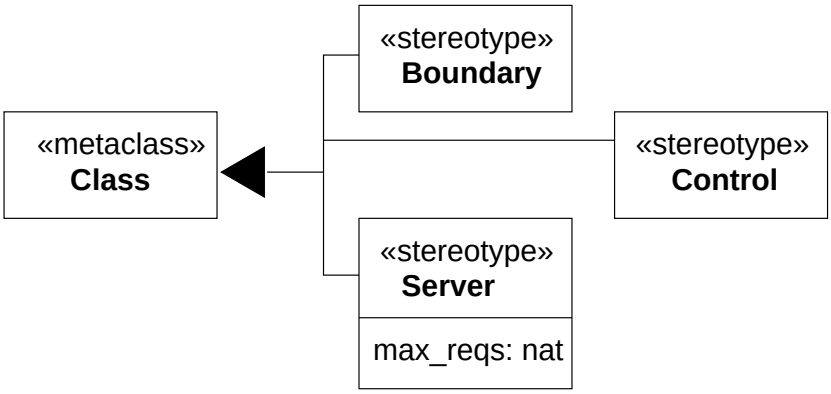

Figura 3.8: Definizione di stereotipi.

La Fig. 3.9 mostra una possibile applicazione degli stereotipi cosí definiti (maggiori dettagli sulla notazione UML nelle prossime sezioni).

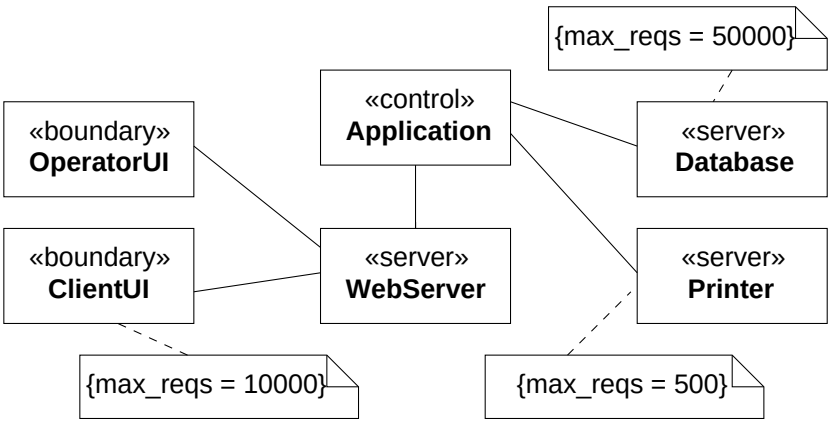

Figura 3.9: Uso di stereotipi.

Il nome di uno stereotipo viene scritto fra i caratteri $\ll$ e $\gg$, e può essere accompagnato o sostituito da un'icona.

[10]Una metaclasse è una classe che descrive un elemento di modello.

Come si è detto piú sopra, i meccanismi di estensione permettono di adattare il linguaggio UML alle esigenze di particolari campi di applicazione o processi di sviluppo. Un *profilo* è un insieme coerente e documentato di estensioni. Numerosi profili sono stati standardizzati dall'OMG (Object Management Group).

I meccanismi di estensione si possono applicare a qualsiasi elemento di modello, per cui la possibilità della loro presenza generalmente verrà sottintesa nelle successive descrizioni dei vari tipi di elementi.

3.6.2 Classi e oggetti

Come abbiamo visto, una classe rappresenta astrattamente un insieme di oggetti che hanno gli stessi attributi, operazioni, relazioni e comportamento. In un modello di analisi una classe si usa per rappresentare un concetto pertinente al sistema che viene specificato, concetto che può essere concreto (per esempio, un controllore di dispositivi) o astratto (per esempio, una transazione finanziaria). In un modello di progetto una classe rappresenta un'entità software introdotta per implementare l'applicazione.

Una classe viene rappresentata graficamente da un rettangolo contenente il nome della classe e, opzionalmente, l'elenco degli attributi e delle operazioni.

Se la classe ha uno o piú stereotipi, i loro nomi vengono scritti sopra al nome della classe. Uno stereotipo può anche essere rappresentato da un'icona nell'angolo destro in alto del rettangolo.

La rappresentazione minima di una classe consiste in un rettangolo contenente solo il nome ed eventualmente lo stereotipo. Se la classe ha uno stereotipo rappresentabile da un'icona, la rappresentazione minima consiste nell'icona e nel nome.

Attributi

Ogni attributo ha un *nome*, che è l'unica informazione obbligatoria. Le altre informazioni associate agli attributi sono:

tipo: può essere uno dei tipi fondamentali predefiniti dall'UML (corrispondenti a quelli usati comunemente nei linguaggi di programmazione), un

tipo definito in un linguaggio di programmazione, o una classe definita (in UML) dallo sviluppatore;

visibilità: *privata*, *protetta*, *pubblica*, *package*; quest'ultimo livello di visibilità significa che l'attributo è visibile da tutte le classi appartenenti allo stesso package (Sez. 4.4.1);

***scope*:** *istanza*, se l'attributo appartiene al singolo oggetto, *statico* se appartiene alla classe, cioè è condiviso fra tutte le istanze della classe, analogamente ai campi statici del C++ o del Java;

molteplicità: índica se l'attributo può essere replicato, cioè avere piú valori (si può usare per rappresentare un array);

valore iniziale: valore assegnato all'attributo quando si crea l'oggetto.

La sintassi UML per gli attributi è la seguente:

```
<visibilita‘> <nome> <molteplicita‘> : <tipo> = <val-iniziale>
```

Se un attributo ha scope statico, viene sottolineato.

La visibilità si rappresenta con i seguenti simboli:

`+`	pubblica
`#`	protetta
`~`	package
`-`	privata

La molteplicità si índica con un numero o un intervallo numerico fra parentesi quadre, come nei seguenti esempi:

`[3]`	tre valori
`[1..4]`	da uno a quattro valori
`[1..*]`	uno o piú valori
`[0..1]`	zero o un valore (attributo opzionale)

Operazioni

Ogni operazione viene identificata da una *segnatura* (*signature*) costituita dal nome della funzione e dalla *lista dei parametri*, eventualmente vuota. Per ciascun parametro si specifica il nome e, opzionalmente, le seguenti informazioni:

direzione: *ingresso* (in), *uscita* (out), *ingresso e uscita* (inout);
tipo;
valore default: valore passato al metodo che implementa la funzione, se l'argomento corrispondente al parametro non viene specificato.

Inoltre le operazioni, analogamente agli attributi, hanno scope, visibilità e tipo; quest'ultimo è il tipo dell'eventuale valore restituito. Anche queste informazioni sono opzionali.

La sintassi per le operazioni è la seguente:

```
<visibilita‘> <nome> (<lista-parametri>) : <tipo>
```

dove ciascun parametro della lista ha questa forma:

```
<direzione> <nome> : <tipo> = <val-default>
```

Solo il nome del parametro è obbligatorio, ed i parametri sono separati da virgole.

È utile osservare la distinzione fra *operazione* e *metodo*. Un'operazione è la specifica di un comportamento, specifica che si può ridurre alla semplice descrizione dei parametri o includere vincoli (per esempio, precondizioni, invarianti e postcondizioni) e varie annotazioni. Un metodo è l'implementazione di un'operazione. L'UML non ha elementi di modello destinati a descrivere i metodi, che comunque non interessano in fase di specifica. Se bisogna descrivere l'implementazione di un'operazione, per esempio nella fase di codifica, si possono usare delle note associate all'operazione.

Oggetti

Un oggetto viene rappresentato da un rettangolo contenente i nomi dell'oggetto e della classe d'appartenenza, sottolineati e separati dal carattere ':', ed opzionalmente gli attributi con i rispettivi valori. Il nome della classe o quello dell'oggetto possono mancare. In questo caso, il nome della classe viene preceduto dal carattere ':'. È possibile esprimere uno stereotipo come nella rappresentazione delle classi.

La forma minima di un oggetto consiste in un rettangolo col nome dell'oggetto e/o della classe e l'eventuale stereotipo, oppure, se è il caso, nell'icona dello stereotipo col nome dell'oggetto.

Nell'UML la rappresentazione esplicita di oggetti è piuttosto rara, in quanto il lavoro di analisi, specifica e progettazione si basa essenzialmente sulle classi e le loro relazioni, però gli oggetti possono essere usati per esemplificare delle situazioni particolari o tipiche. La figura 3.10 mostra la rappresentazione di una classe e di una sua istanza.

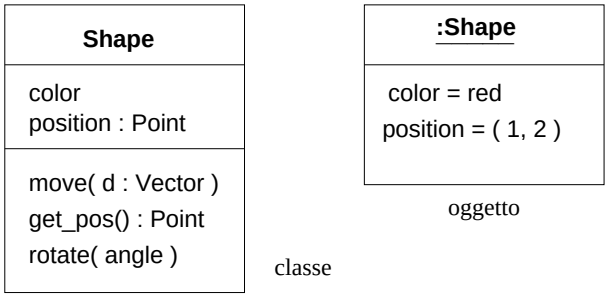

Figura 3.10: Una classe ed un oggetto

3.6.3 Associazioni e link

Un'associazione (o un link), nella forma piú semplice, si rappresenta come una linea fra le classi (o oggetti) coinvolte, etichettata col nome della relazione (o link). Se un'associazione coinvolge piú di due classi, si rappresenta con una losanga unita da linee alle classi coinvolte (analogamente per i link).

