ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

CORSO DI LAUREA SPECIALISTICA IN INGEGNERIA INFORMATICA

DIPARTIMENTO DI INFORMATICA SCIENZA E INGEGNERIA

TESI DI LAUREA SPECIALISTICA

in
Linguaggi e Modelli Computazionali

SPECIFICA E SINTESI AUTOMATICA DI SISTEMI SOFTWARE: UNA NOTAZIONE IN LINGUAGGIO Z

CANDIDATO:

Fabio De Simone

RELATORE:

Chiar.mo Prof. Enrico Denti

Anno Accademico 2013/14

Sessione III

Specifica e sintesi automatica di sistemi software: una notazione in linguaggio Z

Indice

1
Introduzione

1.1 La Natura del Software

La costruzione di un sistema di elaborazione delle informazioni incorporato all'interno di un computer (ossia il *software*), si è dimostrata un'attività complessa che richiede un notevole sforzo creativo. A partire da un insieme di requisti circa le funzionalità del prodotto da costruire, occorre colmare un divario fra quanto richiesto e quanto attualmente disponibile nelle attuali tecnologie informatiche. A differenza di altre tipologie di prodotti, il software è una ben più nuova esigenza di costruzione. Ciò significa che quando si prova a costruirlo, non si può beneficiare della vasta esperienza nella costruzione di strumenti di simile tipologia, che è disponibile in altre branche dell'ingegneria. In aggiunta a questo, le particolari proprietà del software rendono l'impresa ben più ardua. Il fatto che il software sia totalmente privo di massa e sembri apparentemente economico e facile da modificare, concorrono a causare la produzione di sistemi scritti con poca o nessuna attenzione a una qualche teoria sottostante. Il progetto e la costruzione sono, apparentemente, due fasi non nettamente distinte nel software. La apparente semplicità nel modificare una linea di codice si rivela spesso problematica, data la mancanza nel sistema di una architettura, o nei casi in cui essa sia presente, concorre a erodere la sua qualità col passare del tempo. Per questo motivo il software non si usura a causa dell'uso bensì a causa della sua manutenzione. Questa flessibilità porta inoltre spesso a rilasciare prodotti incompleti o scorretti, poiché si confida di poter “mettere a posto” in un secondo momento le cose. La presenza di standard laschi e strumenti tutt'altro che efficaci nel supportare la vera e propria costruzione completano il quadro. Per finire, controllare la correttezza di quanto costruito rappresenta un ulteriore problema. L'industria del software, allo stato attuale, dispone esclusivamente del codice di test come strumento di verifica, ma produrre tale codice è un'attività estremamente lunga e costosa ed attraverso di esso si possono rivelare fault ma non dimostrarne l'assenza[1].

1.2 L'attuale processo di sviluppo del software

Attualmente la costruzione del software è portata a termine secondo due differenti “filosofie”. La prima (detta *Model Based* [19]), ritiene conveniente affrontare la costruzione del sistema attraverso una serie di modelli, a livelli di astrazione via via decrescente, che fungano da riferimento e supportino tutte le fasi della produzione.
La seconda (*Extreme Programming* [20]) parte invece dall'assunto che il miglior modello di un sistema software rimane comunque il codice. La progettazione è vista come parte integrante del processo di programmazione: il progetto si modifica al modificarsi del codice e occorre utilizzare metodologie che abilitano e sfruttano questo fatto, garantendo pieno controllo sui cambiamenti e sui rischi.
Come è facile immaginare la seconda filosofia mostra serie difficoltà durante lo sviluppo di sistemi complessi, mentre la prima può incontrarle al variare delle pretese sul sistema. Questo poiché i modelli, costruiti in base a determinate richieste, ora cambiate, necessitano di essere aggiornati e possono disallinearsi con quanto costruito sinora, non descrivendo più correttamente il sistema sottostante. La apparente maggior capacità di gestire dei requisiti variabili e una maggiore informalità, induce i produttori di software a ricorrere spesso al secondo approccio, andando pienamente incontro a tutte le problematiche esposte sopra. Chi invece prova ad adottare la prima filosofia si trova attualmente in grande difficoltà vista la attuale mancanza di strumenti non ambigui per la costruzione dei modelli, che risultano spesso troppo poco precisi o in alcuni casi addirittura devianti per la costruzione. Nell'eventualità di una variazione nei requisiti, il tempo speso nella costruzione di tali “modelli” risulta inoltre buttato a causa della necessità di doverli aggiornare. In tale ambito si colloca l'approccio *MDSD* (*Model Driven Software Development*) [13], che si propone di concentrarsi sui modelli per definire il sistema, generando poi automaticamente da essi il codice che rappresenta il prodotto finale. Questo consente al variare dei requisiti di aggiornare esclusivamente i modelli, generando la nuova codifica del sistema analogamente a come un compilatore produce codice macchina da un linguaggio di alto livello. Il modello diviene dunque il “codice sorgente”. Tale approccio risulta ad oggi inapplicabile, proprio per la mancanza di uno strumento non ambiguo, che consenta di definire modelli precisi del sistema in casi di reale complessità.
La costituzione di un tale strumento è proprio l'obiettivo di questa tesi.
Forse, dopotutto, la filosofia migliore consiste nell'*utilizzare modelli come codice sorgente*

all'interno di un approccio come quello dell'Extreme Programming, in modo da avere maggior controllo possibile su rischi e cambiamenti e minori costi possibili grazie alla capacità di generare automaticamente un sistema funzionante.

1.3 Obiettivi della Tesi

Nell'approccio Model Based i modelli sono equiparabili a mera documentazione, poiché il legame fra essi e il codice che implementa il sistema è solo intensionale, non formale.
D'altro canto l'approccio MDSD prevede la presenza di un meta-modello [13] (cioè un linguaggio), il quale fornisca lo spazio concettuale necessario ad esprimere i modelli del sistema. Una volta noto il modello del sistema, che dunque risulta un'istanza del meta-modello, ci si propone di elaborarlo tramite trasformatori *model to code* [13] (M2C), o *model to model* [13] (M2M). I primi danno la semantica operazionale del meta-modello in termini di codice. I secondi la danno in termini dei costrutti di un linguaggio di più basso livello. Questo rende le metafore del meta-modello fortemente dipendenti dai trasformatori. La modifica di questi ultimi (ad esempio per passare da una tecnologia ad un'altra), incrina le astrazioni del meta-modello, che dunque finisce per non avere alcuna semantica precisa.
Questo è in gran parte dovuto alla mancanza di uno strumento per definire le astrazioni in modo chiaro e indipendente da trasformazioni successive.

L'obiettivo della tesi è dunque quello di *costruire un nuovo linguaggio di modellazione* che abbia le seguenti proprietà:

- possieda una semantica precisa e indipendente
- supporti tutte le fasi del processo di sviluppo
- guidi nella definizione delle astrazioni

Si vuole dunque costruire un nuovo linguaggio che consenta di esprimere interi sistemi software come sue frasi, capace di supportare l'intero processo di sviluppo, fino alla generazione automatica di codice. Il linguaggio dovrà quindi essere general purpose, dovrà rendere possibile la creazione di modelli a granularità crescente del sistema e dovrà

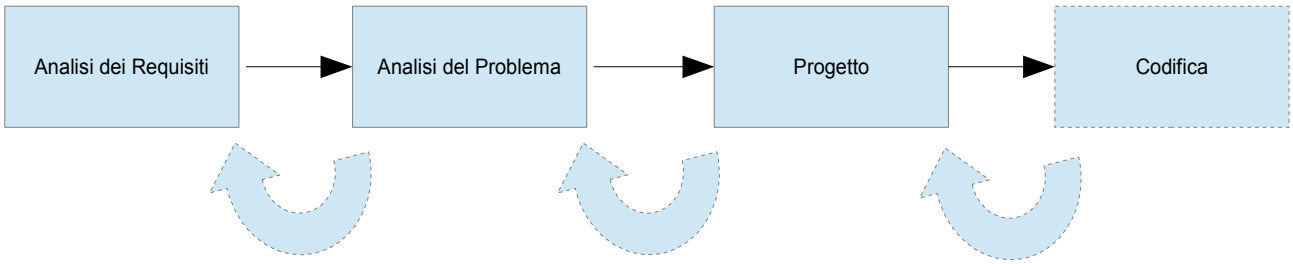
offrire una metodologia per la definizione delle astrazioni.

Una volta definite tali astrazioni dovrà essere possibile progettarle. In modo che il sistema porti a termine quanto gli è richiesto nel più conveniente dei modi. Infine, terminato il progetto, si vogliono utilizzare tali modelli come codice sorgente, automatizzando la codifica.

Per farlo si svilupperà una notazione con costrutti adatti alle varie fasi della costruzione. Tale notazione sarà dotata di una chiara semantica formale, che renda indipendenti i concetti definiti da trasformazioni successive. Consentirà inoltre la specifica di quanto desiderato ad un livello più elevato di quello implementativo. Si costruirà infine una piattaforma, la quale attribuirà una semantica operazionale alla notazione, consistente con la sua semantica formale. Tale piattaforma consentirà la traduzione dei modelli espressi nella notazione, verso un ambiente esecutivo.

2
Fondamenti del Processo di Sviluppo del Software

2.1 Le Fasi del Processo di Sviluppo del software



Si propone il seguente processo di sviluppo, che verrà preso a riferimento nella parte restante della tesi. Si noti che il processo proposto, pur non discostandosi molto nei suoi ingredienti fondamentali, da quanto individuato attualmente in modo generale dall'Ingegneria del software, precisa alcuni dettagli tutt'altro che secondari.

Analisi dei Requisiti: Si tratta della formalizzazione delle funzionalità richieste al sistema. Le pretese sul sistema concordate assieme al cliente vengono organizzate in un modello (*modello dei requisiti*) che ha il duplice ruolo di riassumerle e chiarirle. In tal modo è possibile identificare richieste formulate in modo ambiguo a causa del linguaggio naturale e approfondire con precisione l'entità e la vera natura di ciò che si richiede al sistema. L'impatto e la complessità che ogni requisito rappresenta, così come la distinzione fra quelli che sono requisiti funzionali del sistema da pretese secondarie (o totalmente irrilevanti) dal punto di vista tecnico devono divenire esplicite grazie a questo primo modello del sistema.

Analisi del Problema: Noto il modello dei requisiti bisogna ora definire *cosa* occorre nel sistema affinché esso sia in grado di soddisfare i requisiti. Avendo come riferimento il modello dei requisiti, viene costruito un secondo modello del sistema (*modello di analisi*), in cui si specifica attraverso *astrazioni architetturali*, l'insieme di macro-parti adeguate alle

problematiche che il sistema dovrà affrontare. Per *architettura* si intende un insieme di regole e schemi che riducono i gradi di libertà nella costruzione di un sistema. Alla definizione di una architettura, segue poi il raffinamento di ciascuna delle macro-parti individuate nelle sue sotto-parti costituenti, tramite *astrazioni di blocco*.
A questo punto il modello di analisi dovrebbe comprende tutto ciò che è necessario nel sistema, specificando la sua *struttura*, l'*interazione* fra le parti (blocchi) e il *comportamento* di ciascuna parte (blocco), senza però esplicitare i dettagli relativi a come ciascun blocco realizza il comportamento richiesto. Si noti che l'interazione fra due sotto-parti potrebbe essere distribuita e che la natura di tali interazioni e le interazioni fra i blocchi di una sottoparte o fra blocchi di sotto-parti differenti dovrebbero essere precisate già a questo livello.
In questa fase è possibile anche la costituzione di un modello di strutture e comportamenti comuni a tutti i sistemi di una certa area applicativa (*modello del dominio*). Tale modello conterrà blocchi individuati come generali e riusabili per altri sistemi nella stessa area applicativa.

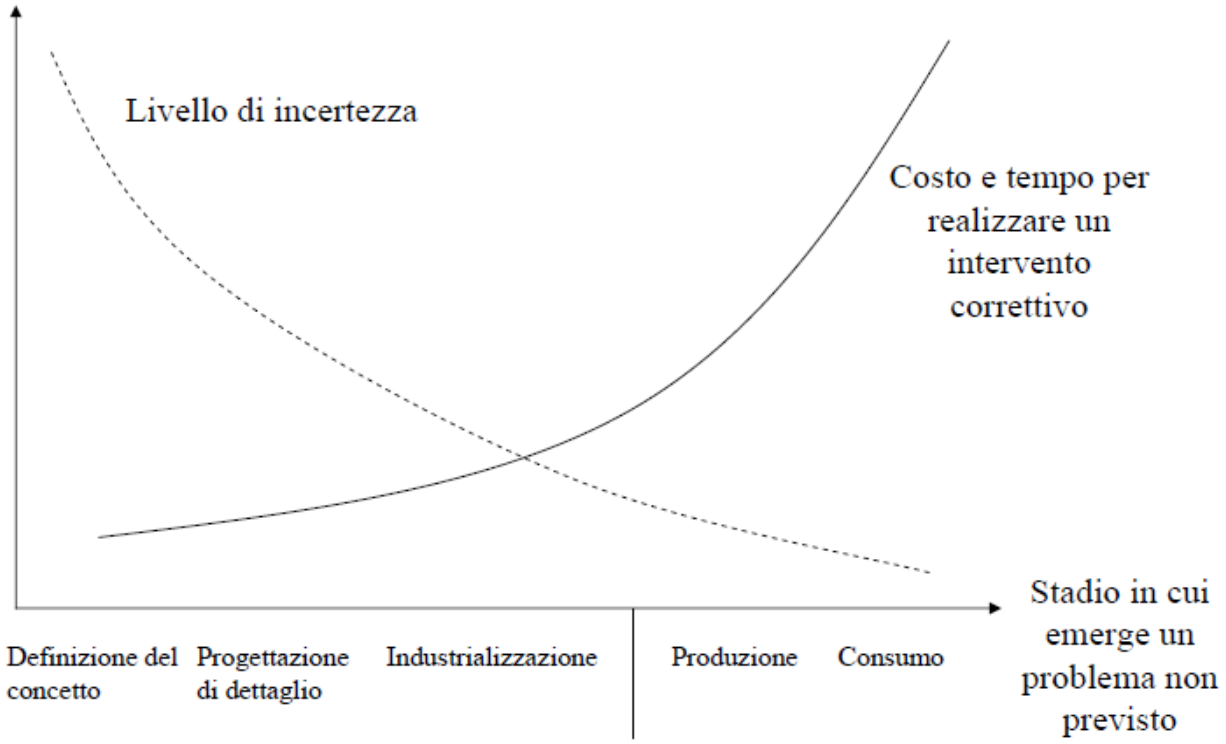
Progetto: Definito cosa occorre nel sistema, grazie al modello di analisi, occorre stabilire *come* farlo, in accordo ai vincoli posti dalle specifiche di analisi. In particolare ciascuno dei blocchi specificato in fase di analisi viene decomposto in un sotto-sistema, di cui si esplicita l'interfaccia concreta e la struttura interna che realizza il comportamento richiesto. In questa fase, oltre ad esplicitare la rappresentazione concreta di stato dei pezzi che compongono ciascun sottosistema e la loro interazione all'interno di esso, occorre specificare i singoli passi che ciascuno dei pezzi intraprende. In questo modo viene a costituirsi un *modello di progetto*, il quale rappresenta un raffinamento dell'architettura individuata in fase di analisi in una forma più articolata, che preserva i vincoli stabiliti al livello precedente. Un ruolo importante in questa fase è l'uso dei pattern di progettazione [21], che promuovono una motivazione sistematica delle scelte che progressivamente portano il raffinamento del sistema fino alla sua forma finale. La soluzione al problema posto in fase di analisi deve scaturire come punto di equilibrio fra le forze in gioco (trade-off). Si noti che un sottosistema è qui inteso come un'entità concentrata.
Analogamente, se richiesto, in questa fase anche i blocchi del modello del dominio trovano la loro concretizzazione.

Codifica: Completato il modello di progetto è possibile a questo punto generare in modo

automatizzato il codice completo del sistema corrispondente.

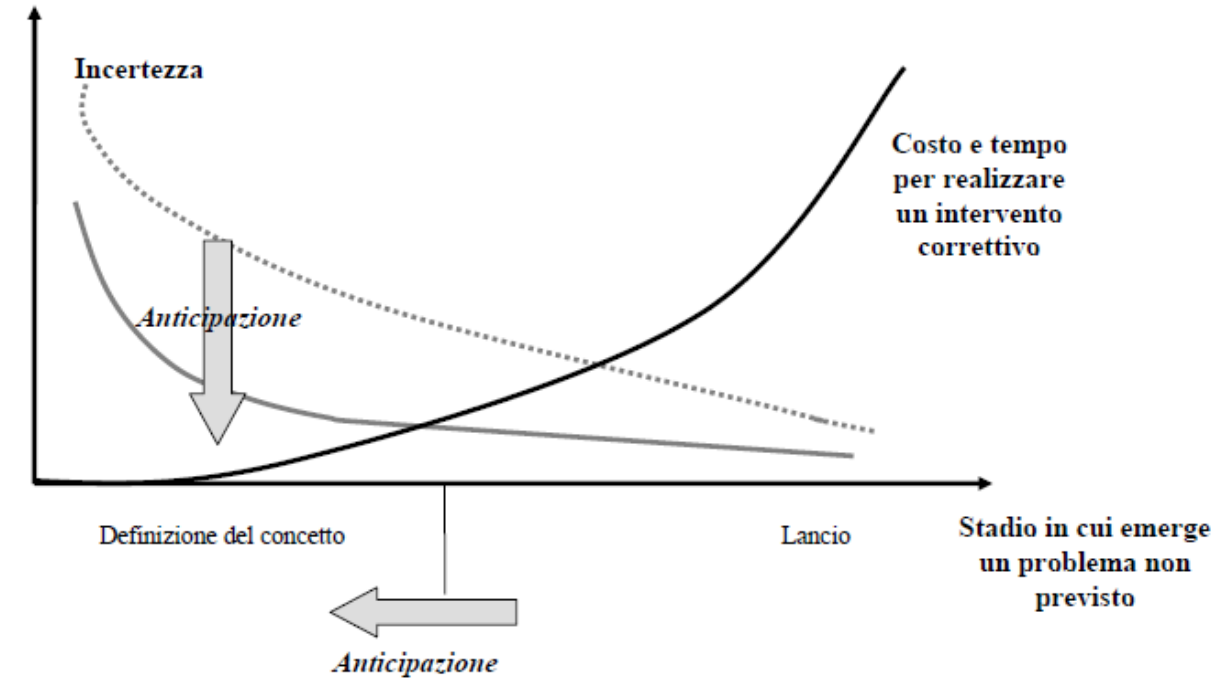
Si noti che ciascuna fase dipende inevitabilmente da quella precedente e non può esistere senza di essa. Poiché la correttezza assoluta è una parola senza significato, la fase successiva può portare alla luce mancanze in quella precedente e generare feedback e correzioni su di essa in uno stile tipicamente iterativo, fino alla convergenza verso un modello finale di progetto.

2.2 L'importanza dell'Analisi

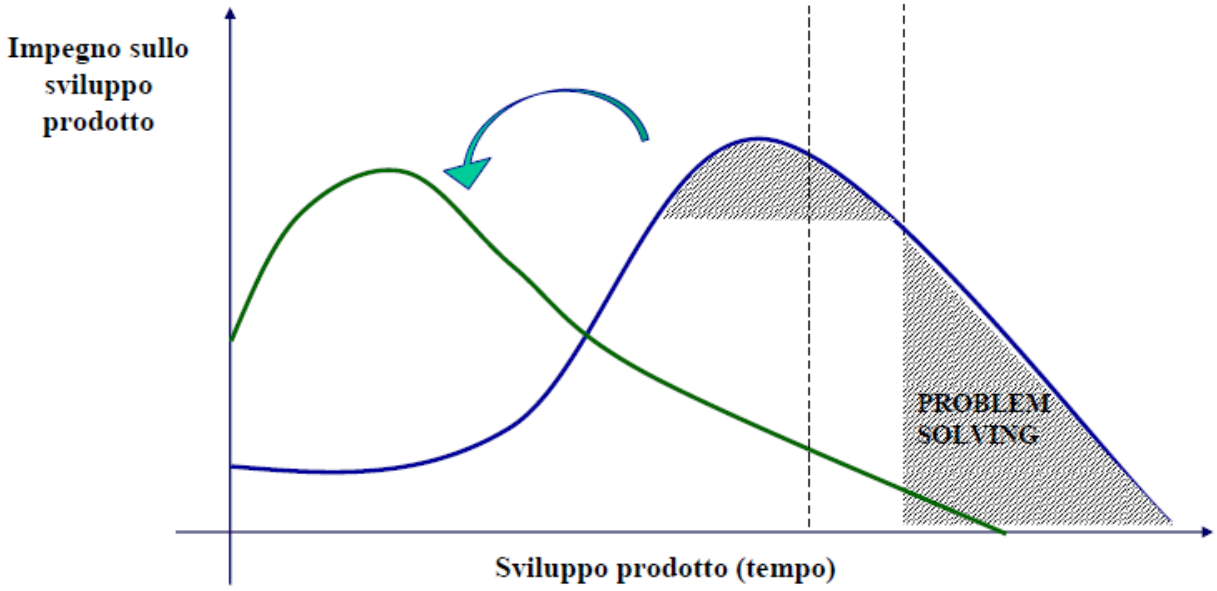


L'esperienza sinora maturata nella costruzione del software ha portato alla luce che la maggior parte dei costi sostenuti durante un progetto è dovuta a correzioni necessarie nelle fasi più tarde dello sviluppo e della produzione. Queste correzioni effettuate in fondo al processo di sviluppo sono molto costose poiché molto lavoro è già stato fatto e diversi pezzi del sistema sono già concretizzati. La maggior parte di queste correzioni si è rivelata dovuta ad errori o imprecisioni nelle prime fasi dello sviluppo. Questo avviene tipicamente a causa di indicazioni troppo lasche o per la mancanza della precisazione di dettagli essenziali in fase di specifica. La conseguenza di ciò sono errate decisioni di progetto nelle fasi successive, la cui correzione, una volta a valle, diviene molto costosa a causa

dell'ampio lavoro già svolto o in corso.



La combinazione dei costi crescenti della correzione e l'origine della maggior parte degli errori nelle prime fasi del processo di sviluppo, portano alla convinzione che maggiori sforzi nelle prime fasi dello sviluppo e l'utilizzo in esse di tecniche di costruzione più potenti, possa portare a una sostanziale riduzione degli errori e dunque dei costi da sostenere durante lo sviluppo.



L'analisi, ponendo vincoli sul progetto ed anticipando una definizione il più possibile precisa degli ingredienti fondamentali richiesti dal sistema, riduce notevolmente l'incertezza presente nel sistema da realizzare. Un modello di analisi ben fatto è dunque in grado di abbattere notevolmente i costi e *rappresenta il vero valore aggiunto di un progetto*.

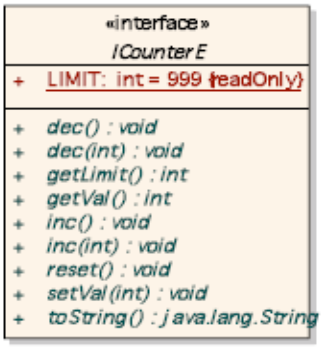
2.3 Limiti degli attuali strumenti

Nonostante quanto esposto sopra, oggi gli analisti (e anche i progettisti) di sistemi software, si trovano nella frequente impossibilità di formulare modelli precisi di un sistema complesso, a causa della inadeguatezza degli strumenti a disposizione per esprimerli. Questo causa in gran parte la frequente creazione di modelli incompleti (e quindi problematici) o la loro completa omissione.
Attualmente sono disponibili diversi linguaggi di modellazione, molti di essi grafici come lo UML[2], i quali tipicamente approcciano il problema utilizzando il principio della decomposizione gerarchica in parti più semplici. Tipicamente tali linguaggi consentono di catturare relazioni statiche fra i dati su cui occorre operare e di connotare i flussi di informazione all'interno del sistema. Strutturalmente è ormai affermata una stretta relazione fra i dati e le operazioni da effettuare su di essi nel paradigma object-oriented. Quelle di cui si parla sono tutte proprietà strutturali. Non appena si scende ad una granularità più fine cercando di specificare l'interazione fra i blocchi ed il loro comportamento, si è costretti a ricorrere al linguaggio naturale, ambiguo per sua natura, o a notazioni più o meno ispirate a diagrammi per automi a stati finiti, i quali divengono rapidamente ingestibili all'aumentare della complessità, soprattutto se lo spazio degli stati del componente da specificare è (teoricamente) infinito.
Potrebbe allora venire in mente di utilizzare i linguaggi di programmazione per la specifica dei blocchi, tuttavia tali linguaggi sono studiati per definire la sequenza di istruzioni che un elaboratore dovrà svolgere, e dunque avendo una granularità così fine, costringono a dire “come” un certo blocco fa una cosa in luogo del “cosa” un certo blocco dovrebbe fare. Appare quindi evidente come i linguaggi di programmazione siano inadatti ad esprimere il comportamento atteso di un blocco come si necessita nell'analisi. Ciò di cui si necessita è un *linguaggio di specifica*, ossia di un veicolo che consenta di esprimere concetti in modo preciso, senza però obbligare a una granularità così fine come quella dei linguaggi di

programmazione, che non potrebbe risultare che sviante in fase di analisi. Si vorrebbe inoltre poter essere in grado di condurre ragionamenti rigorosi sui modelli, in modo da poter affermare se il sistema possieda o meno una data proprietà. La capacità di essere precisi e di poter condurre ragionamenti rigorosi è ciò che si intende quando si connota un linguaggio con la parola *formale*. Essere in grado di esprimere ciò che è richiesto senza dover specificare come ottenerlo è ciò che si intende per *astrazione*.
Una *implementazione* è dunque la specifica di come realizzare una astrazione. Una implementazione *corretta* è una qualunque implementazione che soddisfa la specifica dell'astrazione.
Dunque occorre un linguaggio di specifica formale che consenta di esprimere efficacemente l'astrazione di blocco, che rappresenta il mattone fondamentale nella costruzione del sistema.
Nel seguito ci si concentrerà dunque su di esso.

2.4 Verso un nuovo linguaggio di modellazione

Come si è detto ciò che occorre è essere in grado di modellare efficacemente i blocchi componenti una sotto-parte del sistema.
L'intera disciplina dell'Ingegneria del Software è basata sul *Principio di Inversione delle Dipendenze* [24]: *occorre dipendere dalle astrazioni, non dalle concretizzazioni. Devono essere le implementazioni a conformarsi alla specifica delle astrazioni da realizzare e non viceversa*
Ciò implica essere in grado di modellare astrazioni ad un livello più elevato rispetto a quello implementativo.
Una astrazione rappresenta tipicamente un *concetto* usualmente introdotto per decomporre ed affrontare problematiche in un certo dominio applicativo.
Per modellare un blocco occorre definire una *interfaccia*, ossia identificare un insieme di *eventi* che possono avvenire a tale blocco, indicando per ciascuno di essi l'eventuale *informazione necessaria da produrre in input* e l'eventuale *informazione restituita in output* al verificarsi di un evento.
Solitamente si esprime tutto ciò attraverso un insieme di *signature*

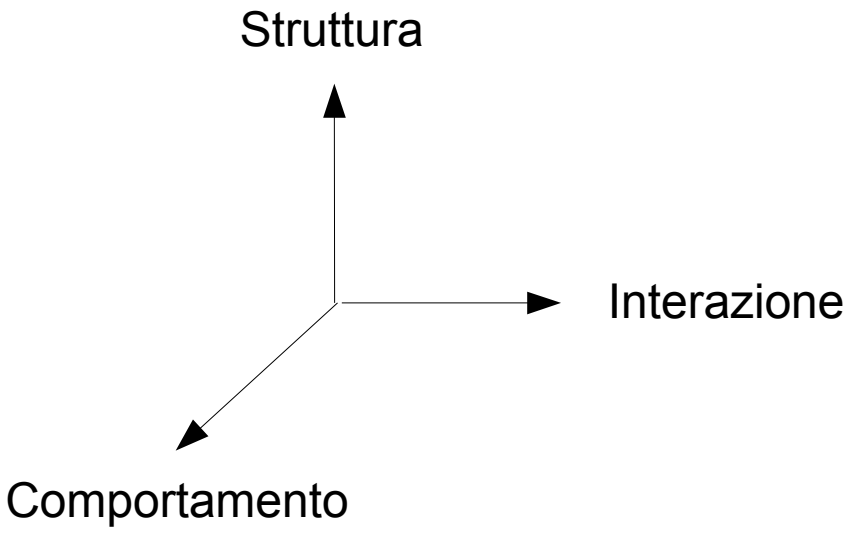


Un insieme di signature da solo non costituisce un'astrazione. Occorre definire una specifica formale del comportamento atteso osservabile dall'esterno. Se tale specifica manca, il principio di inversione delle dipendenze è violato e l'ingegneria del software perde il suo fondamento e perde di significato.

Il linguaggio obiettivo dovrà dunque essere in grado di definire concetti e fornire la possibilità di definire in modo non ambiguo tale specifica.

Si vuole essere in grado, una volta giunti in fondo alla fase di progetto, di generare automaticamente il codice completo del sistema. Ciò implica che il linguaggio obiettivo dovrà essere in grado di esprimere le computazioni. Il concetto di computazione può essere definito individuando le mosse elementari di un automa esecutore, ossia di un sistema formale. Ad esempio uno degli automi esecutori preso più di frequente a riferimento è la macchina di Turing, per cui computare corrisponde a cambiare di stato, leggere/scrivere su un nastro, spostarsi su tale nastro.