Alcuni semplici modi di rappresentare le associazioni sono mostrati in Fig. 3.11: la classe **User** rapresenta gli utenti di un sistema di calcolo, e la classe **Account** rappresenta i relativi account, ciascuno contraddistinto da un numero identificatore (uid).

La *molteplicità* di una classe in un'associazione è il numero di istanze di tale classe che possono essere in relazione con istanze della classe associata, e può essere indicata con intervalli numerici alle estremità della linea che rappresenta l'associazione: per esempio, `1`, `0..1` (zero o uno), `1..*` (uno o piú), `0..*` (zero o piú), `*` (equivalente a `0..*`).

Alle estremità di una linea si possono specificare anche i rispettivi *nomi di ruolo* (*rolenames*[11]) che suggeriscono i ruoli delle classi coinvolte nell'associazione. Il nome di ruolo serve a specificare la relazione fra le istanze della classe etichettata dal ruolo e le istanze della classe associata. Per esempio, in

[11]La documentazione ufficiale dell'UML2 usa il termine *association end*.

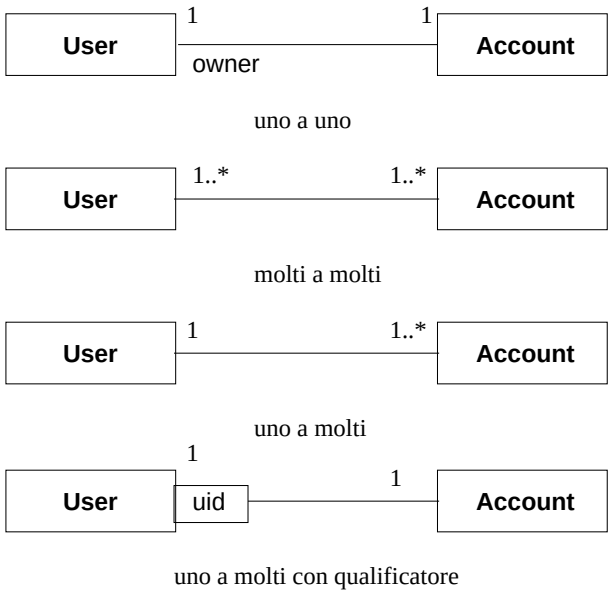

Figura 3.11: Associazioni

Fig. 3.17 c'è un'associazione fra la classe **Docente** e la classe **Dipartimento**. All'estremità di questa associazione corrispondente alla classe *Docente* si trova il nome di ruolo presidente: questo significa che un docente può assumere il ruolo di presidente di un dipartimento. Invece di indicare il ruolo, in questo caso si potrebbe dare un nome all'associazione, per esempio presiede. Questa scelta non comporterebbe ambiguità nell'interpretazione dell'associazione, essendo abbastanza ovvio che è un docente a presiedere un dipartimento, e non viceversa, ma in generale l'uso dei ruoli è piú chiaro. In ogni modo è possibile usare insieme il nome dell'associazione e i nomi dei ruoli. L'uso dei ruoli è inoltre utile nelle *autoassociazioni*, in cui oggetti di una classe sono collegati a oggetti della stessa classe, come in Fig. 3.12.

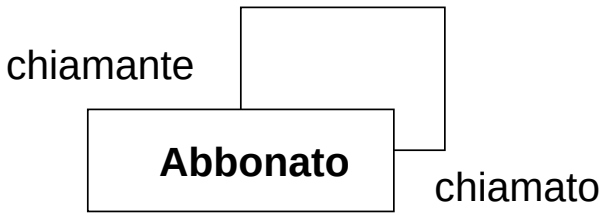

Figura 3.12: Ruoli in un'autoassociazione.

A un'estremità di un'associazione può apparire anche un *qualificatore*, rappresentato da un rettangolo avente un lato combaciante con un lato della

classe. Il qualificatore è un attributo dell'associazione, che distingue i diversi oggetti di una classe che possono essere in relazione con oggetti dell'altra. Il qualificatore rappresenta cioè un'informazione che permette di individuare una particolare istanza, fra molte, di una classe associata. Nella Fig. 3.11, la penultima associazione mostra che un utente può avere piú account, e nell'associazione successiva questa molteplicità viene ridotta a uno, grazie al qualificatore uid che identifica i singoli account.

Un'associazione può avere degli attributi e delle operazioni: si parla in questo caso di *classe associazione*. Una classe associazione viene rappresentata collegando alla linea dell'associazione il simbolo della classe (in forma estesa o ridotta), mediante una linea tratteggiata.

Aggregazione

La *aggregazione* è un'associazione che lega un'entità complessa (aggregato) alle proprie parti componenti. Nei diagrammi di classi e di oggetti, una relazione di aggregazione viene indicata da una piccola losanga all'estremità dell'associazione che si trova dalla parte della classe (o dell'oggetto) che rappresenta l'entità complessa.

La differenza fra un'associazione pura e semplice e un'aggregazione non è netta. L'aggregazione è un'annotazione aggiuntiva che esprime il concetto di "appartenenza", "contenimento", "ripartizione", o in generale di una forma di subordinazione strutturale non rigida. Per esempio, nella Fig. 3.13 la relazione fra una banca e i suoi clienti viene modellata da una semplice associazione, poiché i clienti non "fanno parte" della loro banca, mentre la relazione fra una squadra e i suoi giocatori si può modellare piú accuratamente con una aggregazione. In questo secondo caso, però, sarebbe stata accettabile anche una semplice associazione.

Un'istanza di una classe può appartenere a piú d'una aggregazione. La Fig. 3.14 mostra che la classe **Studente** partecipa, come componente, alle aggregazioni con le classi **Squadra** e **Coro**. Il diagramma di oggetti nella stessa figura mostra una possibile configurazione di istanze compatibile col diagramma delle classi: la studentessa anna appartiene a due istanze della classe **Squadra**, lo studente beppe appartiene ad un'istanza della classe **Squadra** e ad una della classe **Coro**, lo studente carlo appartiene a un'istanza della classe **Coro**.

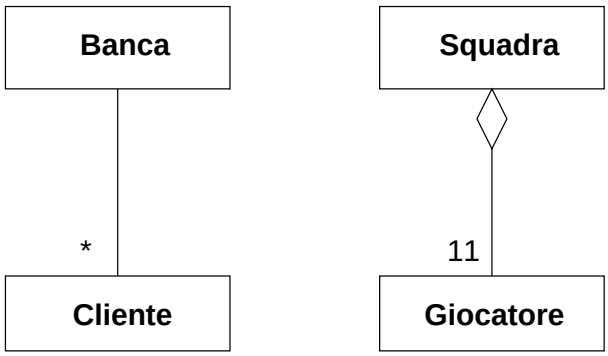

Figura 3.13: Associazioni e aggregazioni.

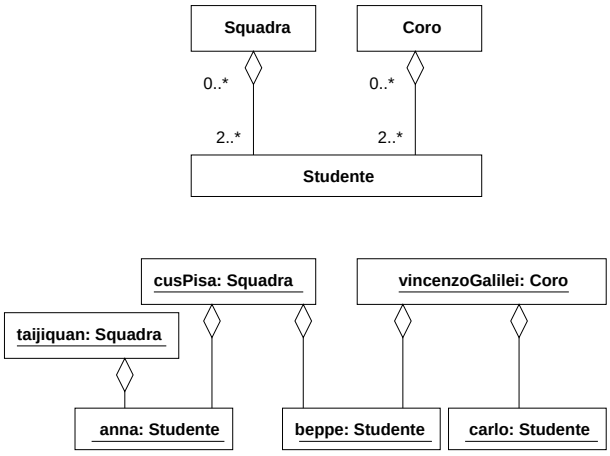

Figura 3.14: Aggregazioni multiple.

3.6.4 Composizione

La *composizione* modella una subordinazione strutturale rigida, tale cioè che l'aggregato abbia il completo e unico controllo delle parti componenti. Mentre nell'aggregazione le parti sono indipendenti dall'aggregato (esistono anche al di fuori dell'aggregazione), nella composizione esiste una dipendenza stretta fra composto e componenti. Spesso la composizione rappresenta una situazione in cui l'esistenza dei componenti coincide con quella del composto, per cui la creazione e la distruzione del composto implicano la creazione e la distruzione dei componenti. In ogni caso, un componente può appartenere ad un solo composto, e il composto è il "padrone" del componente.

La composizione si rappresenta con una losanga nera dalla parte dell'entità complessa, oppure si possono disegnare i componenti all'interno dell'en-

tità stessa. La Fig. 3.15 mostra queste due notazioni.

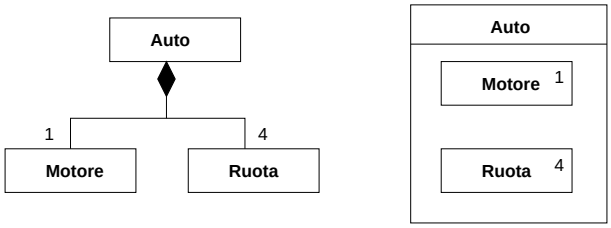

Figura 3.15: Composizione.