Se dunque un blocco è dotato di uno *stato*, i *cambiamenti di stato* possono aiutarci a rappresentare le computazioni in modo astratto. Questo consente di modellare il *comportamento* di un blocco, tuttavia per giungere ad un modello computazionale completo, occorre definire i concetti che il linguaggio mette a disposizione per esprimere la *struttura* di un blocco e l'*interazione* (anche distribuita) fra blocchi.

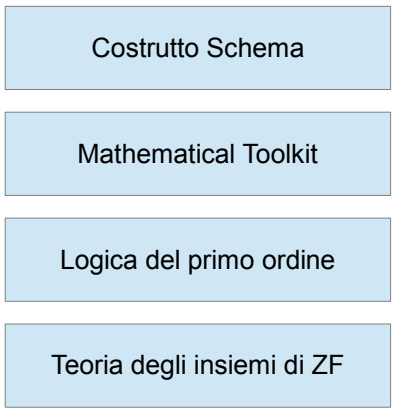


Si vuole inoltre poter essere in grado di specificare entità *attive* oltre che *passive*.

3
La Notazione Z

3.1 Introduzione al Linguaggio Z

Z è un linguaggio per esprimere *teorie matematiche strutturate* [3]. E' basato sulla teoria degli insiemi di Zermelo e Fraenkel (ZF) secondo cui “tutto è un insieme”. Alle nozioni di base della teoria degli insiemi di ZF il linguaggio Z aggiunge l'idea che gli insiemi del suo universo possiedano un *tipo*. Non può dunque esservi intersezione fra due insiemi di tipo diverso. Anche i tipi sono insiemi.
Sono presenti gli usuali costrutti e operatori della logica del primo ordine e addizionalmente i concetti di base della matematica, come l'idea di relazione o di funzione.
Solitamente una specifica scritta in Z è un insieme di proposizioni formali e matematiche, inframezzate con testo (opzionale) scritto il linguaggio naturale. Le parti formali danno una descrizione precisa del sistema da costruire, mentre i commenti in linguaggio naturale rendono la specifica più leggibile e collegano i costrutti matematici al mondo reale.
(Le specifiche degli esempi seguenti e quelle presenti nel resto della tesi sono stati creati tramite il tool CZT [5]).



I tipi primitivi in Z vengono introdotti nel seguente modo:

— [NAME , DATE] ⌙

Si stanno in tal modo connotando l'universo dei nomi e l'universo delle date. Si noti che così facendo è possibile astrarre (se lo si desidera), da dettagli irrilevanti al fine della specifica, come ad esempio la struttura di una data o la rappresentazione di un nome.

{ 1 , 2 , 3 } (x , y , z) ⟨ a , b , c ⟩

denotano rispettivamente l'insieme contenente i valori 1, 2, 3 , la n-upla contenente i valori x,y,z e la sequenza contente i valori a,b,c .

ℙ , ×

denotano rispettivamente l'insieme potenza e il prodotto cartesiano

↔ , ⇸ , seq

sono i simboli di relazione, funzione e sequenza

∧ , ∨ , ¬ , ⇒ , ⇔ , = , ≠

Denotano gli usuali operatori della logica del primo ordine

∀ x | p • e

Denota “per ogni x per cui è vero p allora (⇒) è vero e”

∃ x | p • e

Denota “esiste un x per cui sono veri p ed (∧) e”

{ variables | predicate }

Denota l'insieme dei possibili assegnamenti alla n-upla di valori variables che soddisfa il predicato predicate

{ variables | predicate • expression }

Denota l'insieme di valori presi da expression quando le sue variabili variables assumono tutti i possibili valori che soddisfano predicate

Le ultime due forme sintattiche sono dette *set comprehension* in Z.

```
┌
  limit : ℕ
|
  limit ≤ 65535
└
```

Denota una definizione assiomatica, ossia la definizione di un simbolo statico e delle proprietà di cui gode che può essere riferito da quel punto in avanti nel resto della specifica utilizzando il suo nome

```
┌═[ X , Y ]
  first : X × Y ⟶ X
|
  ∀ x : X ; y : Y • first(x,y) = x
└
```

Denota un'altra definizione assiomatica che però definisce dei tipi generici

Un secondo concetto molto importante introdotto da Z è lo *schema*. Questo costrutto consente di specificare una lista di attributi, ossia di coppie nome-valore. Il valore di ciascun attributo appartenente ad un ben determinato insieme. Contestualmente consente di asserire proposizioni su questa lista di attributi. A tale lista di attributi e alle proposizioni asserite su di essi viene attribuito un nome unico che consente di riferire lo schema all'interno della specifica.
Si noti che anche uno schema è un insieme, ossia l'insieme di tutti i possibili assegnamenti di valori alle sue variabili (*binding*) che soddisfano le proposizioni asserite nello schema. In conseguenza di ciò uno schema può rappresentare un tipo ed essere utilizzato per costruirne di nuovi partendo da quelli primitivi o da altri schema già definiti.

```
┌─ BirthdayBook
 known : ℙ NAME
 birthday : NAME ⇸ DATE
|
 known = dom birthday
└─
```

Definisce lo schema BirthdayBook contenente gli attributi known e birthday. Tali attributi sono rispettivamente un insieme di nomi e una funzione parziale dall'universo dei nomi all'universo delle date. Nello schema viene inoltre detto che il dominio della funzione birthday è sempre l'insieme known. Un predicato di questo tipo va sotto il nome di *invariante*.

Lo schema è dunque il mattone fondamentale offerto da Z per strutturare le nostre specifiche. E' possibile specificare eventi (anche detti operazioni) che avvengono ad uno schema nel seguente modo:

```
┌─ AddBirthday
 ΔBirthdayBook
 name? : NAME
 date? : DATE
|
 name? ∉ known
 birthday′ = birthday ∪ { name? ↦ date? }
└─
```

E' un altro schema e denota un evento di nome AddBirthday. Tale evento dato un nome ed una data forniti in ingresso, sotto opportune condizioni produce (in modo atomico) un nuovo stato per uno schema di tipo BirthdayBook. Più precisamente dato lo stato iniziale di un BirthdayBook, ed un nome ed una data forniti in ingresso (decorazione ?), produce uno stato finale (cioè un nuovo assegnamento di variabili) per uno schema di tipo BirthdayBook. Le variabili dello stato finale sono distinte da quelle dello stato iniziale

tramite la decorazione ′ .

Questa volta vi sono due predicati sotto la lista di attributi. Elencare predicati in questo modo equivale a esprimerne la loro congiunzione tramite l'operatore ∧.

Il primo predicato afferma che lo schema AddBirthday può essere verificato solo se la variabile name? Non è già contenuta in known. Un predicato di questo tipo che esprima vincoli sulle variabili in ingresso o sullo stato iniziale è detto *precondizione*.

Il secondo predicato afferma che nello stato finale la funzione birthday conterrà i vecchi mapping più una nuova voce. Un predicato di questo tipo che esprima vincoli sulle variabili di output o sullo stato finale prende il nome di *postcondizione*.

Si noti che si sta semplicemente stabilendo che dato uno stato iniziale, se avviene un certo evento, al termine di esso lo stato finale sarà in un certo modo. Dunque *si sta affermando cosa avviene senza dire come avviene*. L'uso di precondizioni e postcondizioni è la chiave per specificare sistemi nel modo desiderato.

Lo schema AddBirthday definisce dunque una relazione tra uno stato iniziale “prima” del verificarsi dell'evento e uno stato finale “dopo” il verificarsi dell'evento. Più precisamente si tratta di una relazione tra un insieme di assegnamenti di tipo BirthdayBook e variabili di ingresso e un nuovo assegnamento dello stesso tipo (e nessuna variabile di uscita).

Si noti che non avviene nessuna modifica al binding iniziale: *la semantica è per valore*. Dunque data la definizione di uno schema, di un evento su tale schema e di un binding iniziale per tale schema che soddisfa le precondizioni dell'evento, è possibile inferire un nuovo binding (ossia un nuovo valore) che soddisfa le postcondizioni espresse dall'evento.

Si possono analogamente denotare eventi che non modificano lo stato di uno schema (Ξ in luogo di Δ)e variabili di uscita (decorazione !):

```
┌─ FindBirthday
 Ξ BirthdayBook
 name? : NAME
 date! : DATE
|
```

```
 name? ∈ known
 date! = birthday ( name? )
└
```

Uno stato iniziale legale per uno schema può essere espresso nel seguente modo:

```
┌ InitBirthdayBook
 BirthdayBook'
|
 known' = { }
└
```

Si noti che tale schema *include* lo schema BirthdayBook e ne definisce una signature le cui variabili avranno tutte la decorazione ' . L'inclusione fra schema è il meccanismo grazie al quale possiamo definire gli eventi di modifica e di query. Quando includiamo uno schema in un altro schema, il secondo avrà tutte le variabili del primo e i predicati del primo saranno aggiunti in congiunzione ai predicati espressi nel secondo. E' possibile quindi dare una definizione degli schemi Δ e Ξ nel seguente modo:

```
┌ ΔState
  State
  State'
└
```

```
┌ ΞState
  State
  State'
|
  θ State' = θ State
└
```

dove l'operatore θ denota un binding di tipo pari al nome dello schema che lo segue (che essendo citato nella parte predicativa dello schema dovrà essere stato definito o incluso nella parte dichiarativa.

E' possibile definire uno schema componendo altri schema attraverso i connettivi della logica proposizionale (*schema calculus*). Lo schema risultante avrà tutte le componenti degli schema di partenza. Eventuali componenti con lo stesso nome devono avere lo stesso tipo e vengono unificate.

```
┌─ Success
 result! : REPORT
|
 result! = ok
└─
```

```
┌─ AlreadyKnown
 Ξ BirthdayBook
 name? : NAME
 result! : REPORT
|
 name? ∈ known
 result! = already_known
└─
```

```
— RAddBirthday == ( AddBirthday ∧ Success ) ∨ AlreadyKnown
```

Le specifiche saranno ospitate in *sezioni* che potranno includere le affermazioni di altre sezioni definite in precedenza:

— section spec parents standard_toolkit
⌙

Si noti che uno schema una volta definito, può essere utilizzato nelle frasi del linguaggio Z come *dichiarazione*, come *predicato* e come *espressione*.

Schema come dichiarazione: introduce le variabili definite nello schema in una dichiarazione e le vincola secondo quando asserito nella sua parte predicativa.

Schema come predicato: nella parte dichiarativa della frase devono essere presenti tutte le variabili indicate nello schema, con nome e tipo analoghi. Il predicato schema sarà vero se i valori attribuiti alle variabili soddisfano i predicati espressi nello schema e falso altrimenti.

Schema come espressione: il valore di uno schema usato come espressione è l'insieme di assegnamenti alle sue variabili che soddisfano i predicati espressi nello schema (ossia l'insieme denotato dallo schema stesso).

Per finire, in Z sono presenti teoremi che consentono di affermare se una specifica **raffina** o meno un'altra specifica.
Informalmente, affinché un'entità R sia raffinata da S occorre che siano soddisfatte le seguenti condizioni:

Condizione di applicabilità

- quando R è applicabile allora lo è anche S

Condizione di correttezza

- quando R è applicabile ma viene applicato S, allora il risultato è consistente con R

Si ricorda che in Z, qualunque cosa non venga espressamente asserita, resta

parzialmente specificata e dunque rappresenta un grado di libertà per la sottostante implementazione.

In questo paragrafo si è data una breve introduzione al linguaggio Z, in quanto tale linguaggio, possedendo una chiara semantica formale, verrà utilizzato nel resto della tesi per descrivere in modo preciso i concetti quando necessario.
Per una più ampia trattazione del linguaggio Z è possibile consultare [3,4].

3.2 Z e l'analisi dei requisiti

Come si è visto, Z consente tramite astrazioni matematiche e pre e post condizioni, di specificare cosa debba esserci o debba avvenire in un sistema, senza necessariamente esplicitare quale sia la struttura concreta dei componenti e come essi portino a compimento il lavoro richiesto loro. Questo poiché si esprime un evento asserendo cosa debba essere vero affinché l'evento possa avvenire e cosa è vero dopo che un evento è avvenuto. La specifica non dice nulla riguardo a ciò che avviene durante il verificarsi dell'evento. Il fatto che tali eventi manipolino tipologie di dati espressi utilizzando le metafore della matematica completa il quadro.
Si è inoltre visto come tramite composizione di schema, sia possibile creare tipi di complessità arbitraria. Poiché in tal modo si è in grado di specificare cosa ci si aspetta da un sistema senza dire come esso lo faccia, è possibile rappresentare efficacemente in Z i requisiti senza dover andare troppo in dettaglio circa la struttura interna del sistema da costruire.
Il fatto che l'unico costrutto di strutturazione offerto sia un “record” , ossia un concetto di basso livello, non fa altro che giovare all'attività della specifica dei requisiti. L'obiettivo di tale fase è infatti catturare in modo preciso ed essenziale le richieste sul sistema, non fare asserzioni su come è fatto il sistema. In tal modo il focus attentivo dell'analista dei requisiti è sempre fisso sulle richieste. Se si avessero a disposizione costrutti di più alto livello come le classi, si potrebbe incorrere nell'errore di cercare di risolvere i requisiti condizionando la struttura interna del sistema. Queste false assunzioni sarebbero poi prese come fatti nella successiva fase di analisi, generando errori e cattive scelte di progetto a valanga. Un modello basato su classi induce infatti per sua natura una classificazione della realtà, che difficilmente sarà aderente ad essa, in quanto la realtà

riguarda la struttura del sistema da costruire e non i requisiti su di esso. Ciò che vogliamo fare è solo precisare le richieste, non ordinarle in una ontologia. Il fatto che alcuni requisiti possano essere stati formulati in modo ambiguo in linguaggio naturale rende una tassonomia derivata da essi estremamente sviante.
Dunque Z “standard” si rivela un ottimo strumento per la formalizzazione dei requisiti.