Una classe può appartenere come componente a piú di una composizione, ma un'istanza può appartenere ad una sola istanza. Nella Fig. 3.16 la classe **Motore** è in relazione di composizione con **Nave** e **Auto**, ma una qualsiasi sua istanza può essere componente di una sola istanza di una delle due classi.

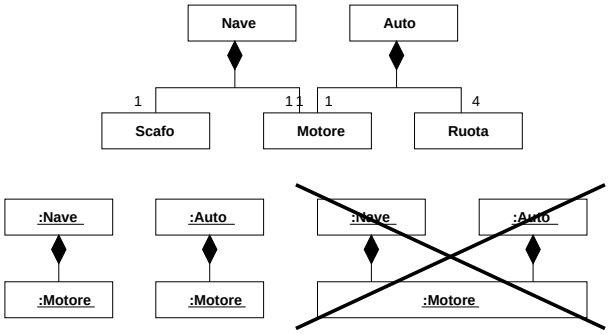

Figura 3.16: Composizione fra classi e fra oggetti.

Infine, la Fig. 3.17 mostra un semplice diagramma delle classi con associazioni, aggregazioni e composizioni.

3.6.5 Generalizzazione

Una classe (che chiameremo *classe base* o *superclasse*) *generalizza* un'altra classe (che chiameremo *classe derivata* o *sottoclasse*) quando definisce un insieme di elementi piú ampio che include l'insieme di elementi definiti dalla classe derivata: quest'ultima, cioè, è un sottoinsieme della classe base. La classe base ha meno caratteristiche (attributi, operazioni, associazioni, vincoli...) della classe derivata.

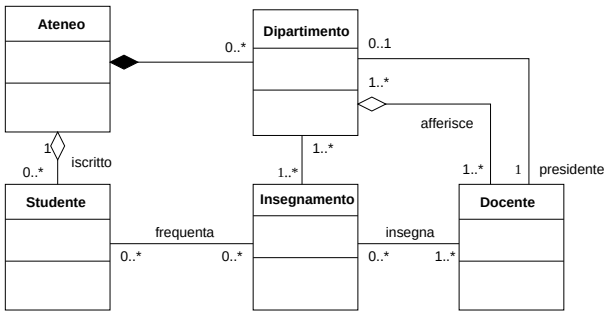

Figura 3.17: Un diagramma delle classi.

Una classe base può avere piú classi derivate, e in questo caso ne riassume alcune caratteristiche comuni (attributi, operazioni, associazioni, vincoli...). Le caratteristiche di tale classe, cioè, definiscono un insieme di oggetti che include l'unione degli insiemi di oggetti definiti dalle sottoclassi. Un oggetto appartenente ad una classe derivata, quindi, appartiene anche alla classe base. È possibile che ogni oggetto appartenente alla classe base appartenga ad almeno una delle classi derivate, oppure che alcuni oggetti appartengano solo alla classe base.

La relazione di generalizzazione si può anche chiamare *specializzazione*, cambiando il punto di vista ma restando immutato il significato: una classe è una specializzazione di un'altra classe se aggiunge delle caratteristiche alla sua struttura o al suo comportamento (è quindi un'*estensione*) oppure vi aggiunge dei vincoli (si ha quindi una *restrizione*). Per esempio, data una classe "ellisse" con gli attributi "asse maggiore" e "asse minore", la classe "ellisse colorata" è un'estensione, poiché aggiunge l'attributo "colore", mentre la classe "cerchio" è una restrizione, poiché pone il vincolo che gli attributi "asse maggiore" e "asse minore" abbiano lo stesso valore. Osserviamo, a scanso di equivoci, che anche un'estensione è un sottoinsieme della superclasse.

Il fatto che le istanze di una classe derivata siano un sottoinsieme delle istanze della classe base si esprime anche col *principio di sostituzione* (di B. Liskov): un'istanza della classe derivata può sostituire un'istanza della classe base. Questo principio è un utile criterio per valutare se una certa classe può essere descritta come una specializzazione di un'altra. È particolarmente importante verificare l'applicabilità del principio della Liskov nel caso in cui la classe derivata venga ottenuta per restrizione dalla classe base, cioè aggiungendo dei vincoli. Se, per esempio, supponiamo che la classe "ellisse" abbia due operazioni che permettono di modificare gli assi separatamente, allora la classe "cerchio" deve implementare queste operazioni in

modo che si rispetti il vincolo di uguaglianza fra i due assi: questo fa sí che le operazioni caratteristiche di "ellisse" siano applicabili a "cerchio".

Poiché una sottoclasse ha le stesse caratteristiche della superclasse, si dice che tali caratteristiche vengono *ereditate*. In particolare, vengono ereditate le operazioni, di cui si può effettuare anche una *ridefinizione* (*overriding*) nella classe derivata. In una classe derivata, cioè, si può avere un'operazione che ha la stessa *segnatura* (nome dell'operazione, tipo del valore restituito, numero, tipo e ordine degli argomenti) di un'operazione della classe base, ma una diversa implementazione. Naturalmente la ridefinizione è utile se la nuova implementazione è compatibile col significato originario dell'operazione, altrimenti non varrebbe il principio di sostituzione. Per esempio, consideriamo una classe **Studente** con l'operazione iscrivi(), che rappresenta l'iscrizione dello studente secondo la procedura normale. Se una categoria di studenti, per esempio quelli già in possesso di una laurea, richiede una procedura diversa, si può definire una classe **StudenteLaureato**, derivata da **Studente**, che ridefinisce opportunamente l'operazione iscrivi().

Una classe derivata può essere ulteriormente specializzata in una o piú classi, e una classe base può essere ulteriormente generalizzata (finché non si arriva alla classe universale che comprende tutti i possibili oggetti). Quando si hanno delle catene di generalizzazioni, può essere utile distinguere le classi basi o derivate *dirette* da quelle *indirette*, a seconda che si ottengano "in un solo passo" o no, a partire da un'altra classe. Le classi base e derivate indirette si chiamano anche, rispettivamente, *antenati* e *discendenti*.

La generalizzazione si rappresenta con una freccia terminante in un triangolo vuoto col vertice che tocca la superclasse.

Classi astratte e concrete

Una classe è *concreta* se esistono degli oggetti che siano istanze *dirette* di tale classe, cioè appartengano ad essa e non a una classe derivata. Se consideriamo due o piú classi concrete che hanno alcune caratteristiche in comune, possiamo generalizzarle definendo una classe base che riassume queste caratteristiche. Può quindi accadere che non possano esistere delle istanze dirette di questa classe base, che si dice allora *astratta*.

Per esempio, consideriamo delle classi come **Uomo**, **Lupo**, **Megattera** eccetera[12]. Gli animali appartenenti a queste classi hanno delle caratteristiche

[12]Ovviamente stiamo usando il termine "classe" in modo diverso da come viene usato

in comune (per esempio, sono omeotermi e vivipari), che si possono riassumere nella definizione di una classe base *Mammifero*. Questa classe è astratta perché non esiste un animale che sia un mammifero senza appartenere anche a una delle classi derivate.

In UML l'astrattezza di una classe (e di altri elementi di modello) si rappresenta con la proprietà abstract. Per convenzione, nei diagrammi i nomi delle entità astratte si scrivono in corsivo; se non è pratico scrivere in corsivo (per esempio in un diagramma fatto a mano) si può scrivere la parola abstract fra parentesi graffe sotto al nome della classe.

Eredità multipla

Si parla di *eredità multipla* quando una classe derivata è un sottoinsieme di due o piú classi che non sono in relazione di generalizzazione/specializzazione fra di loro. In questo caso, le istanze della classe derivata ereditano le caratteristiche di tutte le classi base.

Aggregazione ricorsiva

L'uso combinato della generalizzazione e dell'aggregazione permette di definire strutture ricorsive[13]. Nella Fig. 3.18 il diagramma delle classi specifica la struttura delle operazioni aritmetiche, e il diagramma degli oggetti ne mostra una possibile realizzazione. Si osservi che, mentre la specifica è ricorsiva, la realizzazione è necessariamente gerarchica.

Insiemi di generalizzazioni

Nei modelli di analisi la relazione di generalizzazione viene usata spesso per classificare le entità del dominio analizzato, mettendone in evidenza le reciproche affinità e differenze. Per esempio, può essere utile classificare i prodotti di un'azienda, i suoi clienti o i suoi dipendenti.

Per descrivere precisamente una classificazione, l'UML mette a disposizione il concetto di *insieme di generalizzazioni*. Informalmente, un insieme di generalizzazioni è un modo di raggruppare le sottoclassi di una classe base,

nelle scienze naturali.

[13]Osserviamo che si parla di strutture *ricorsive*, non *cicliche*. L'aggregazione è una relazione gerarchica, cioè priva di cicli.