3.2.1 Un esempio di specifica dei requisiti

Si supponga di avere il seguente documento dei requisiti:

“Si vuole costruire un sistema formato di due pezzi, in cui il primo di essi, di nome CentralinaFissa, riceva dati da sensori sul consumo istantaneo di un insieme di elettrodomestici e debba stimarne per ciascuno l'andamento in base a tali osservazioni. (…)
Il secondo pezzo invece, di nome CentralinaMobile, impartisce comandi al primo per leggere il consumo degli apparecchi e modificare lo stato di accensione e spegnimento dei relativi interruttori (...).
La centralina fissa dovrà inoltre gestire automaticamente gli elettrodomestici evitando che i consumi complessivi superino una certa soglia. In particolare in base al loro consumo deve essere attuata una politica che spenga per primi gli elettrodomestici di consumo più alto e accesi per ultimi. Un elettrodomestico a basso consumo una volta acceso non può essere spento automaticamente. Qualora l'accensione di un elettrodomestico causi il superamento del valore di soglia, l'accensione va evitata a priori. Se la centralina non riceve per più di due secondi informazioni relative a uno degli apparecchi, spegne per sicurezza il relativo interruttore (...)”.

```
— section RequirementsConcepts parents standard_toolkit
└
```

Si potrebbe a tal fine definire l'universo degli elettrodomestici e l'universo degli interruttori:

— [INTERRUTTORE] ⌙

— [ELETTRODOMESTICO] ⌙

Nei requisiti vi è scritto che un interruttore può trovarsi in un due possibili stati, dunque si potrebbe definire a tal fine un tipo enumerativo per questa coppia di stati:

—

StatoInterruttore ::= ON | OFF

⌙

I requisiti affermano inoltre che i due pezzi del sistema dovranno essere in grado di comunicare. Tipicamente ci si aspetta che questa comunicazione avvenga in modo distribuito fra due sistemi distinti, dunque affinché possano riferirsi bisognerà assegnare loro degli indirizzi:

— [INDIRIZZO] ⌙

Si sa inoltre che la CentralinaMobile dovrà inviare messaggi alla Centralina fissa per modificare lo stato dell'interruttore di un certo apparecchio:

⌐ MessaggioRichiestaCentralinaMobile

idApparato : ELETTRODOMESTICO

richiesta : StatoInterruttore

destinazione : INDIRIZZO

⌙

in accordo alle usuali convenzioni di Z si specifica lo stato iniziale di quest'ultimo schema:

⌐ MessaggioRichiestaCentralinaMobileInit

```
  MessaggioRichiestaCentralinaMobile ’
  idApparato? : ELETTRODOMESTICO
  richiesta? : StatoInterruttore
  destinazione? : INDIRIZZO
|
  idApparato’ = idApparato?
  richiesta’ = richiesta?
  destinazione’ = destinazione?
└
```

Questo schema afferma che per poter inizializzare un messaggio occorre fornire in ingresso un identificativo di elettrodomestico, lo stato richiesto e la destinazione del messaggio.

La CentralinaFissa deve mantenere tutta una serie di informazioni sugli elettrodomestici per poter essere in grado di attuare la sua politica.
Può essere utile a tal fine definire una nuova sezione del modello contenente le strutture dati da essa utilizzate:

```
— section InfoElettrodomestico parents RequirementsConcepts
└
```

```
—
  CategoriaConsumo ::= BASSO | MEDIO | ALTO
└
```

```
┌─ InfoElettrodomestico
  consumo : ℕ↘1↖
  categoria : CategoriaConsumo
  consumoMinimo : ℕ↘1↖
```

dataAge : ℕ

tolleranzaAccensione : ℕ

ordineAttivazione : ℕ

|

consumo ≥ consumoMinimo

consumo < 45 ⇒ categoria = BASSO

45 ≤ consumo < 75 ⇒ categoria = MEDIO

consumo ≥ 75 ⇒ categoria = ALTO

└

┌ InfoElettrodomesticoInit

InfoElettrodomestico ’

consumoMinimo? : ℕ↘1↖

|

consumo’ = consumoMinimo?

consumoMinimo’ = consumoMinimo?

dataAge’ = 0

tolleranzaAccensione’ = 0

ordineAttivazione’ = 0

└

┌ AggiornaInformazioniTemporali

ΔInfoElettrodomestico

|

consumo’ = consumo

categoria’ = categoria

consumoMinimo’ = consumoMinimo

dataAge’ = dataAge + 1

```
  tolleranzaAccensione > 0 ⇒ tolleranzaAccensione′ = tolleranzaAccensione −
1
  tolleranzaAccensione = 0 ⇒ tolleranzaAccensione′ = tolleranzaAccensione
  ordineAttivazione′ = ordineAttivazione
└─
```

```
┌─ AggiornaStima
  ΔInfoElettrodomestico
  consumo? : ℕ↘1↖
|
  consumo? ≥ consumoMinimo

  consumo′ = consumo?
  consumoMinimo′ = consumoMinimo
  dataAge′ = 0
  tolleranzaAccensione′ = tolleranzaAccensione
  ordineAttivazione′ = ordineAttivazione
└─
```

. . .

Si noti che queste non sono le strutture dati che avrà il sistema, bensì informazioni che si può inferire sia richiesto mantenere al sistema per svolgere i compiti espressi nei requisiti.

Una volta definite le strutture dati ausiliare richieste al sistema si potrebbe in una nuova sezione definire lo schema di stato della CentralinaFissa e gli eventi compositi che ad essa avvengono:

```
— section CentralinaFissa parents InfoElettrodomestico
└─
```

. . .

┌─ CentralinaFissa

indirizzo : INDIRIZZO

interruttori : □□1↖ INTERRUTTORE

sogliaConsumi : ℕ

consumoTotale : ℕ

statoInterruttore : INTERRUTTORE ⇸ StatoInterruttore

configurazione : ELETTRODOMESTICO ⤖ INTERRUTTORE

info : ELETTRODOMESTICO ⇸ InfoElettrodomestico

inAccensione : iseq ELETTRODOMESTICO

inEsercizio : □□ELETTRODOMESTICO

attivazioni : ℕ

metodoStima : ℕ↘1↖ × CategoriaConsumo × ℕ↘1↖ × ℕ↘1↖ ⟶ ℕ↘1↖

|

consumoTotale = ∑ { e : inEsercizio • e ↦ (info e).consumo }

dom statoInterruttore = interruttori

ran configurazione ⊆ dom statoInterruttore

∀ i : interruttori ⧵ ran configurazione • statoInterruttore(i) = OFF

dom configurazione = dom info

ran inAccensione ⊆ dom configurazione

∀ e : ran inAccensione • (statoInterruttore ∘ configurazione)(e) = ON ∧ (info e).tolleranzaAccensione > 0

inEsercizio ⊆ dom configurazione

∀ e : inEsercizio • (statoInterruttore ∘ configurazione)(e) = ON ∧ (info e).tolleranzaAccensione = 0

∀ e1 , e2 : inEsercizio • (info e1).ordineAttivazione = (info e2).ordineAttivazione ⇔ e1 = e2

∀ e : inEsercizio • (info e).ordineAttivazione ≤ attivazioni

∀ e : dom info | (info e).dataAge > 2 • (statoInterruttore ∘ configurazione)(e) = OFF

∀ inputParam : dom metodoStima • metodoStima(inputParam) ≥ inputParam.3

∑ { e : dom configurazione | (statoInterruttore ∘ configurazione)(e) = ON ∧ (info e).consumoMinimo ≤ 44 • e ↦ 44 } ≤ sogliaConsumi

└

. . .

Ad esempio nel caso sia richiesta la disattivazione di un elettrodomestico potrebbero essere richiesti i seguenti eventi:

┌─ DisattivaElettrodomestico

ΔCentralinaFissa

e? : ELETTRODOMESTICO

|

e? ∈ dom configurazione

indirizzo′ = indirizzo

interruttori′ = interruttori

sogliaConsumi′ = sogliaConsumi

statoInterruttore′ = statoInterruttore ⊕ {configurazione(e?) ↦ OFF}

configurazione′ = configurazione

info′ = info

inAccensione′ = inAccensione ⩊ (ran inAccensione ⩤ {e?})

inEsercizio′ = inEsercizio ⩤ {e?}

attivazioni′ = attivazioni

metodoStima′ = metodoStima

└

```
┌─ RicezioneMessaggioRichiestaCentralinaMobile
  ΔCentralinaFissa
  m? : MessaggioRichiestaCentralinaMobile
|
  m?.destinazione = indirizzo
  m?.idApparato ∈ dom configurazione

  ∃ e? : ELETTRODOMESTICO
   | e? = m?.idApparato
   • [| m?.richiesta = ON ⇒ AttivaElettrodomestico
        m?.richiesta = OFF ⇒ DisattivaElettrodomestico ]
└─
```

Come si vede è possibile definire entità di crescente complessità senza essere obbligati a definire quale sarà la struttura reale del sistema. In tale fase l'unico obiettivo è precisare i requisiti e le proprietà richieste al sistema che sono dirette conseguenze logiche di essi. In tal modo è immediato rilevare porzioni dei requisiti mal definite o contraddittorie, poiché la loro traduzione in formule non può portare ad una coerente elaborazione delle informazioni.

Si noti che ad esempio la funzione `metodoStima()` della centralina fissa risulta parzialmente specificata e vincolata in modo molto lasco. Questo poiché i requisiti affermano che il sistema dovrà stimare l'andamento dei consumi degli elettrodomestici ma non dicono (e non devono dire) nulla su “come ciò possa essere possibile” o debba avvenire.

3.3 Z e l'Analisi del Problema

Si è visto come Z standard sia un valido strumento per la formalizzazione dei requisiti. Una

volta completato un modello dei requisiti un analista può procedere all'analisi di ciò che occorre al sistema per poterli soddisfare. In questa fase del processo di sviluppo viene delineata l'architettura logica del sistema e vengono individuate astrazioni chiave per la costruzione di esso. In questa fase l'utilizzo di metodologie object-oriented e in particolar modo delle classi sono un valido riferimento per la definizione dei concetti.

Purtroppo a questo punto il linguaggio Z comincia a mostrare i suoi limiti. Pur avendo le ottime caratteristiche mostrate, in Z non vi è alcun supporto per la definizione di classi, tassonomie di ereditarietà e reti di oggetti loro istanze che si riferiscono. Per non parlare della possibilità di definire astrazioni che rappresentano entità attive. L'analista si trova a questo punto di nuovo privo di uno strumento adatto.

Il problema della *object-orientation in Z* non è nuovo e nonostante molto lavoro sia stato fatto per cercare di ovviare al problema, attualmente sembrerebbe non esserci una via naturale per utilizzare lo stile object-oriented all'interno di Z [6].

Molti tentativi sono stati intrapresi in tal senso per cercare di sviluppare una notazione object-oriented per Z. Ad esempio in [7,8] vengono poste delle buone definizioni di base concettuali ma si riesce a catturare le gerarchie di classi solo in modo incompleto. Inoltre il suggerito utilizzo del costrutto della set-comprehension, anche per effettuare le invocazioni dei metodi, mostra seri problemi all'aumentare della complessità delle entità da definire.

In [9] si propone di utilizzare le definizioni assiomatiche per associare all'istanza di una classe le sue proprietà. Quando una classe viene estesa si suggerisce di aggiungere nuove proposizioni assiomatiche. Tuttavia questo significa definire un simbolo statico, per ogni nome di attributo di ogni classe, visibile all'interno di tutta la specifica. All'aumentare della complessità oltre all'esplosione del numero di costrutti utilizzati per definire una singola classe, andiamo incontro a seri problemi di name clashing fra classi che indicano attributi (anche di natura diversa) con lo stesso nome. La notazione non sembra dunque adatta a descrivere sistemi di complessità reale.

In [10] compare per la prima volta l'idea di vista che consente di strutturare le specifiche secondo una architettura di riferimento. Tuttavia in essa si esprimono le operazioni polimorfe di una classe astratta come disgiunzioni di tutte le sue possibili versioni concrete presenti nella specifica. Ciò significa che finché le sottoclassi non vengono definite concretamente, non si ha la possibilità di invocare il metodo della classe astratta, da parte di un'altra classe che debba usarla, utilizzando la sola interfaccia della superclasse. Ciò rende ad esempio impossibile definire strutture che utilizzino pattern come il Template Method [11]. Inoltre l'aggiunta di una nuova sottoclasse richiede di aggiornare la

definizione della superclasse, che viene dunque a dipendere dalle sottoclassi. Poiché anche le sottoclassi dipendono necessariamente dalla superclasse, si viene ad instaurare una dipendenza circolare.
Nemmeno estendendo Z introducendo nuovi costrutti si sono raggiunti risultati ottimali. Il linguaggio Object-Z estende la notazione Z per andare incontro al paradigma object-oriented [12]. Tuttavia così facendo complica notevolmente la semantica del linguaggio sottostante. Introducendo meccanismi molto vicini alla programmazione a oggetti, viene a perdersi in esso l'enfasi sulla natura astratta della analisi e progettazione *orientata* agli oggetti, che invece è ciò che interessa.
Insomma molti tentativi falliti poiché si riesce a catturare lo spazio concettuale object-oriented solo in modo parziale o non robusto all'aumentare della complessità del sistema da costruire. Eppure Z parte da basi estremamente semplici e generali.
Quello che occorre dunque è una notazione che consenta di esprimere concetti object-oriented in modo agevole, ma possegga allo stesso tempo una semplice semantica sottostante. Sarebbe dunque auspicabile non estendere Z con nuovi costrutti ma sviluppare una particolare notazione per esso, in modo da “ereditarne” le buone caratteristiche evidenziate e la semplicità della semantica. Per aiutare l'analista, tale notazione dovrà suggerire una metodologia sistematica per la definizione di nuovi concetti. Dovrà inoltre essere possibile definire entità sia attive che passive ed esprimere efficacemente le interazioni distribuite.

3.4 Z e il Progetto

Da quanto detto nel paragrafo precedente non risulta sorprendente che Z si riveli inadatto anche nella fase di progetto. La sua più grave mancanza in questa fase, consiste nel fatto che poiché si esprime un evento tramite pre e post condizioni, non vi è alcun supporto diretto per dire come passare concretamente da una precondizione verificata alla rispettiva post condizione. Questo passo è necessario se si vuole generare il codice del sistema dalla specifica di progetto. Inoltre in fase di progetto, si fa un ampio uso a livello logico del costrutto di interfaccia come identificatore di tipo, per strutturare la realizzazione di un blocco in accordo a selezionati schemi di soluzione (pattern). In Z non vi è alcun supporto diretto al concetto di interfaccia.

4
Analisi in Concepts Z

4.1 Obiettivi

Si è visto come Z standard presenti delle buone caratteristiche per il costruttore di software, ma al contempo grossi limiti, che lo rendono inadatto a supportare le cruciali fasi di Analisi del Problema e Progetto nel processo di sviluppo.
Noto un modello dei requisiti, l'analista si trova ora senza uno strumento adeguato per definire le astrazioni chiave del sistema. Si vuole dunque individuare uno strumento che consenta di supportare efficacemente le necessità dell'analista. Tale strumento deve:

- consentire di esprimere concetti complessi in modo agevole e astratto
- suggerire una metodologia sistematica per la definizione di tali concetti
- consentire la descrizione di entità sia attive che passive
- fornire un insieme di metafore in grado di esprimere in modo preciso l'interazione fra entità, anche distribuite
- avere una semplice semantica sottostante

Poiché si desidera una semantica semplice e il più possibile precisa e consentire all'analista di lavorare in modo astratto, si sceglie di utilizzare come base di partenza Z standard, che già possiede queste caratteristiche. Si è però visto come il costrutto schema non fornisca adeguato supporto alla definizione di concetti che invece possono essere espressi in modo agevole tramite l'uso di metodologie object-oriented. Si sceglie dunque di sviluppare una particolare notazione (di nome *Concepts Z*), sopra il linguaggio Z, la quale supporti l'object-orientation, senza il bisogno di introdurre nuovi costrutti nel linguaggio.

4.2 Definizioni preliminari

Prima di addentrarsi nel cuore della notazione, conviene dare alcune definizioni preliminari che torneranno utili nel resto della tesi:

— section BaseConcepts parents standard_toolkit

⌙

┌═[X]

∑ : (X ⇸ ℤ) ⟶ ℤ

|

∑ ∅ = 0

∀ o : X ; n : ℤ ; f : X ⇸ ℤ | o ∉ dom f • ∑ ({o ↦ n} ∪ f) = n + ∑ f

⌙

Definisce l'operatore di somma su un insieme di interi etichettati

—

□□:= True | False

⌙

Definizione del tipo primitivo boolean

┌

ℝ : ℙ □□

⌙

Denota l'insieme dei numeri reali

┌

ℝ↘1↖ : ℙ ℝ

|

ℝ↘1↖ = ℝ ∖ {0}

⌙

Denota l'insieme dei numeri reali escluso lo zero

⌈

$$\text{ceiling} : \mathbb{R} \longrightarrow \mathbb{Z}$$

⌋

$$\forall\, r : \mathbb{R} \bullet \text{ceiling}(r) = \min\{\, n : \mathbb{Z} \mid n \geq r \,\}$$

⌙

Denota l'intero superiore di un dato numero reale

Si noti che poiché l'insieme dei numeri reali e le operazioni su di essi sono ben note dalla matematica, eviteremo per brevità di darne qui una definizione esplicita.

4.3 Una Teoria Logica delle Classi

I concetti di classe e di ereditarietà fra classi sono la chiave per raggiungere l'obiettivo di poter definire concetti agevolmente. Occorre dunque trovare il modo di fornire supporto a questi all'interno della nostra notazione.

— section AnalysisConcepts parents BaseConcepts

⌙

Si inizia definendo l'universo di tutte le possibili classi

— [CLASS] ⌙

Si è detto di voler definire tassonomie di classi in relazione di ereditarietà. Poiché si vuole mantenere il più possibile semplice la notazione, si sceglie di concentrarsi su una *ereditarietà fra classi di tipo singolo*. Un argomento a favore di ciò, è che lo sforzo di definire una buona tassonomia risulta abbastanza vano se due categorie differenti

finiscono per avere una intersezione. Per quale motivo allora ci si è sforzati di classificarle in modo differente? Questo solitamente avviene a causa di una terza categoria che le estende entrambe. Si sceglie dunque di definire la relazione di ereditarietà come una funzione. La scelta di adottare l'ereditarietà singola aiuta inoltre l'analista a tenere il più possibile semplici, puliti e coesi i concetti definiti. Si noti che al fine di non causare dipendenze circolari fra le classi, tale relazione deve necessariamente essere aciclica.

```
|
  isA : CLASS ⇸ CLASS
|
  isA ↗+↙ ∩ id CLASS = ∅
└─
```

Secondo la metodologia object-oriented una classe rappresenta un insieme e le istanze di quella classe sono gli elementi di tale insieme. Si definisce quindi l'universo di tutte le possibili istanze:

```
── [ INSTANCE ] └─
```

E' ora possibile definire alcune relazioni sugli insiemi definiti:

```
── relation ( abstract _ )
└─
|
  extension : CLASS ↔ INSTANCE
  direct : CLASS ↔ INSTANCE
  abstract _ : ℙ CLASS
|
  ∀ cSub , cSuper : CLASS | isA(cSub) = cSuper • extension(|{cSub}|) ⊆
extension(|{cSuper}|)
```

direct = extension ⩤ (isA~ ⨾ extension)

direct~ ∈ INSTANCE ⇸ CLASS

∀ c : CLASS • abstract c ⇔ direct(⦇{c}⦈) = ∅

∀ c1 , c2 : CLASS | isA(c1) = isA(c2) • (extension(⦇{c1}⦈)) ∩ (extension(⦇{c2}⦈))
≠ ∅ ⇔ c1 = c2

└

La prima relazione definisce l'*estensione* di una classe come l'insieme di tutte le sue possibili istanze. Se due classi sono in relazione di ereditarietà l'estensione della sottoclasse è contenuta in quella della superclasse. Quindi un'istanza di sottoclasse è anche un'istanza di superclasse.
La seconda relazione lega una classe con le sue sole *istanze dirette*, ossia istanze di una classe che non sono istanze di sue sottoclassi. Si può essere istanze dirette di una sola classe.
La terza relazione definisce il concetto di *classe astratta* come una classe che non può avere istanze dirette. Una classe astratta è infatti per sua natura una categoria parzialmente definita.
Per finire le estensioni di classi con la stessa superclasse sono disgiunte.

Si definisce ora la *radice comune* di tutte le classi che verranno definite:

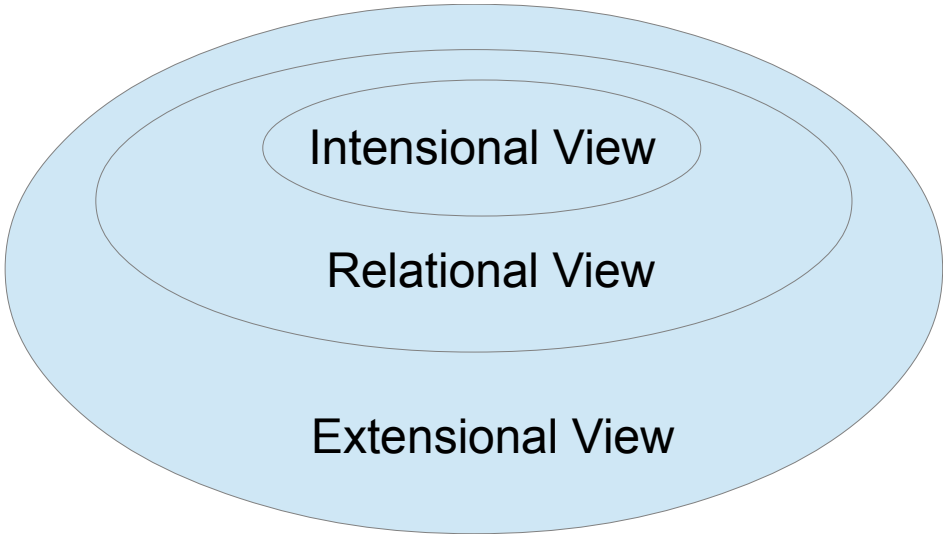
┌
Class : CLASS
│
Class ∉ dom isA
extension(⦇{Class}⦈) = INSTANCE
abstract Class
└

Tutte le classi che verranno definite sono sottoclassi di Class. Essa rappresenta la classe più generica, il concetto stesso di categoria. In quanto radice di un albero di tassonomia,

essa non può avere un padre, dunque la funzione isA non è applicabile ad essa.

4.4 Viste di un blocco

Si è detto che la notazione dovrà riflettere una metodologia sistematica per la definizione dei concetti. Un possibile modo per giungere a questo, consiste nel mettere a disposizione dell'analista una modalità per *definire classi secondo una certa architettura*. Più precisamente risulta conveniente specificare un blocco attraverso un insieme di *viste*, le quali consentano di descrivere un concetto seguendo un approccio “a cipolla”.



In particolare è possibile descrivere un concetto secondo le seguenti viste:

Intensional View: Questa vista consente di esprimere la struttura di stato (*intensione*) di una classe e di asserire su di essa invarianti “di basso livello”, ossia predicati che facciano asserzioni sullo stato di un'istanza di quella classe, senza coinvolgere lo stato di altre istanze riferite. In tale vista è possibile inoltre esprimere le firme (*signature*) degli eventi che possono avvenire a istanze della classe e specificare se si tratta di eventi di modifica dello stato o eventi di query. Si noti che fra essi ci sono anche l'evento di creazione e l'evento di finalizzazione di un'istanza.

Relational View: Tale vista mette a disposizione un meccanismo per associare a una data istanza il proprio stato e consente di completare la definizione della struttura di stato

asserendo su di essa invarianti che coinvolgono altre entità riferite. In tale vista è inoltre possibile specificare relazioni di cui godono le istanze di tale classe (ad esempio un ordinamento definito sulle istanze della classe).

Extensional View: In tale vista vengono fornite le descrizioni degli eventi che possono avvenire alla classe. Questa vista contiene inoltre dei meccanismi sintattici che rendono un'istanza di una certa classe riferibile dall'esterno.

4.4.1 La classe radice

E' possibile portare un primo esempio dell'uso di tali viste definendo la radice comune delle tassonomie, la classe Class:

```
— section ClassIntensional parents AnalysisConcepts
└
```

```
— [ ORDER ] └
universo delle label per gli ordinamenti definiti su classi
```

```
┌ ł Class
  this : INSTANCE
└
```

```
┌ ł ClassInit
  ł Class ’
  this? : INSTANCE
└
```

```
┌ ł ClassFin
  ł Class
```

└

Si assume la convenzione di utilizzare il carattere ł seguito dal nome della classe, come nome dello schema che definisce l'intensione della classe. Si premetterà inoltre inoltre ł seguito dal nome della classe, ai nomi degli schema che rappresentano le definizioni intensionali degli eventi.

L'intensione della classe Class è molto semplice e asserisce solamente che ogni istanza di una classe, mantiene un riferimento a se stessa.
Subito dopo si trova la definizione della signature degli eventi di creazione e di finalizzazione di una classe.
Il primo specifica che al termine dell'evento di creazione esisterà un'intensione e che l'evento di creazione prende in input un riferimento.
Il secondo afferma semplicemente che all'inizio dell'evento di finalizzazione esisterà un'intensione, ma non richiede input addizionali per poter procedere alla sua finalizzazione.

La specifica della classe Class non prevede ulteriori eventi che possano avvenire su di essa, che verranno via via aggiunti dalle sue sottoclassi astratte e concrete.

Si passa ora a descrivere la vista relazionale di Class

— section ClassRelational parents ClassIntensional
└

┌ ΩClass
 łClass : INSTANCE ⤖ łClass
|
 ∀ c : dom łClass • (łClass c).this = c
└

```
┌─ ℝClass
 ł Class
 Ω Class
|
 this ∈ dom ŁClass
└
```

In questa sezione si trovano due schema. Il primo descrive una funzione che consente di associare ad una istanza di classe Class il proprio stato (*link function*). Ogni classe conterrà una simile funzione per associare alle proprie istanze il rispettivo stato. Si assumerà la convenzione di chiamare Ω seguito dal nome della classe, lo schema in cui viene definita la link function della classe. Si noti come la funzione ŁClass risulti *parzialmente specificata*. Viene data indicazione sul suo universo di partenza, sul suo universo di arrivo, sul fatto che si tratti di una iniezione parziale sull'universo di partenza ed un vincolo sui valori in essa contenuti. Tuttavia non viene detto nulla circa il suo contenuto, che dunque secondo le usuali convenzioni di Z, può essere qualunque.

Il secondo schema completa la definizione dello stato di una classe inserendo invarianti che coinvolgono altre entità eventualmente riferite. Poiché una classe mantiene un riferimento a sé stessa, ora che si ha a disposizione la link function, si possono esprimere invarianti che coinvolgono tale riferimento. Lo schema degli invarianti relazionali ℝClass della classe Class asserisce semplicemente che this non è mai “nullo”. Si noti come questo schema includa l'intensione ŁClass (e dunque aggiunga invarianti a quelli già definiti in essa, vincolando ulteriormente l'intensione) e lo schema in cui è definita la link function della classe, per poterla utilizzare nella definizione degli invarianti.

Si completa ora la definizione del concetto astratto Class descrivendo la sua vista estensionale:

```
— section ClassExtensional parents ClassRelational
```

└

┌ ÆClass

ΩClass

classes : □□łClass

|

∀ c1 , c2 : classes • c1.this = c2.this ⇔ c1 = c2

łClass = { c : classes • c.this ↦ c }

└

Poiché questa vista rappresenta il guscio più esterno di una classe, è anche quello che consente ad essa di riferirsi ad altre classi (ad esempio per poterne invocare gli eventi).

Il primo schema che si incontra in una vista estensionale include lo schema ΩClass definito in precedenza e definisce un insieme finito di intensioni che rappresenta le istanze di quella classe presenti nel sistema in un dato momento. Si noti come, pur essendo i binding di uno schema dei valori in Z, il fatto che ciascuna intensione contenga un riferimento alla istanza a cui viene associata, consente di distinguere univocamente le intensioni anche nel caso in cui i restanti attributi di due istanze presentino gli stessi valori. Dunque se ci si preoccupa di attribuire un identificatore distinto a ciascuna istanza, l'insieme classes risulta concettualmente equivalente ad un *heap* per gli oggetti di quella classe.

Questo schema vincola poi ulteriormente la definizione della link function łClass, definendo il suo contenuto in funzione dell'insieme classes.

Ogni vista estensionale presenterà questo schema, dunque si adotterà la convenzione di indicarlo come Æ seguito dal nome della classe.

┌ ÆClassInit

ÆClass ’

|

classes’ = Ø

ŁClass′ = ∅

└

Questo schema descrive lo stato iniziale di quello precedentemente definito. Lo si indicherà come Æ seguito dal nome della classe, seguito da Init.

┌ ÐClass

ÆClass

└

Questo schema contiene esplicitamente le classi (blocchi) da cui dipende quella che si sta definendo (*dependency schema*). Rappresenta un utile ausilio per tracciare le dipendenze di un concetto definito e rende più compatta la notazione come vedremo fra poco.

┌ ÐClassInit

ÆClassInit

└

Questo schema è simile al precedente e consente di definire in modo “comodo” lo stato iniziale delle estensioni.

┌ ƒClass

ΔÐClass

ΔłClass

oc? : INSTANCE

|

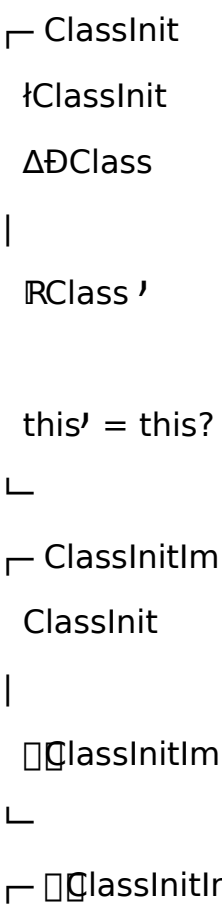
oc? ∈ dom ŁClass

θ łClass = ŁClass oc?

└

Quando un evento viene invocato su una classe, è possibile che a seguito di ciò, si verifichi una modifica dello stato dell'istanza su cui viene invocato o di quello di altre istanze da essa riferite (ad esempio l'invocazione di un metodo su un oggetto può causare in esso invocazioni su altri oggetti).