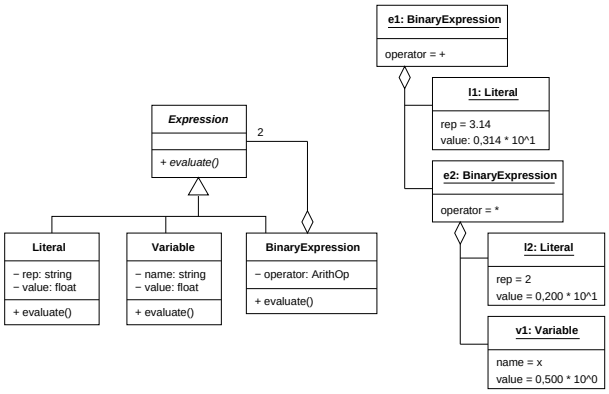

Figura 3.18: Aggregazione ricorsiva fra classi.

cioè un insieme di sottoinsiemi, a cui si può dare un nome che descriva il criterio con cui si raggruppano le sottoclassi. Un insieme di generalizzazioni è *completo* se ogni istanza della classe base appartiene ad almeno una delle sottoclassi, e *disgiunto* se le sottoclassi sono disgiunte (l'intersezione di ciascuna coppia di sottoclassi è vuota). Per default, un insieme di generalizzazioni è incompleto e disgiunto.

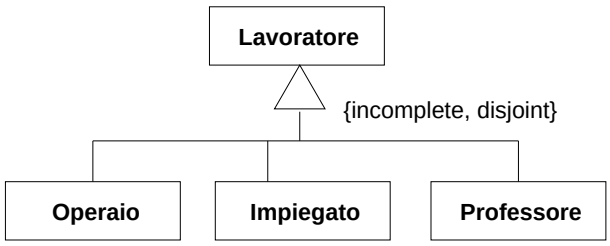

Figura 3.19: Insieme di generalizzazioni incompleto e disgiunto.

Nell'esempio di Fig. 3.19 si suppone che una persona possa fare un solo lavoro; l'insieme di generalizzazioni è quindi disgiunto, ed è incompleto perché evidentemente esistono molte altre categorie di lavoratori. In Fig. 3.20, invece, si suppone che una persona possa praticare piú di uno sport, quindi l'insieme di generalizzazioni è overlapping (non disgiunto).

Nell'esempio di Fig. 3.21 l'insieme dei dipendenti di un'azienda viene classificato secondo due criteri ortogonali, retribuzione e mansione. Si hanno quindi due insiemi di generalizzazioni, ognuno completo e disgiunto. Ogni dipendente è un'istanza sia di una classe di retribuzione (**Classe1** eccetera) che di una classe di mansioni (**Tecnico** o **Amministrativo**). Poiché nessun dipendente può appartenere *solo* a una classe di retribuzione o a una classe

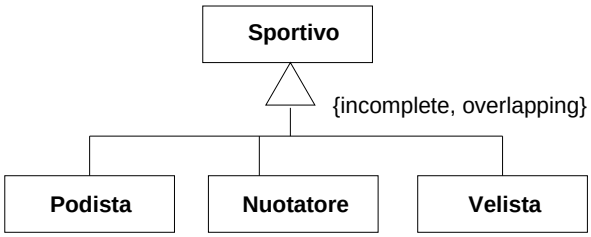

Figura 3.20: Insieme di generalizzazioni incompleto e non disgiunto.

di mansioni, tutte le classi della Fig. 3.21 (a) sono astratte. Nella Fig. 3.21 (b) si mostra una classe concreta costruita per eredità multipla da una classe di retribuzione e una classe di mansioni.

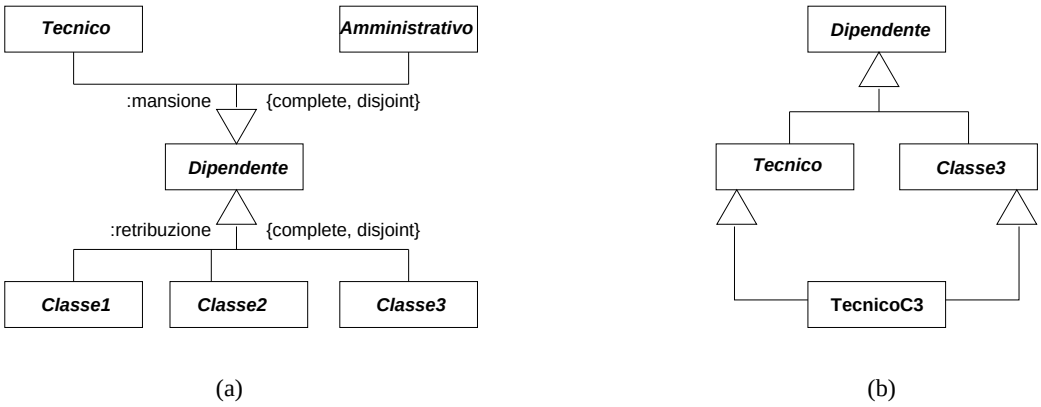

Figura 3.21: Due insiemi di generalizzazioni.

Nelle versioni dell'UML precedenti alla 2.0 c'era il concetto di *discriminante*, che aveva un significato simile a quello degli insiemi di generalizzazioni, anche se era definito in modo diverso.

Osservazione. L'esempio mostrato in Fig. 3.21 serve a chiarire il significato degli insiemi di generalizzazioni, ma non è necessariamente il modo migliore per modellare la situazione presa ad esempio. Si può osservare, infatti, che un dipendente rappresentato da un'istanza di **TecnicoC3**, secondo questo modello, non può cambiare mansione né classe di retribuzione, poiché la relazione di generalizzazione è statica e fissa rigidamente l'appartenza delle istanze di **TecnicoC3** alle superclassi *Tecnico* e *Classe3*. Si avrebbe un modello piú realistico considerando la retribuzione e la mansione come concetti *associati* ai dipendenti, come mostra la Fig. 3.22. I link fra istanze possono essere creati e distrutti dinamicamente, per cui le associazioni permettono di costruire strutture logiche piú flessibili rispetto alla generalizzazione.

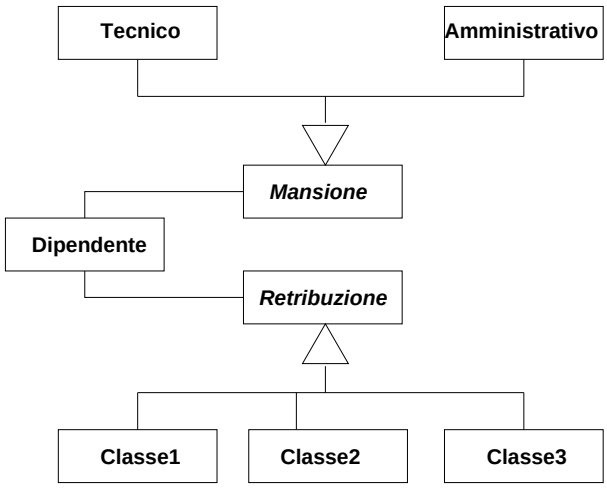

Figura 3.22: Associazioni invece di generalizzazioni.

3.6.6 Diagrammi dei casi d'uso

Un diagramma dei casi d'uso schematizza il comportamento del sistema dal punto di vista degli utenti, o, piú in generale, di altri sistemi che interagiscono col sistema specificato. Un *attore* rappresenta un'entità esterna al sistema, un *caso d'uso* rappresenta un servizio offerto dal sistema. Ciascun servizio viene espletato attraverso sequenze di messaggi scambiati fra gli attori ed il sistema.

Attori e casi d'uso sono legati da associazioni che rappresentano comunicazioni. I casi d'uso *non rappresentano sottosistemi*, per cui non possono interagire, cioè scambiarsi messaggi, fra di loro, ma possono essere legati da relazioni di *inclusione*, *estensione* e *generalizzazione*.

Il comportamento richiesto al sistema per fornire il servizio rappresentato da un caso d'uso può essere specificato in vari modi, per esempio per mezzo di macchine a stati o di descrizioni testuali. In fase di analisi può essere sufficiente una descrizione in linguaggio naturale, oppure una descrizione piú formale delle di sequenze di interazioni (*scenari*) fra attori e sistema previste per lo svolgimento del servizio.

La Fig. 3.23 mostra un semplice diagramma di casi d'uso, relativo a un sistema di pagamento POS (*Point Of Sale*). Gli attori sono il cassiere e il cliente, i servizi forniti dal sistema sono il pagamento, il rimborso e il login; quest'ultimo coinvolge solo il cassiere. Il servizio di pagamento ha due possibili estensioni, cioè comportamenti aggiuntivi rispetto a quello del caso

d'uso fondamentale.

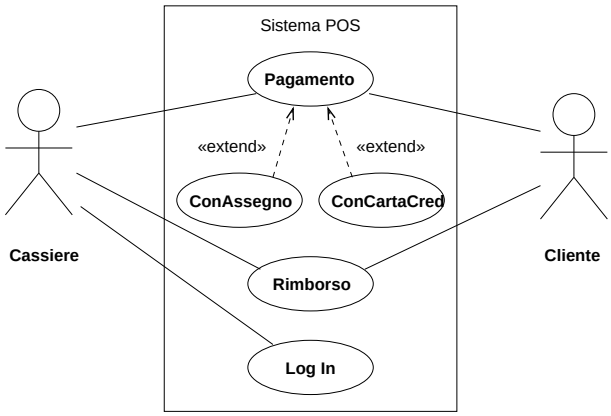

Figura 3.23: Una diagramma di casi d'uso.

3.6.7 Diagrammi di stato

Il *diagramma di stato* è uno dei diagrammi che fanno parte del modello dinamico di un sistema, e rappresenta una *macchina a stati* associata a una classe, a una *collaborazione* (interazione di un insieme di oggetti), o a un metodo. La macchina a stati descrive l'evoluzione temporale degli oggetti e la loro risposta agli stimoli esterni.

Le macchine a stati impiegate nell'UML sono basate sul formalismo degli automi a stati finiti visto in precedenza, che qui, però, viene esteso notevolmente in modo da renderlo piú espressivo e sintetico. Nell'UML viene usata la notazione degli *Statecharts*, che permette una descrizione strutturata degli stati di un automa. Questa notazione è ricca di concetti e di varianti grafiche, e ne saranno illustrati solo gli aspetti principali.

I concetti fondamentali rappresentati dal diagramma di stato sono:

occorrenze: avvenimenti associati a istanti nel tempo e (spesso implicitamente) a punti nello spazio. Per esempio, se viene premuto un tasto di un terminale, questa è un'occorrenza della pressione di un tasto. Quando lo stesso tasto (o un altro) viene premuto di nuovo, è un'altra occorrenza.

eventi: insiemi di occorrenze di uno stesso tipo. Per esempio, l'evento **TastoPremuto** potrebbe rappresentare l'insieme di tutte le possibili occorrenze della pressione di un tasto. Nel séguito, però, spesso si userà il termine "evento" al posto di "occorrenza".

stati: situazioni in cui un oggetto soddisfa certe condizioni, esegue delle attività, o semplicemente aspetta degli eventi.

transizioni: conseguentemente al verificarsi di un evento, un oggetto può modificare il proprio stato: una transizione è il passaggio da uno stato ad un altro, che si può modellare come se fosse istantaneo.

azioni: possono essere eseguite in corrispondenza di transizioni, e non sono interrompibili, per cui si possono modellare come se fosero istantanee.

attività: possono essere associate agli stati, possono essere interrotte dal verificarsi di eventi che causano l'uscita dallo stato, ed hanno una durata non nulla.

Un evento può essere una *chiamata di operazione*, un *cambiamento*, un *segnale*, o un *evento temporale*.

Un evento di chiamata avviene quando viene ricevuta una richiesta di esecuzione di un'operazione.

Un evento di cambiamento è il passaggio del valore di una condizione logica da falso a vero, e si rappresenta con la parola **when** seguita da un'espressione booleana. Per esempio, l'espressione **when** $(p > P)$ rappresenta l'evento "*p supera la soglia P*".

I segnali sono delle entità che gli oggetti si possono scambiare per comunicare fra di loro, e possono essere strutturati in una gerarchia di generalizzazione: per esempio, un ipotetico segnale **Input** può essere descritto come generalizzazione dei segnali **Mouse** e **Keyboard**, che a loro volta possono essere ulteriormente strutturati. Una gerarchia di segnali si rappresenta graficamente in modo simile ad una gerarchia di classi. Un segnale si rappresenta come un rettangolo contenente lo stereotipo ≪signal≫, il nome del segnale ed eventuali attributi.

Un evento temporale si verifica quando il tempo assume un particolare valore assoluto (per esempio, il 31 dicembre 1999), oppure quando è trascorso un certo periodo da un determinato istante (per esempio, dieci secondi dall'arrivo di un segnale). Gli eventi temporali del primo tipo si rappresentano con la parola **at** seguita da una condizione booleana (per esempio, **at** (date = 2002-12-31)), quelli del secondo tipo con la parola **after** seguita da un intervallo di tempo.

Se un evento si verifica nel corso di una transizione, non ha influenza sull'eventuale azione associata alla transizione (ricordiamo che le azioni non sono interrompibili) e viene accantonato in una riserva di eventi (*event pool*) per essere considerato nello stato successivo.

Se, mentre un oggetto si trova in un certo stato, si verificano degli eventi che non innescano transizioni associate a quello stato, l'oggetto si può comportare in due modi: i) questi eventi vengono cancellati e quindi non potranno piú influenzare l'oggetto (è come se non fossero mai accaduti), oppure ii) questi eventi vengono marcati come *differiti* (*deferred*) e memorizzati finché l'oggetto non entra in uno stato in cui tali eventi non sono piú marcati come differiti. In questo nuovo stato, gli eventi cosí memorizzati o innescano una transizione, o vengono perduti definitivamente. Gli eventi differiti in uno stato vengono dichiarati come tali nel simbolo dello stato, con la parola defer (o deferred, in UML1).

Gli stati si rappresentano come rettangoli ovalizzati contenenti opzionalmente il nome dello stato, un'eventuale attività (preceduta dalla parola do) ed altre informazioni che vedremo piú oltre. In particolare, uno stato può contenere dei sottostati. Uno stato può contenere la parola entry seguita dal nome di un'azione (separato da una barra). Questo significa che l'azione deve essere eseguita ogni volta che l'oggetto entra nello stato in questione. Analogamente, la parola exit etichetta un'azione da eseguire all'uscita dallo stato. Una coppia *evento/azione* entro uno stato significa che al verificarsi dell'evento viene eseguita l'azione corrispondente, e l'oggetto resta nello stato corrente (*transizione interna*). In questo caso non vengono eseguite le eventuali azioni di entry o di exit.

Quando bisogna indicare uno stato iniziale, si usa una freccia che parte da un cerchietto nero e raggiunge lo stato iniziale. Uno stato finale viene rappresentato da un cerchio contenente un cerchietto annerito (un "bersaglio"). Se un oggetto ha un comportamento ciclico, non viene indicato uno stato finale.

Le transizioni sono rappresentate da frecce fra gli stati corrispondenti, etichettate col nome dell'evento causante la transizione, con eventuali attributi dell'evento, con una condizione (*guardia*) necessaria per l'abilitazione della transizione (racchiusa fra parentesi quadre), e con un'azione da eseguire, separata dalle informazioni precedenti per mezzo di una barra obliqua. Ciascuna di queste tre informazioni è opzionale. Se manca l'indicazione dell'evento, la transizione avviene al termine dell'attività associata allo stato di partenza: si tratta di una transizione *di completamento*.

Un'azione può inviare dei segnali, e in questo caso si usa la parola **send** seguita dal nome e da eventuali parametri del segnale. L'invio di un segnale si può rappresentare anche graficamente, mediante una figura a forma di cartello indicatore, etichettata col nome e i parametri del segnale.

Macchine a stati gerarchiche

La descrizione del modello dinamico generalmente è gerarchica, cioè articolata su diversi livelli di astrazione, in ciascuno dei quali alcuni elementi del livello superiore vengono raffinati ed analizzati.

Un'attività associata ad uno stato può quindi essere descritta a sua volta da una macchina a stati. Questa avrà uno stato iniziale ed uno o piú stati finali. Il sottodiagramma che descrive l'attività può sempre essere disegnato all'interno dello stato che la contiene. Se non ci sono transizioni che attraversano il confine del sottodiagramma, questo può essere disegnato separatamente.

In generale, qualsiasi stato (superstato) può essere decomposto in sottostati, che ereditano le transizioni che coinvolgono il superstato.

Consideriamo, per esempio, la macchina a stati associata all'interazione fra l'utente e un centralino (Fig. 3.24). Si suppone che l'utente possa comporre numeri di tre cifre, oppure premere un tasto che seleziona un numero memorizzato. Il diagramma di questa macchina a stati non sfrutta la possibilità di composizione gerarchica offerta dagli Statechart, per cui le transizioni causate dagli eventi riaggancio devono essere mostrate per ciascuno stato successivo a quello iniziale.

Il diagramma si semplifica se raggruppiamo questi stati in un superstato (**Attivo**) e ridisegnamo le transizioni come in Fig. 3.25. La transizione in ingresso al superstato **Attivo** porta la macchina nel sottostato iniziale (**Attesa1**) di quest'ultimo, mentre la transizione di completamento fra i due stati ad alto livello avviene quando la sottomacchina dello stato **Attivo** termina il proprio funzionamento. La transizione attivata dagli eventi riaggancio viene ereditata dai sottostati: questo significa che, in qualsiasi sottostato di **Attivo**, il riaggancio riporta la macchina nello stato **Inattivo**.

Stati concorrenti

Uno stato può essere scomposto anche in *regioni concorrenti*, che descrivono attività concorrenti nell'àmbito dello stato che le contiene. Queste attività, a loro volta, sono descritte da macchine a stati. La Fig. 3.26 mostra il funzionamento di un termoventilatore, in cui il controllo della velocità e quello della temperatura sono indipendenti.