Dunque l'heap della classe e di quelle da cui essa dipende può variare. Questo schema (*schema di framing*) è strumentale a rendere un evento definito in una classe invocabile dall'esterno, dato il riferimento all'istanza su cui lo si vuole invocare. *Si noti come la variazione degli heap risulti parzialmente specificata*. Si denoteranno tali schema con *ƒ* più il nome della relativa classe. Questo schema non è strettamente necessario per la definizione della classe Class, in quanto essendo Class astratta e non specificando eventi ulteriori oltre quelli astratti di creazione e finalizzazione, non necessita di rendere nulla “invocabile” dall'esterno. E' stato introdotto per semplificare l'esposizione di alcuni concetti e poiché come si vedrà in seguito *è possibile generare automaticamente tutti gli schema ausiliari per il supporto alla object orientation data la dichiarazione di una classe* (per capirsi lo schema di framing è necessario in una classe astratta, poiché essa potrebbe voler definire vincoli su alcuni eventi che verranno specializzati nelle sottoclassi. Questo è importante poiché consente di rendere invocabile il metodo astratto. Chi invoca gli eventi utilizzando l'interfaccia della classe astratta, si attende il rispetto di un ben definito “contratto” per l'evento, che deve essere definito nella classe astratta).



łClass ’
this? : INSTANCE
|
this’ = this?
└

Lo schema ClassInit contiene la descrizione dell'evento astratto di creazione di una classe. Come si vede tale schema include la signature definita a livello intensionale. Questo schema asserisce che nell'intensione creata, dovranno essere verificati gli invarianti relazionali. Afferma inoltre che il riferimento fornito in ingresso risulterà assegnato all'attributo this. Poiché di una classe astratta non è possibile creare istanze, l'evento resta astratto, dunque non viene messo in combinazione con uno schema di framing e non sarà invocabile dall'esterno. Sarà tuttavia utile per le sottoclassi, le quali specificando i loro costruttori, potranno richiamare il costruttore della superclasse astratta per inizializzare parte dello stato (come è noto, costruttori e finalizzatori fanno eccezione rispetto agli usuali modi di definire metodi nel paradigma object oriented, poiché sono specifici per una certa classe).
Si ignorino i due schema sottostanti. Sono stati qui inseriti unicamente per correttezza formale. Il loro significato diverrà chiaro più avanti.

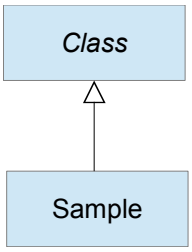
┌─ ClassFin
łClassFin
ΔÐClass
|
ℝClass
└
┌─ ClassFinImpl
ClassFin
└

Analogamente lo schema ClassFin asserisce la verità sugli invarianti relazionali nella intensione che si sta per eliminare. Come prima si ignori lo schema sottostante.

Si è a questo punto completata la definizione della radice comune Class. Tale descrizione è abbastanza minimale e come visto lascia ampi gradi di libertà per la definizione delle sue sottoclassi.

4.4.2 Un primo esempio di classe concreta

Verrà ora mostrato un esempio di definizione di una semplice classe concreta.
Riprendendo l'esempio del paragrafo 3.2.1 si supponga che al termine dell'analisi dei requisiti, al fine di rendere possibile la realizzazione della funzione metodoStima() l'analista abbia giudicato conveniente la creazione di un modello del dominio, in cui siano presenti entità (molto generali), capaci di fornire la stima di una variabile casuale a valori reali data una successione di prove (o campioni). In particolare si darà un'occhiata alla specifica dell'analista circa la classe che modella il concetto di campione di un valore numerico.



— section DomainModelConcepts parents AnalysisConcepts
⌙

. . .

|

Sample : CLASS

SAMPLE : ℙ INSTANCE

| (1)

```
  isA( Sample ) = Class
  extension(⦇{Sample}⦈) = SAMPLE
  ¬ abstract Sample
└
        . . .

— section SampleIntensional parents DomainModelConcepts , ClassIntensional
└

┌─ łSample
  łClass
  value : ℝ
  next : SAMPLE
|
  this ∈ SAMPLE
  value ≥ 0
└
```

Includendo l'intensione della classe estesa si è in grado di “ereditare” la sua definizione di stato

```
┌─ łSampleInit
  łClassInit
  łSample ’
  value? : ℝ
└

┌─ łSampleFin
  łClassFin
  łSample
```

└

Lo stesso vale per le definizioni intensionali degli eventi di creazione e finalizzazione. Si noti che poiché essi costituiscono delle eccezioni, possono aggiungere variabili di ingresso o uscita addizionali. Qui ad esempio si afferma che il costruttore di Sample prende in ingresso un valore reale, oltre a quanto già definito dal costruttore di Class.

```
┌─ PrełSampleGetValue
 łSample
└
┌─ PostłSampleGetValue
 ΞłSample
 PrełSampleGetValue
 value! : ℝ
└

┌─ PrełSampleGetNext
 łSample
└
┌─ PostłSampleGetNext
 ΞłSample
 PrełSampleGetNext
 next! : SAMPLE
└

┌─ PrełSampleSetNext
 łSample
 n? : SAMPLE
└
```

```
┌─ PostłSampleSetNext
  ΔłSample
  PrełSampleSetNext
└─
```

Le *signature* degli eventi vengono definite specificando dapprima le variabili in ingresso (schema di nome Pre + ł + nome della classe + nome evento). Successivamente tale schema viene esteso, specificando se si tratta di un evento di modifica o di query dello stato e aggiungendo le variabili di uscita (schema di nome Post + ł + nome della classe + nome evento). Si noti che occorrerebbe evitare di definire variabili che contengano un vincolo implicito nella loro dichiarazione (ad es. i naturali non nulli indicati con ℕ↘1↖), poiché come si vedrà le descrizioni degli eventi vengono realizzate *specificando separatamente le precondizioni e le post condizioni di un evento*. Questo è necessario se si vuole essere in grado di specializzare la descrizione di un evento in una sottoclasse. Definire variabili con vincoli impliciti, “mischia” le precondizioni con le postcondizioni, dunque rende problematica la successiva specializzazione della classe. *Le parti predicative degli schema che definiscono le signature devono restare vuote*.
Le signature degli eventi ordinari non possono essere modificate con l'aggiunta di variabili addizionali, da successive definizioni dell'evento in eventuali sottoclassi (i costruttori come detto fanno eccezione).

```
— section SampleRelational parents SampleIntensional , ClassRelational
└─
```

```
┌─ ΩSample
  łSample : SAMPLE ⤖ łSample
|                                                              ( 2 )
  ∀ s : dom łSample • (łSample s).this = s
└─
```

┌─ ℝSample
 ℝClass
 łSample
 ΩSample
|
 this ∈ dom ŁSample
 next ∈ dom ŁSample
└─

Poiché Sample estende Class gli invarianti relazionali di Sample si aggiungono a quelli definiti in Class.

— section SampleExtensional parents SampleRelational , ClassExtensional
└─

┌─ ÆSample
 ΩSample
 samples : □□łSample
| (3)
 ∀ s1 , s2 : samples • s1.this = s2.this ⇔ s1 = s2
 ŁSample = { s : samples • s.this ↦ s }
└─

┌─ ÆSampleInit
 ÆSample ′
| (4)
 samples′ = Ø
 ŁSample′ = Ø

└

┌─ SampleIsClass

ÆSample

ÆClass

| (5)

{s : samples • s.this} ⊆ {c : classes • c.this}

∀ s : samples ; c : classes | s.this = c.this • (λ ł Sample • θ ł Class)(s) = c

└

Questo schema asserisce che l'insieme delle istanze attualmente presenti nel sistema di una sottoclasse è contenuto nell'insieme delle istanze attualmente presenti della superclasse. E' necessario per la correttezza formale del modello espresso.

┌─ ÐSample

ÐClass

ÆSample

| (9)

SampleIsClass

└

Sample estende Class, dunque dipende da essa. Si noti come nominando il dependecy schema delle classi da cui si dipende si realizza un “dependency hiding”. In tal modo la specifica diviene ulteriormente *modulare*: se si fossero elencate esplicitamente tutte le estensioni di classe da cui si finisce per dipendere (a causa della transitività della relazione di dipendenza con una classe), qualora la classe da cui si dipende fosse cambiata, aggiungendo nuove dipendenze, si sarebbe dovuto aggiornare anche questo schema. In tal modo invece le dipendenze possono essere comodamente tracciate andando ad esplorare le definizioni dei vari dependency schema, senza che essi debbano essere continuamente aggiornati a seguito di un cambiamento nella rete di dipendenze.

┌─ ÐSampleInit

ÐClassInit (10)

ÆSampleInit

└

Questo schema utilizza lo stesso principio del precedente per definire gli stati iniziali delle estensioni.

┌─ ƒSampleN

ΔÐSample

łSample ’

os! : SAMPLE

| (6)

os! ∈ direct(⦇{Sample}⦈) ⧹ dom łSample

└

Questo *framing schema* è *specifico per l'evento di creazione*. Esso si occupa di fornire un identificatore univoco per ogni nuova istanza creata di quella classe. Sono necessari solo per le classi non astratte. Li indicheremo con ƒ, seguito dal nome della classe, seguito da N .

┌─ ƒSampleD

ΔÐSample

łSample

os? : SAMPLE

| (7)

os? ∈ dom łSample

θ łSample = łSample os?

└

Analogo al precedente per la finalizzazione

┌─ ƒSample

ΔÐSample

```
ΔłSample
os? : SAMPLE
|                                                                    ( 8 )
os? ∈ dom ŁSample
θ łSample = ŁSample os?
└
```

```
┌─ SampleInit
łSampleInit
ΔĐSample
|
ℝSample ’
ClassInit
this? ∈ SAMPLE
value? ≥ 0
value’ = value?
next’ = this?
└
```

```
—
SampleNew == ∃ łSample ’ • ƒSampleN ∧ SampleInit[os!/this?]
└
```

La classe Sample è una classe concreta. Dunque il suo costruttore deve essere invocabile per poterne creare delle istanze. Come si vede il framing schema di creazione passa l'identificatore selezionato allo schema che descrive il costruttore. Il riferimento all'oggetto creato viene poi restituito in uscita al chiamante. Contestualmente vengono nascoste all'esterno le variabili di stato dell'istanza creata dalla invocazione dell'evento.

Lo schema SampleNew così definito è dunque utilizzabile dall'esterno per creare dei Sample.

Si noti inoltre, come la descrizione del costruttore di Sample si appoggi a quella del costruttore di Class.

┌─ SampleFin
 łSampleFin
 ΔĐSample
|
 ℝSample
 ClassFin
└
—
 SampleDelete == ∃ łSample • ƒSampleD ∧ SampleFin
└

┌─ PreSampleGetValue
 PrełSampleGetValue
 ĐSample
|
 ℝSample
└
┌─ PostSampleGetValue
 PostłSampleGetValue
 ΞĐSample
|
 ΔℝSample
 value! = value
└
—
 SampleGetValue == ∃ ΞłSample • ƒSample ∧ PreSampleGetValue ∧
 PostSampleGetValue

Una analoga procedura avviene per rendere disponibili all'esterno gli altri eventi. Si noti come l'evento invocabile, risulti essere una congiunzione delle precondizioni e postcondizioni definite per esso. In particolare la precondizione dell'evento GetValue è sempre verificata, mentre la postcondizione afferma che il valore corrente dell'attributo value viene copiato in output.
Si noti inoltre come lo schema che definisce la postcondizione dell'evento asserisca la verità degli invarianti intensionali e relazionali prima e dopo l'evento (ΔℝSample), analogamente a quanto avveniva definendo un evento in modo classico su uno Z schema (vedi par. 4.1).

PreSampleGetNext
PreSampleGetNext
ĐSample
|
ℝSample

PostSampleGetNext
PostSampleGetNext
ΞĐSample
|
ΔℝSample
next! = next

SampleGetNext == ∃ ΞSample • ƒSample ∧ PreSampleGetNext ∧
PostSampleGetNext

PreSampleSetNext

```
 PreŁSampleSetNext
 ĐSample
|
 ℝSample
 n? ∈ dom ŁSample
└
┌─ PostSampleSetNext
 PostŁSampleSetNext
 ΔĐSample
|
 ΔℝSample
 this′ = this
 value′ = value
 next′ = n?
└
—
 SampleSetNext == ∃ ΔŁSample • ƒSample ∧ PreSampleSetNext ∧
                                          PostSampleSetNext
└
```

Questo evento definisce una precondizione per la sua esecuzione. In particolare, poiché l'analista ha deciso di consentire ad un campione di riferire il prossimo campione nella successione di osservazioni, al fine di permettere ulteriori elaborazioni, si afferma che il campione successivo fornito non può essere nullo. Nella postcondizione si afferma invece che this e value dovranno mantenere il loro valore mentre a next risulterà assegnato il campione fornito in ingresso.

Si noti come, con le convenzioni adottate, *quasi tutti gli schema ausiliari per il supporto alla object orientation di una classe, vengano definiti secondo schemi ben riconoscibili*. Questo significa che secondo i concetti MDSD costituiscono una “schematic part” [13]

della specifica e *possono essere generati automaticamente* a partire da un template. Ad esempio per la classe Sample, una volta noto lo schema (1), è possibile generare automaticamente gli schema (2), (3), (4), (5), (6), (7), (8). Mentre dallo schema (9) è possibile generare (10). In conseguenza di ciò nel resto della tesi si ometteranno gli schema ausiliari, mostrando solo quelli a carico dell'analista e del progettista.

4.4.3 Tabella dei simboli utilizzati

Simbolo	Significato
ł...	Intensione di classe o definizione intensionale di costruttore/finalizzatore se termina con Init/Fin
Preł... , Postł...	Definizione intensionale di un evento
Ω...	Schema che ospita la definizione di una link function
Ł...	Link function di una classe
ℝ...	Schema degli invarianti relazionali di una classe
Æ...	Estensione attuale di una classe o schema di inizializzazione di una estensione se termina con Init
Đ...	Dependency schema di una classe o schema di inizializzazione della catena di dipendenza se termina con Init
ƒ...	Schema di framing (di creazione/finalizzazione se termina con N/D)
...Init ...Fin	Indica la definizione estensionale di un costruttore/finalizzatore
Pre... Post...	Indicano le pre e post condizioni estensionali di un evento

4.5 Ereditarietà

Si è visto come in Concepts Z sia agevole definire classi e relazioni di ereditarietà fra esse. Tuttavia poiché si è in presenza di una notazione formale per esprimere concetti, ed è dunque possibile condurre su di essi ragionamenti rigorosi, viene da chiedersi se esista un qualche criterio per affermare che una classe estende correttamente un'altra classe.

Un contributo significativo per il corretto uso dell'ereditarietà [14,15], va sotto il nome di

Principio di Sostituibilità di Liskov: le sottoclassi dovrebbero essere sostituibili con le loro superclassi.