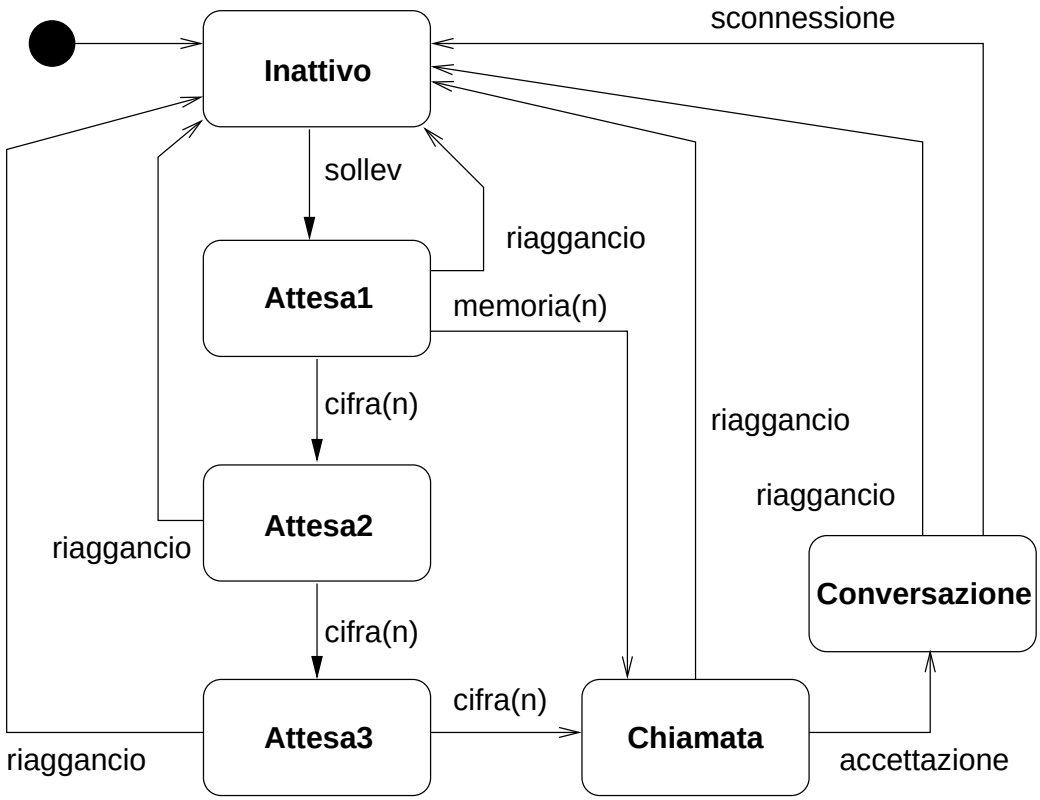

Figura 3.24: Una macchina a stati non gerarchica.

Gli stati concorrenti possono interagire attraverso *eventi condivisi*, *scambio di segnali*, *parametri* degli eventi o dei segnali, e *attributi* dell'oggetto a cui appartiene la macchina a stati. Le interazioni avvengono, oltre che con lo scambio di eventi, anche attraverso la valutazione delle espressioni che costituiscono le guardie e le azioni associate alle transizioni. L'esecuzione delle azioni può modificare gli attributi condivisi fra stati concorrenti, però è bene evitare, finché possibile, di modellare in questo modo l'interazione fra stati. Questo meccanismo di interazione, infatti, è poco strutturato e poco leggibile, e rende piú probabili gli errori nella specifica o nella realizzazione del sistema. La valutazione delle guardie, invece, non può avere effetti collaterali. In una guardia si può verificare se un oggetto si trova in un certo stato, usando l'operatore logico `oclInState` del linguaggio OCL.

È possibile descrivere attività concorrenti anche senza ricorrere alla scomposizione in regioni concorrenti, quando tali attività sono eseguite da oggetti diversi, a cui sono associate macchine a stati distinte. La Fig. 3.27 mostra il comportamento di un produttore e di un consumatore che si sincronizzano

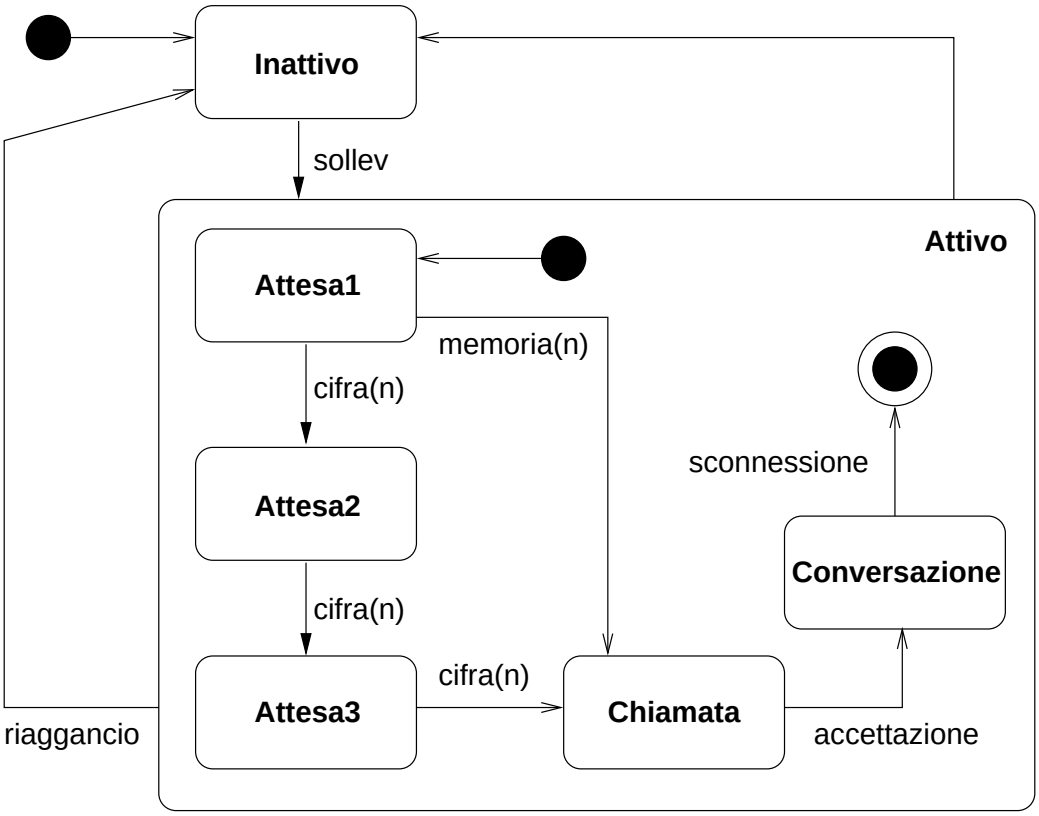

Figura 3.25: Una macchina a stati gerarchica.

scambiandosi segnali.

3.6.8 Diagrammi di interazione

I *diagrammi di interazione* mostrano gli scambi di messaggi fra oggetti. Esistono due tipi di diagrammi di interazione: i diagrammi *di sequenza* e quelli *di comunicazione*.

Un diagramma di sequenza descrive l'interazione fra piú oggetti mettendo in evidenza il flusso di messaggi scambiati e la loro successione temporale. I diagrammi di sequenza sono quindi adatti a rappresentare degli *scenari* possibili nell'evoluzione di un insieme di oggetti. È bene osservare che ciascun diagramma di sequenza rappresenta esplicitamente una o piú istanze delle possibili sequenze di messaggi, mentre un diagramma di stato definisce implicitamente tutte le possibili sequenze di messaggi ricevuti (eventi) o inviati (azioni **send**) da un oggetto interagente con altri.

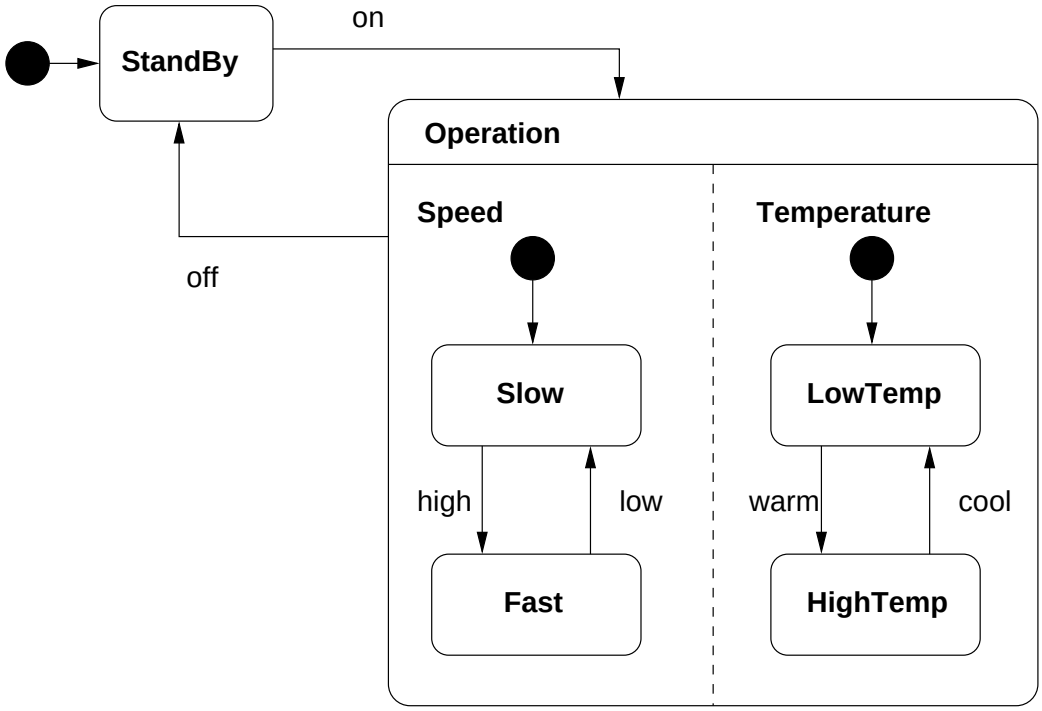

Figura 3.26: Una macchina a stati gerarchica con regioni concorrenti.

Un diagramma di sequenza è costituito da simboli chiamati *lifeline*, che rappresentano i diversi ruoli degli oggetti coinvolti nell'interazione. Sotto ogni lifeline c'è una linea verticale che rappresenta l'evoluzione temporale di ciascun oggetto. Lo scambio di un messaggio, o la chiamata di un'operazione, si rappresenta con una freccia dalla linea verticale dell'oggetto sorgente a quella del destinatario. L'ordine dei messaggi lungo le linee verticali ne rispecchia l'ordine temporale. Si può disegnare anche un asse dei tempi, parallelo alle lifeline, su cui evidenziare gli eventi, etichettando gli istanti corrispondenti con degli identificatori o con dei valori temporali, che possono essere usati per specificare vincoli di tempo. I periodi in cui un oggetto è coinvolto in un'interazione possono essere messi in evidenza sovrapponendo una striscia rettangolare alla linea verticale.

La Fig. 3.28 mostra un semplice diagramma di sequenza che descrive l'interazione di due utenti con un centralino telefonico.

Un diagramma di comunicazione (chiamato *diagramma di collaborazione* in UML1) mette in evidenza l'aspetto strutturale di un'interazione, mostrando esplicitamente i legami (istanze di associazioni) fra gli oggetti, e ricorrendo a un sistema di numerazione strutturato per indicare l'ordinamento

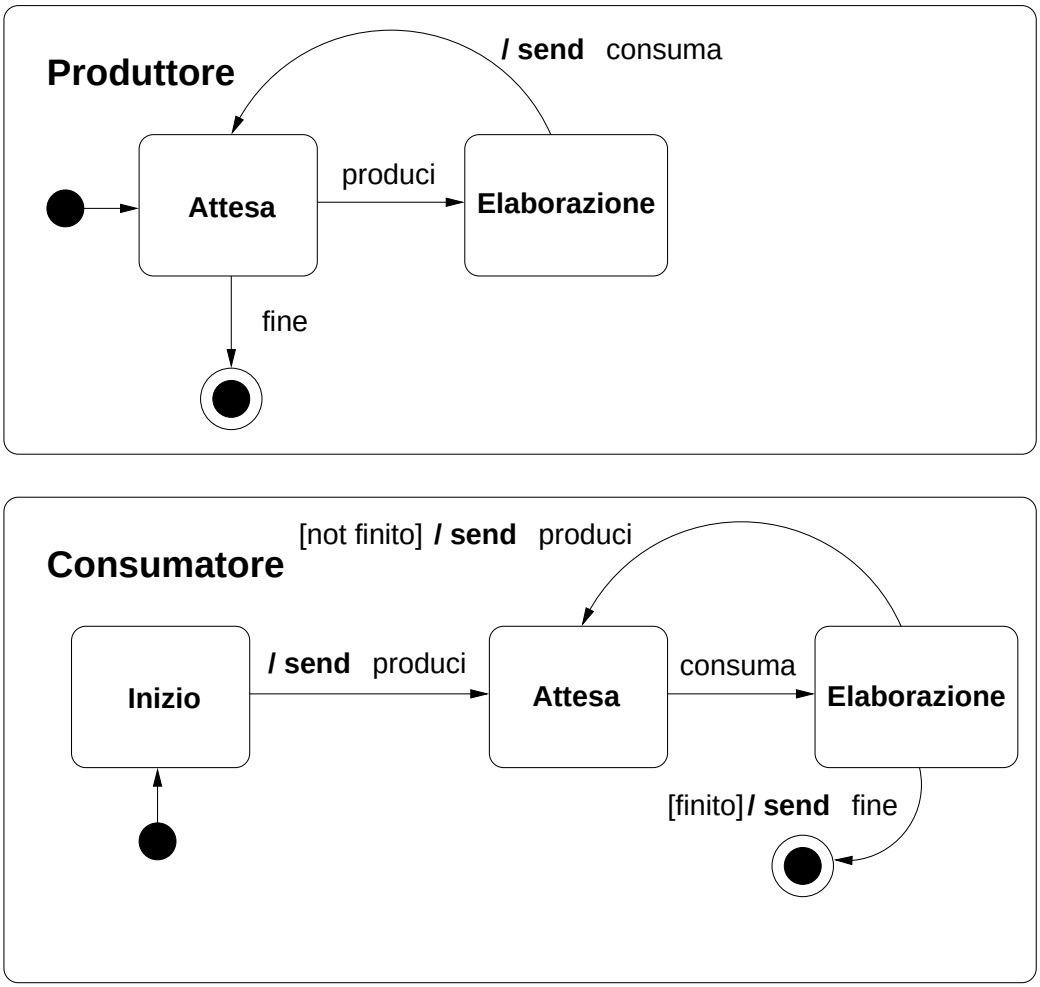

Figura 3.27: Due macchine a stati interagenti.

temporale dei messaggi.

La Fig. 3.29 mostra il diagramma di comunicazione corrispondente al diagramma di sequenza di Fig. 3.28.

3.6.9 Diagrammi di attività

I *diagrammi di attività* servono a descrivere il flusso di controllo e di informazioni dei processi. In un modello di analisi si usano spesso per descrivere i processi del dominio di applicazione, come, per esempio, le procedure richieste nella gestione di un'azienda, nello sviluppo di un prodotto, o nelle

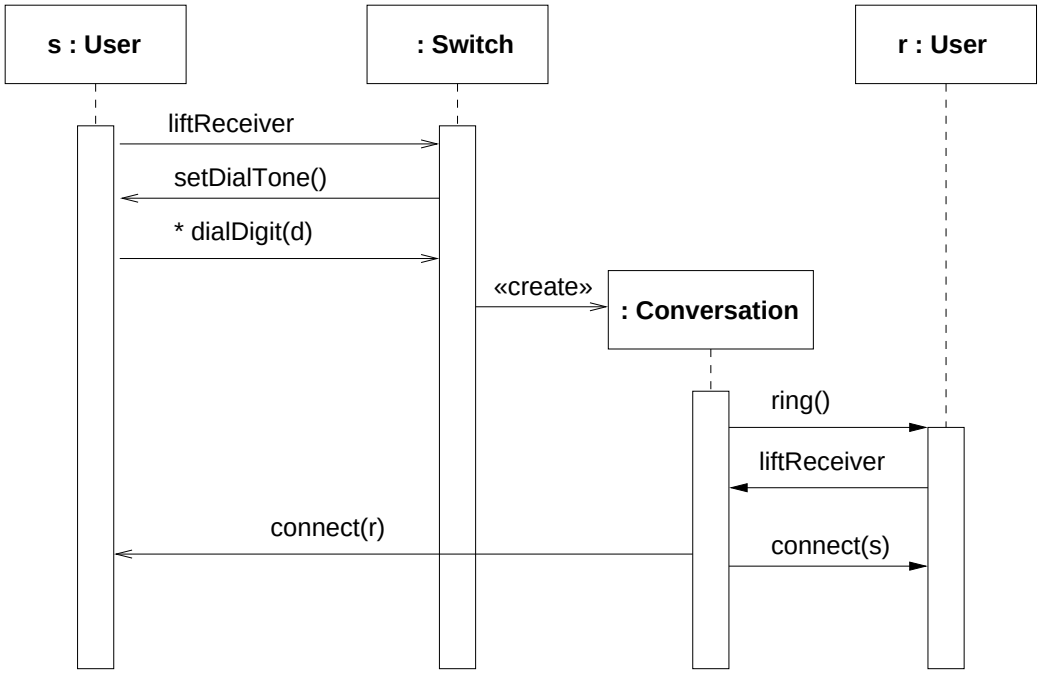

Figura 3.28: Un diagramma di sequenza.

transazioni economiche. In un modello di progetto possono essere usati per descrivere algoritmi o implementazioni di operazioni. Si può osservare che, nella loro forma piú semplice, i diagrammi di attività sono molto simili ai tradizionali *diagrammi di flusso* (*flowchart*).

Un diagramma di attività è formato da *nodi* e *archi*. I nodi rappresentano *attività* svolte in un processo, *punti di controllo* del flusso di esecuzione, o *oggetti* elaborati nel corso del processo. Gli archi collegano i nodi per rappresentare i flussi di controllo e di informazioni.

Il diagramma di Fig. 3.30 descrive un processo di controllo: alla ricezione di un segnale di *start*, se il sistema è abilitato vengono messe in funzione le valvole A e B e la pompa P. Quando tutte e due le valvole sono aperte viene emesso il segnale *A and B open*, e quando la pompa è stata avviata si apre la valvola C, e viene emesso il segnale *finished* (lo schema nel riquadro tratteggiato non fa parte del diagramma).

Le tre linee orizzontali spesse rappresentano un nodo di controllo di tipo *fork* (diramazione del flusso di controllo in attività parallele) e due nodi di tipo *join* (ricongiungimento di attività parallele).