Poiché la principale causa di violazioni al principio di Liskov è dato dalla ridefinizione degli eventi nelle sottoclassi, un secondo significativo contributo [16,17] per evitare tali violazioni, consiste nel *Design by Contract: Le specializzazioni degli eventi nelle sottoclassi devono avere precondizioni identiche o meno stringenti e post condizioni identiche o più stringenti.*

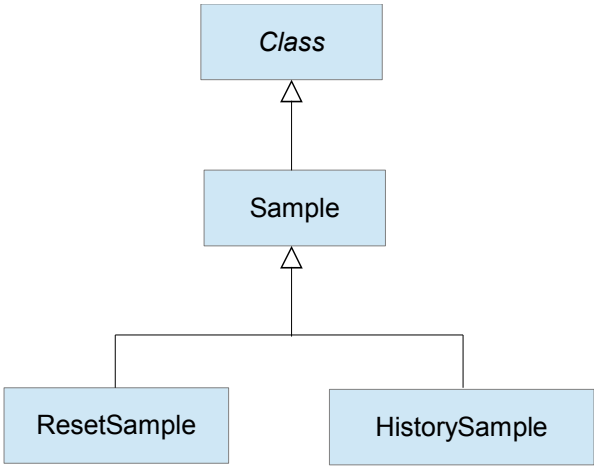
Il principio di Liskov e il Design by Contract consentono di utilizzare istanze di sottoclassi laddove ci si aspetta di utilizzare una loro superclasse, poiché in tal modo il loro comportamento è sempre consistente con il contratto astratto della superclasse.
La presenza del concetto di *raffinamento* in Z consente di fare ulteriori precisazioni.
L'idea di base è quella di vedere una sottoclasse come un raffinamento della sua superclasse (*behavioural conformity*).
E' possibile poi controllare la behavioural conformity, dimostrando la correttezza del raffinamento. In particolare occorre rilassare i vincoli di applicabilità e correttezza citati nel paragrafo 3.1.
La cosa più importante da notare, è che nel caso in cui si erediti da sole classi astratte, è possibile per le sottoclassi vincolare ulteriormente il contratto definito nella superclasse. Una *classe astratta* infatti non ha istanze dirette, dunque *rappresenta una definizione parziale* di un concetto, che ci si aspetta venga completata nelle sue specializzazioni. Gli eventi definiti in una classe astratta non vengono mai eseguiti “così come sono”, dato che non esistono istanze dirette, ne vengono sempre invocate delle versioni specializzate presenti nelle sue sottoclassi concrete. Questo significa che è possibile per l'analista utilizzare una classe astratta come “scheletro” per la definizione di concetti, a cui andranno via via aggiunti dettagli nelle specializzazioni, abilitando il tal modo il riuso.
Le precondizioni espresse in un evento della classe astratta, saranno solo una parte delle vere precondizioni. Dunque nel caso si erediti da sole classi astratte, una sottoclasse può definire ulteriori invarianti sullo stato ereditato e ulteriori precondizioni negli eventi astratti che specializza.
In effetti solo le versioni concrete dei concetti individuati dall'analista (che corrispondono ai blocchi da progettare), sono di interesse per le restanti fasi del processo di sviluppo. L'analista deve però stare attento a non basarsi sulle precondizioni espresse in una classe

astratta, per la specifica delle interazioni fra blocchi, nel caso in cui intenda utilizzare tale classe astratta come scheletro. Dunque una classe astratta scheletro, non dovrebbe mai essere fornita a componenti esterni come generico tipo polimorfo.
Tutto ciò è stato dimostrato in [18], in questa sede ci si limiterà quindi a fornire un insieme di regole che riassumono i criteri da tenere a mente per il corretto uso dell'ereditarietà all'interno della notazione Concepts Z:

- Le signature Z delle operazioni specializzate nella sottoclasse *devono coincidere* con quelle definite nella superclasse
- *E' sempre possibile* per una sottoclasse *definire un nuovo evento*
- Una classe che estende un'altra classe, può definire nuovi invarianti riguardanti la porzione di stato aggiunta dalla sottoclasse e ulteriori postcondizioni negli eventi specializzati, che però non devono risultare contraddittorie con quanto affermato nelle sue superclassi. Può inoltre rilassare le precondizioni di alcuni degli eventi specializzati.
- *Una classe che eredita da una classe concreta non può* definire nuovi invarianti sullo stato ereditato o nuove precondizioni sugli eventi che specializza.
- *Una classe che eredita da sole classi astratte può* introdurre ulteriori invarianti sullo stato ereditato e ulteriori precondizioni sugli eventi specializzati

4.5.1 Un esempio di specializzazione

Verrà ora mostrato un piccolo esempio che mostra la specializzazione di eventi in Concepts Z.



Si supponga di voler definire due sottotipi di Sample. Il primo che accetti anche un successore nullo e in conseguenza di ciò resetti il campo next ed il secondo che mantiene una storia di tutti i suoi successori nel tempo.

```
|
  ResetSample : CLASS
  RESET_SAMPLE : ℙ SAMPLE
|
  isA( ResetSample ) = Sample
  extension(⦇{ResetSample}⦈) = RESET_SAMPLE
  ¬ abstract ResetSample
└
```

```
|
  HistorySample : CLASS
  HISTORY_SAMPLE : ℙ SAMPLE
|
  isA( HistorySample ) = Sample
  extension(⦇{HistorySample}⦈) = HISTORY_SAMPLE
  ¬ abstract HistorySample
└
```

```
— section ResetSampleIntensional parents SampleIntensional
└

┌─ ł ResetSample
 ł Sample
└

┌─ ł ResetSampleInit
 ł SampleInit
 ł ResetSample ’
└

┌─ ł ResetSampleFin
 ł SampleFin
 ł ResetSample
└

┌─ Preł ResetSampleGetValue
 Preł SampleGetValue
 ł ResetSample
└
┌─ Postł ResetSampleGetValue
 Postł SampleGetValue
 ≡ ł ResetSample
 Preł ResetSampleGetValue
└
```

```
┌─ PrełResetSampleGetNext
  PrełSampleGetNext
  łResetSample
└
┌─ PostłResetSampleGetNext
  PostłSampleGetNext
  ∃łResetSample
  PrełResetSampleGetNext
└

┌─ PrełResetSampleSetNext
  PrełSampleSetNext
  łResetSample
└
┌─ PostłResetSampleSetNext
  PostłSampleSetNext
  ΔłResetSample
  PrełResetSampleSetNext
└
```

Come detto una sottoclasse non può cambiare la signature degli eventi specializzati, deve tuttavia includere le eventuali variabili di stato aggiuntive che dichiara, nella descrizione dell'evento.

```
— section ResetSampleRelational parents ResetSampleIntensional ,
SampleRelational
└
```

. . .

┌─ ℝResetSample
 ℝSample
 ⧩ResetSample
 ΩResetSample
└

— section ResetSampleExtensional parents ResetSampleRelational ,
SampleExtensional
└

. . .

┌─ ÐResetSample
 ÐSample
 ÆResetSample
|
 ResetSampleIsSample
└

. . .

┌─ ResetSampleInit
 ⧩ResetSampleInit
 ΔÐResetSample
|
 ℝResetSample ′
 SampleInit
 this? ∈ RESET_SAMPLE

└

—

ResetSampleNew == ∃ łResetSample ’ • ƒResetSampleN ∧

ResetSampleInit[os!/this?]

└

┌─ ResetSampleFin

łResetSampleFin

ΔĐResetSample

|

ℝResetSample

SampleFin

└

—

ResetSampleDelete == ∃ łResetSample • ƒResetSampleD ∧ ResetSampleFin

└

┌─ PreResetSampleGetValue

PrełResetSampleGetValue

ĐResetSample

|

ℝResetSample

PreSampleGetValue

└

┌─ PostResetSampleGetValue

PostłResetSampleGetValue

≡ĐResetSample

|

ΔℝResetSample

PostSampleGetValue

└

—

ResetSampleGetValue == ∃ ≅łResetSample • ƒResetSample ∧

PreResetSampleGetValue ∧ PostResetSampleGetValue

└

Gli eventi GetValue e GetNext restano invariati, dunque la descrizione della precondizione e della postcondizione dell'evento più specifico si appoggiano a quelle della superclasse

┌─ PreResetSampleGetNext

PrełResetSampleGetNext

ÐResetSample

|

ℝResetSample

PreSampleGetNext

└

┌─ PostResetSampleGetNext

PostłResetSampleGetNext

≅ÐResetSample

|

ΔℝResetSample

PostSampleGetNext

└

—

ResetSampleGetNext == ∃ ≅łResetSample • ƒResetSample ∧

PreResetSampleGetNext ∧ PostResetSampleGetNext

└

┌─ PreResetSampleSetNextNotNull
 PrełResetSampleSetNext
 ÐResetSample
|
 ℝResetSample
 PreSampleSetNext
└─

┌─ PostResetSampleSetNextNotNull
 PostłResetSampleSetNext
 ΔÐResetSample
|
 ΔℝResetSample
 PostSampleSetNext
└─

┌─ PreResetSampleSetNextNull
 PrełResetSampleSetNext
 ÐResetSample
|
 ℝResetSample
 n? ∉ dom ŁSample
└─

┌─ PostResetSampleSetNextNull
 PostłResetSampleSetNext
 ΔÐResetSample
|
 ΔℝResetSample
 this′ = this
 value′ = value

next′ = this

└

—

ResetSampleSetNext == ∃ ΔłResetSample • ƒResetSample ∧

((PreResetSampleSetNextNotNull ∧ PostResetSampleSetNextNotNull) ∨

(PreResetSampleSetNextNull ∧ PostResetSampleSetNextNull))

└

La classe ResetSample rilassa la precondizione dell'evento SetNext accettando anche valori “nulli” in ingresso (quindi questa specializzazione ha precondizioni *meno stringenti*). Si noti come quanto specificato nella superclasse, sia trattato come un *caso di funzionamento* in *disgiunzione* con il caso di funzionamento aggiuntivo (PreResetSampleSetNextNull ∧ PostResetSampleSetNextNull), il quale ha una precondizione *mutuamente esclusiva* con il primo.

— section HistorySampleIntensional parents SampleIntensional

└

┌ łHistorySample

łSample

history : seq↘1↖ SAMPLE

|

head history = this

└

Questa classe aggiunge un attributo, dunque può asserire invarianti su di esso e dovrà indicare proposizioni nel suo costruttore circa la sua creazione

┌ łHistorySampleInit

łSampleInit

łHistorySample′

└

┌─ łHistorySampleFin
 łSampleFin
 łHistorySample
└

┌─ PrełHistorySampleGetValue
 PrełSampleGetValue
 łHistorySample
└
┌─ PostłHistorySampleGetValue
 PostłSampleGetValue
 ≡łHistorySample
 PrełHistorySampleGetValue
└

┌─ PrełHistorySampleGetNext
 PrełSampleGetNext
 łHistorySample
└
┌─ PostłHistorySampleGetNext
 PostłSampleGetNext
 ≡łHistorySample
 PrełHistorySampleGetNext
└

┌─ PrełHistorySampleSetNext

PrełSampleSetNext

łHistorySample

└

┌─ PostłHistorySampleSetNext

PostłSampleSetNext

ΔłHistorySample

PrełHistorySampleSetNext

└

Si noti come ΔłHistorySample e ΞłHistorySample consentano di trattare il nuovo attributo all'interno delle descrizioni degli eventi

— section HistorySampleRelational parents HistorySampleIntensional , SampleRelational

└

. . .

┌─ ℝHistorySample

ℝSample

łHistorySample

ΩHistorySample

|

ran history ⊆ dom ŁSample

└

— section HistorySampleExtensional parents HistorySampleRelational , SampleExtensional

└

. . .

┌─ ÐHistorySample
ÐSample
ÆHistorySample
|
HistorySampleIsSample
└─

. . .

┌─ HistorySampleInit
łHistorySampleInit
ΔÐHistorySample
|
ℝHistorySample ′
SampleInit
this? ∈ HISTORY_SAMPLE
history′ = ⟨ this? ⟩
└─
—
HistorySampleNew == ∃ łHistorySample ′ • ƒHistorySampleN ∧
HistorySampleInit[os!/this?]
└─

Qui si asserisce che il riferimento passato in ingresso dovrà essere del tipo corretto e che il nuovo attributo, viene inizializzato ad una lista contenente il solo riferimento passato in ingresso.

┌─ HistorySampleFin
łHistorySampleFin

ΔĐHistorySample

|

ℝHistorySample

SampleFin

└

—

HistorySampleDelete == ∃ łHistorySample • ƒHistorySampleD ∧

HistorySampleFin

└

┌ PreHistorySampleGetValue

PrełHistorySampleGetValue

ĐHistorySample

|

ℝHistorySample

PreSampleGetValue

└

┌ PostHistorySampleGetValue

PostłHistorySampleGetValue

ΞĐHistorySample

|

ΔℝHistorySample

PostSampleGetValue

└

—

HistorySampleGetValue == ∃ ΞłHistorySample • ƒHistorySample ∧

PreHistorySampleGetValue ∧ PostHistorySampleGetValue

└

```
┌─ PreHistorySampleGetNext
  PrełHistorySampleGetNext
  ĐHistorySample
|
  ℝHistorySample
  PreSampleGetNext
└─
┌─ PostHistorySampleGetNext
  PostłHistorySampleGetNext
  ΞĐHistorySample
|
  ΔℝHistorySample
  PostSampleGetNext
└─
─
  HistorySampleGetNext == ∃ ΞłHistorySample • ƒHistorySample ∧
                              PreHistorySampleGetNext ∧ PostHistorySampleGetNext
└─


┌─ PreHistorySampleSetNext
  PrełHistorySampleSetNext
  ĐHistorySample
|
  ℝHistorySample
  PreSampleSetNext
└─
┌─ PostHistorySampleSetNext
```

PostłHistorySampleSetNext

ΔĐHistorySample

|

ΔℝHistorySample

PostSampleSetNext

history′ = history ⌢ ⟨ next ⟩

└

—

HistorySampleSetNext == ∃ ΔłHistorySample • ƒHistorySample ∧

PreHistorySampleSetNext ∧ PostHistorySampleSetNext

└

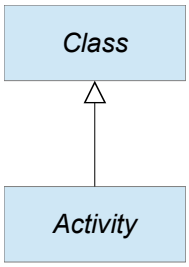
La classe HistorySample definisce una postcondizione aggiuntiva (quindi questa specializzazione ha postcondizioni *più stringenti*), la quale afferma che il vecchio successore viene aggiunto in coda alla sequenza history.

4.6 Modellazione di entità attive

Si è visto come in Concepts Z sia possibile esprimere classi ed effettuare una loro corretta specializzazione. Nel paradigma object-oriented un oggetto è tipicamente associato ad un'entità passiva, sulla quale vengono invocati metodi dall'esterno. Si è però detto voler essere in grado di modellare entità attive, ossia entità che posseggono un loro flusso di controllo interno. Definiamo *attività* una entità dotata di uno stato e di una *sequenza di azioni* espressa in funzione di tale stato. Tale sequenza di azioni sotto opportune condizioni può sospendersi e in seguito essere riavviata.