I diagrammi di attività possono descrivere attività svolte da entità diffe-

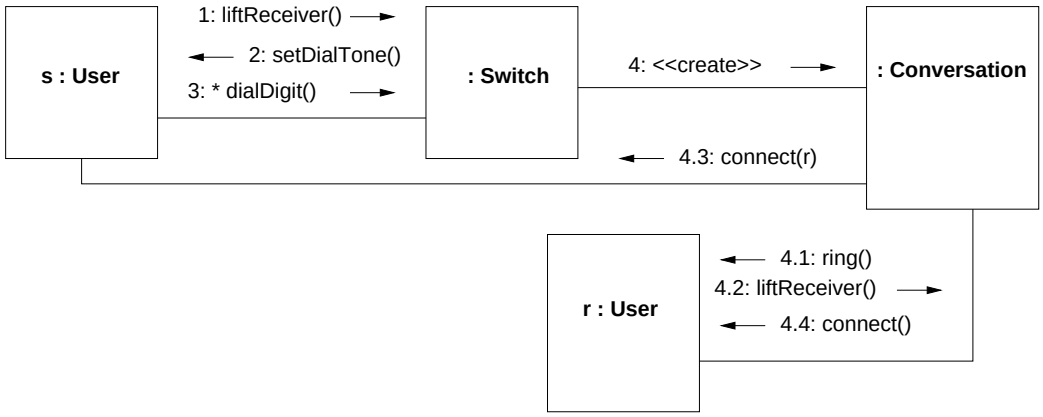

Figura 3.29: Un diagramma di comunicazione.

renti, raggruppandole graficamente. Ciascuno dei gruppi cosí ottenuti è una *partizione*, detta anche *corsia* (*swimlane*).

La Fig. 3.31 mostra un diagramma di attività che descrive (in modo molto semplificato) il processo di sviluppo di un prodotto, mostrando quali reparti di un'azienda sono responsabili per le varie attività.

Il nodo Specification è un nodo oggetto, in questo caso il documento di specifica prodotto dall'attività Design.

Letture

Obbligatorie: Cap. 5 e Sez. 4.6 Ghezzi, Jazayeri, Mandrioli, oppure Cap. 2 (esclusi 2.5.1, 2.5.2, pagg. 90–100 di 2.6.2, 2.6.4, 2.7.3) Ghezzi, Fuggetta et al., oppure Cap. 7 (esclusi 7.4.2, 7.6.2, 7.7), Sez. 8.4–8.9, Sez. 23.1, Sez. 24.1 Pressman.

Facoltative: Cap. 23 Pressman. Sulla logica, Cap. 1 e 2 Quine.

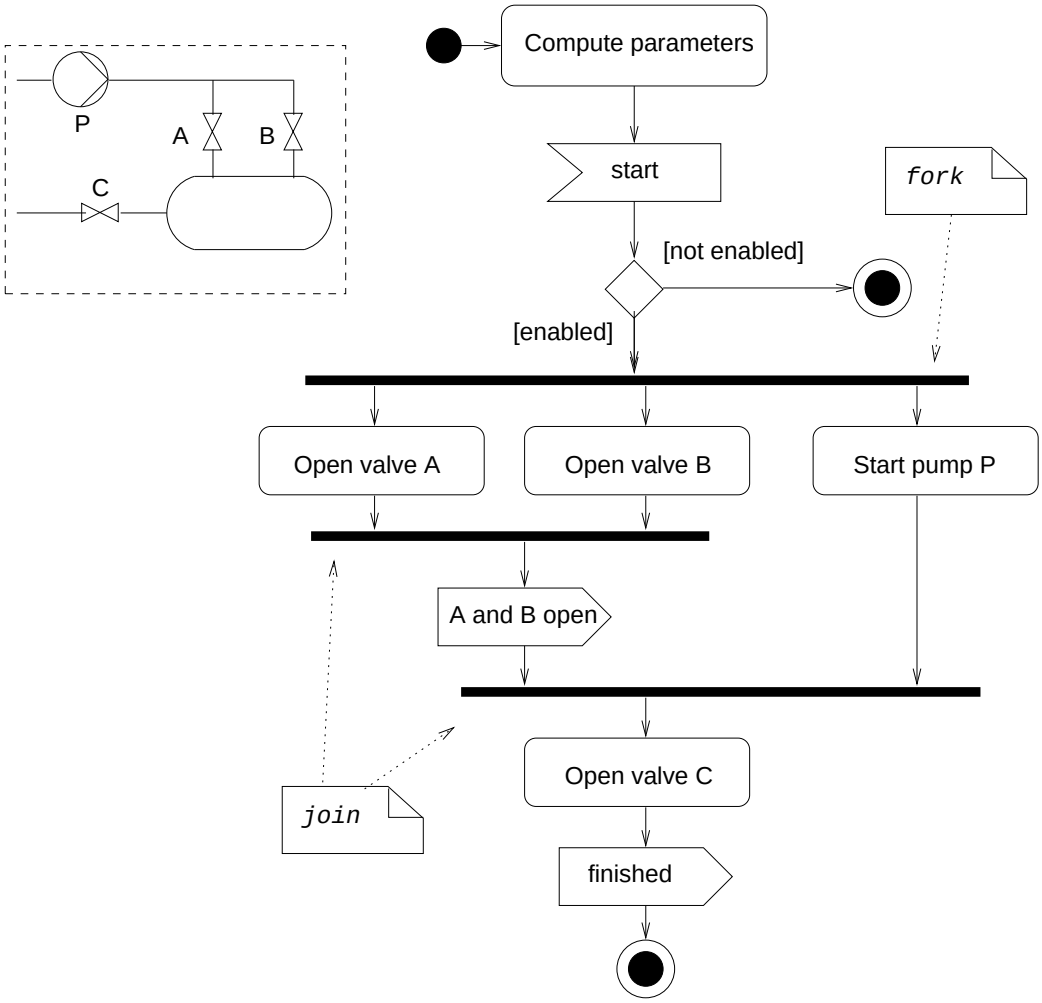

Figura 3.30: Un diagramma di attività.

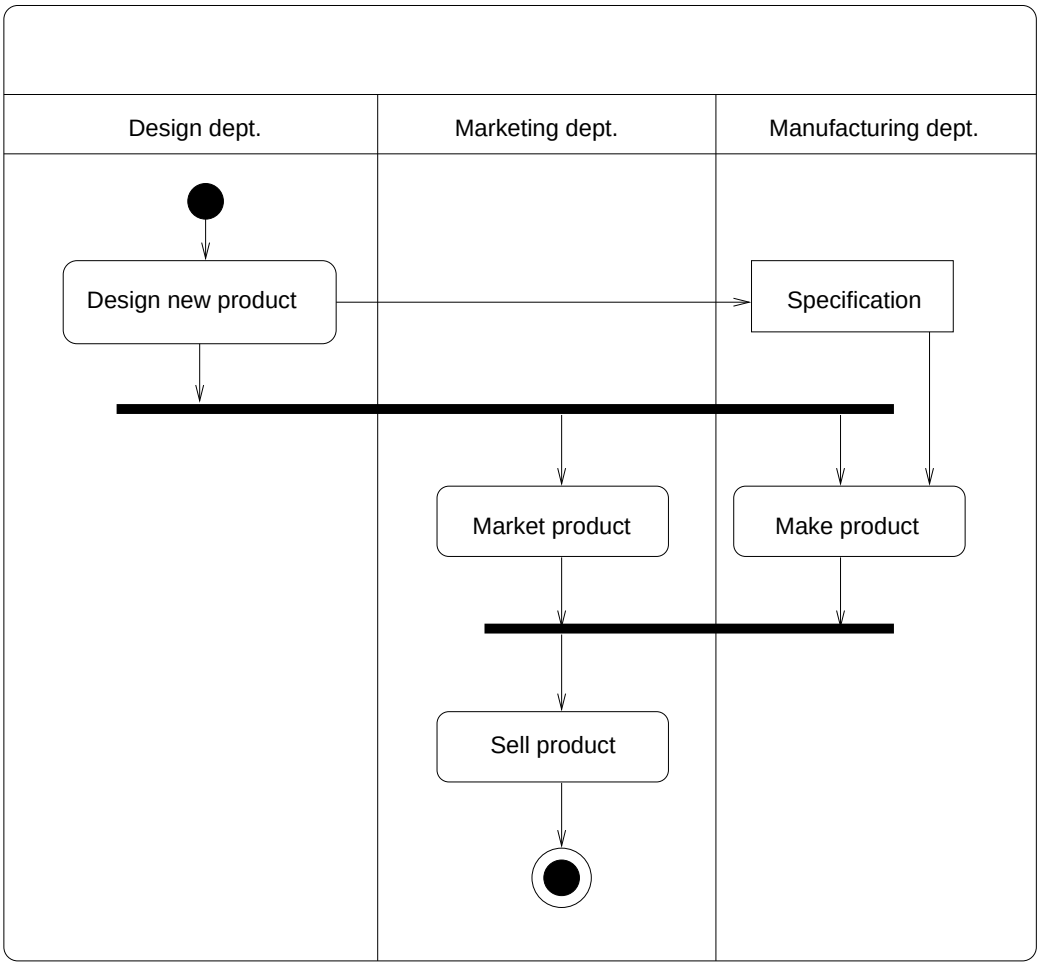

Figura 3.31: Corsie e nodi oggetto.