4.6.1 Il concetto di Attività

Definiamo il concetto di attività utilizzando gli usuali meccanismi di ereditarietà offerti dalla nostra notazione:



— section AnalysisConcepts parents BaseConcepts

⌙

. . .

—

ExecutionState ::= NEW | RUNNABLE | BLOCKED | TERMINATED

⌙

Un attività può trovarsi in uno dei seguenti stati:

New: è stata creata un'istanza di attività ma la sua sequenza di azioni non è stata ancora avviata

Runnable: la sequenza di azioni dell'attività può essere eseguita creando effetti nel sistema

Blocked: la sequenza di azioni è stata eseguita ed ha finito col sospendersi. Non potrà essere eseguita nuovamente finchè l'attività permane in questo stato.

Terminated: l'attività ha completato la sua sequenza di azioni

|

Activity : CLASS

ACTIVITY : ℙ INSTANCE

|

```
  isA(Activity) = Class
  extension(⦇{Activity}⦈) = ACTIVITY
  abstract Activity
└
```

Si procede ora a descrivere nel dettaglio il concetto di attività:

```
— section ActivityIntensional parents ClassIntensional
└
```

```
┌ ł Activity
  ł Class
  executionState : ExecutionState
|
   this ∈ ACTIVITY
└
```

```
┌ ł ActivityInit
  ł ClassInit
  ł Activity ’
└
```

```
┌ ł ActivityFin
  ł ClassFin
  ł Activity
└
```

```
┌ Preł ActivityRun
  ł Activity
```

└

┌─ PostłActivityRun
 ΔłActivity
 PrełActivityRun
└

┌─ PrełActivityStart
 łActivity
└
┌─ PostłActivityStart
 ΔłActivity
 PrełActivityStart
└

┌─ PrełActivitySetExecutionState
 łActivity
 newState? : ExecutionState
└
┌─ PostłActivitySetExecutionState
 ΔłActivity
 PrełActivitySetExecutionState
└

┌─ PrełActivityGetExecutionState
 łActivity
└
┌─ PostłActivityGetExecutionState
 ΞłActivity

PrełActivityGetExecutionState

executionState! : ExecutionState

└

— section ActivityRelational parents ActivityIntensional , ClassRelational

└

. . .

┌ ℝActivity

ℝClass

łActivity

ΩActivity

└

— section ActivityExtensional parents ActivityRelational , ClassExtensional

└

. . .

┌ ÐActivity

ÐClass

ÆActivity

|

ActivityIsClass

└

. . .

┌─ ActivityInit
 ł ActivityInit
 Δ Đ Activity
|
 ℝ Activity ’
 ClassInit
 this? ∈ ACTIVITY
 executionState’ = NEW
└

Quando un'attività viene istanziata, si troverà inizialmente nello stato New.

┌─ ActivityFin
 ł ActivityFin
 Δ Đ Activity
|
 ℝ Activity
 ClassFin
 executionState = TERMINATED
└

Per poter essere finalizzata, un'attività dovrà essere terminata

┌─ PreActivityRun
 Preł ActivityRun
 Đ Activity
|
 ℝ Activity
 executionState = RUNNABLE
└

┌─ PostActivityRun

PostłActivityRun

ΔĐActivity

|

ΔℝActivity

this′ = this

└

—

ActivityRun == ∃ ΔłActivity • ƒActivity ∧ PreActivityRun ∧ PostActivityRun

└

Questo evento può avvenire all'attività quando lo stato è Runnable. Tipicamente un tale evento si verifica quando lo scheduler del sistema operativo sottostante, o del supporto di esecuzione, decide di mettere in esecuzione il thread rappresentato dall'attività.

Questo evento deve essere specializzato e rappresenta il filo di esecuzione sequenziale dell'attività.

Come si vedrà, una specializzazione di tale evento può decidere di sospendersi, in uno dei suoi casi di funzionamento, implicando come ultimo evento “Await” su una variabile condizione. In tal caso, dovrà salvare informazione appropriata nello stato della specializzazione di attività, che le permetta di riprendere l'esecuzione da dove si era interrotta, quando lo stato tornerà ad essere Runnable. Il modo migliore per fare ciò, consiste nel suddividere Run in altri sottoeventi della specializzazione di attività e fare in modo che Run li implichi nel modo appropriato, in base allo stato corrente dell'attività specializzata.

Come si vede, non si afferma nulla circa l'attributo executionState nello stato finale. Dunque la specializzazione di Run deve sempre specificare cosa avviene a executionState al termine dell'evento.

┌ PreActivityStart

PrełActivityStart

ĐActivity

|

ℝActivity

executionState = NEW

└

┌ PostActivityStart

PostłActivityStart

ΔĐActivity

|

ΔℝActivity

this′ = this

executionState′ = RUNNABLE

└

—

ActivityStart == ∃ ΔłActivity • ƒActivity ∧ PreActivityStart ∧ PostActivityStart

└

Questo evento può avvenire solo una volta e rende possibile l'invocazione dell'evento Run da parte dello scheduler (qualunque esso sia).
Si noti che l'evento Start non implica l'evento Run (che avverrà quando avverrà).

┌ PreActivitySetExecutionState

PrełActivitySetExecutionState

ĐActivity

|

ℝActivity

executionState = RUNNABLE ⇒ newState? ∈ { RUNNABLE , BLOCKED , TERMINATED }

executionState = BLOCKED ⇒ newState? ∈ { BLOCKED , RUNNABLE }

executionState = NEW ⇒ newState? ∈ { NEW , RUNNABLE }

executionState = TERMINATED ⇒ newState? = TERMINATED

└

┌ PostActivitySetExecutionState

PostłActivitySetExecutionState

ΔĐActivity

|

ΔℝActivity

this’ = this

executionState’ = newState?

└

—

ActivitySetExecutionState == ∃ ΔłActivity • ƒActivity ∧ PreActivitySetExecutionState ∧ PostActivitySetExecutionState

└

Questo evento consente una evoluzione consistente dello stato dell'attività. Come si vede non è possibile tornare nello stato New e una volta entrati nello stato Terminated vi si può solo permanervi.

┌ PreActivityGetExecutionState

PrełActivityGetExecutionState

ĐActivity

|

ℝActivity

└

┌ PostActivityGetExecutionState

PostłActivityGetExecutionState

Ξ ĐActivity

|

ΔℝActivity

executionState! = executionState

└

—

ActivityGetExecutionState == ∃ ⨋ ł Activity • ƒ Activity ∧

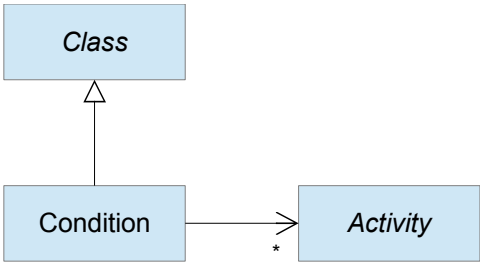
PreActivityGetExecutionState ∧ PostActivityGetExecutionState

⌙

Questo evento consente infine di leggere lo stato di esecuzione dell'attività.

4.6.2 Il concetto di Variabile Condizione

Si definirà ora il concetto di *variabile condizione*, il quale consente ad una attività di sospendersi e consente in un secondo momento ad altri di farla tornare in esecuzione.



— section AnalysisConcepts parents BaseConcepts

⌙

. . .

ı

Condition : CLASS

CONDITION : ℙ INSTANCE

|

isA(Condition) = Class

extension(⦇{Condition}⦈) = CONDITION

¬ abstract Condition

⌙

. . .

— section ConditionIntensional parents ClassIntensional

└

┌─ łCondition
 łClass
 activityQueue : iseq ACTIVITY
|
 this ∈ CONDITION
└

┌─ łConditionInit
 łClassInit
 łCondition ’
└

┌─ łConditionFin
 łClassFin
 łCondition
└

┌─ PrełConditionAwait
 łCondition
 a? : ACTIVITY
└
┌─ PostłConditionAwait
 ΔłCondition
 PrełConditionAwait

└

┌─ PrełConditionSignal
ł Condition
└

┌─ PostłConditionSignal
Δł Condition
Prełł ConditionSignal
└

┌─ PrełConditionGetWaitCount
ł Condition
└

┌─ PostłConditionGetWaitCount
≡ł Condition
Prełł ConditionGetWaitCount
waitCount! : ℕ
└

┌─ PrełConditionSignalAll
ł Condition
└

┌─ PostłConditionSignalAll
Δł Condition
Prełł ConditionSignalAll
└

— section ConditionRelational parents ConditionIntensional , ActivityRelational
└

. . .

┌─ ℝCondition
ℝClass
łCondition
ΩActivity
ΩCondition
|
ran activityQueue ⊆ dom ŁActivity
∀ a : ran activityQueue • (ŁActivity a).executionState = BLOCKED
└─

— section ConditionExtensional parents ConditionRelational ,
ActivityExtensional
└─

. . .

┌─ ÐCondition
ÐActivity
ÆCondition
|
ConditionIsClass
└─

. . .

┌─ ConditionInit
łConditionInit
ΔÐCondition

|

ℝCondition ’

ClassInit

this? ∈ CONDITION

activityQueue’ = ⟨ ⟩

└

—

ConditionNew == ∃ łCondition ’ • ƒConditionN ∧ ConditionInit[oc!/this?]

└

Il costruttore afferma che this? Dovrà essere del giusto tipo e che la coda di attività è inizialmente vuota.

┌ ConditionFin

łConditionFin

ΔĐCondition

|

ℝCondition

ClassFin

activityQueue = ⟨ ⟩

└

—

ConditionDelete == ∃ łCondition • ƒConditionD ∧ ConditionFin

└

Quì si afferma che è possibile eseguire la finalizzazione di una variabile condizione solo se nessuna attività è sospesa su di essa

┌ PreConditionAwait

PrełConditionAwait

ĐCondition

|

ℝCondition

a? ∈ dom ŁActivity

(ŁActivity a?).executionState = RUNNABLE

└

┌─ PostConditionAwait

PostłConditionAwait

ΔĐCondition

|

ΔℝCondition

∃ newState? : ExecutionState

| newState? = BLOCKED

• ActivitySetExecutionState[a?/oa?]

this′ = this

activityQueue′ = activityQueue ⌢ ⟨ a? ⟩

└

—

ConditionAwait == ∃ ΔłCondition • ƒCondition ∧ PreConditionAwait ∧ PostConditionAwait

└

Questo evento, quando invocato da una specializzazione di attività passando se stessa, le consente di sospendersi su tale variabile condizione. Si noti come questo implichi l'invocazione dell'evento SetExecutionState sull'attività chiamante.

┌─ PreConditionSignal

PrełConditionSignal

ĐCondition

|

ℝCondition

└

┌ PostConditionSignal

PostłConditionSignal

ΔĐCondition

|

ΔℝCondition

activityQueue ≠ ⟨ ⟩ ⇒ ∃ a : ACTIVITY ; newState? : ExecutionState

| a = head activityQueue ∧ newState? = RUNNABLE

• ActivitySetExecutionState[a/oa?]

this′ = this

activityQueue ≠ ⟨ ⟩ ⇒ activityQueue′ = tail activityQueue

activityQueue = ⟨ ⟩ ⇒ activityQueue′ = activityQueue

└

—

ConditionSignal == ∃ ΔłCondition • ƒCondition ∧ PreConditionSignal ∧ PostConditionSignal

└

Questo evento consente di svegliare la prima attività della coda. Si noti che Signal non implica l'evento Run sull'attivita svegliata. L'evento Run su tale attivita avverrà quando avverrà.

┌ PreConditionGetWaitCount

PrełConditionGetWaitCount

ĐCondition

|

ℝCondition

└

┌ PostConditionGetWaitCount

PostłConditionGetWaitCount

ΞĐCondition

|

ΔℝCondition

waitCount! = #activityQueue

└

—

ConditionGetWaitCount == ∃ ΞłCondition • ƒCondition ∧ PreConditionGetWaitCount ∧ PostConditionGetWaitCount

└

┌ AwakeAll

ΔĐCondition

c? : CONDITION

|

#(ŁCondition c?).activityQueue = 0 ⇒ ΞĐCondition

#(ŁCondition c?).activityQueue > 0 ⇒ ConditionSignal[c?/oc?] § AwakeAll

└

┌ PreConditionSignalAll

PreŁConditionSignalAll

ĐCondition

|

ℝCondition

└

┌ PostConditionSignalAll

PostŁConditionSignalAll

ΔĐCondition

|

ΔℝCondition

AwakeAll[this/c?]

this′ = this

└

—

ConditionSignalAll == ∃ ΔłCondition • ƒCondition ∧ PreConditionSignalAll ∧ PostConditionSignalAll

└

Consente di svegliare tutte le attività sospese sulla variabile condizione. Si noti l'uso di uno schema ausiliario *ricorsivo*, per invocare altri eventi sulla istanza corrente.

4.6.3 Un esempio di attività

Diamo ora un esempio di attività concreta. Si supponga che un evento di basso livello (ad esempio un timer), svegli periodicamente un'attività, causando una Signal sulla variabile condizione su cui l'attività è in attesa. Quest'ultima ha il compito di invocare su un certo oggetto (CentralinaFissa) un evento (Tick) per notificargli il passaggio del tempo ed infine tornare in attesa, finché non viene arrestata.

|

ConteggioTempo : CLASS
CONTEGGIO_TEMPO : ℙ ACTIVITY

|

isA(ConteggioTempo) = Activity
extension(⦇{ConteggioTempo}⦈) = CONTEGGIO_TEMPO
¬ abstract ConteggioTempo

└

— section ConteggioTempoIntensional parents CentralinaFissaConcepts , ActivityIntensional

└

┌─ łConteggioTempo

łActivity

centralina : CENTRALINA_FISSA

attesaSecondi : CONDITION

stop : □□

|

this ∈ CONTEGGIO_TEMPO

└

Lo stato della attività definisce un riferimento alla centralina fissa, uno alla variabile condizione su cui deve sospendersi ed un booleano che indica se è stata richiesta la terminazione dell'attività.

┌ łConteggioTempoInit

łActivityInit

łConteggioTempo'

centralina? : CENTRALINA_FISSA

attesaSecondi? : CONDITION

└

┌ łConteggioTempoFin

łActivityFin

łConteggioTempo

└

┌ PrełConteggioTempoRun

PrełActivityRun

łConteggioTempo

└

┌ PostłConteggioTempoRun

PostłActivityRun

ΔłConteggioTempo

PrełConteggioTempoRun

└

┌ PrełConteggioTempoStart

PrełActivityStart

łConteggioTempo

└

┌ PostłConteggioTempoStart

PostłActivityStart

ΔłConteggioTempo

PrełConteggioTempoStart

└

┌ PrełConteggioTempoSetExecutionState

PrełActivitySetExecutionState

łConteggioTempo

└

┌ PostłConteggioTempoSetExecutionState

PostłActivitySetExecutionState

ΔłConteggioTempo

PrełConteggioTempoSetExecutionState

└

┌ PrełConteggioTempoGetExecutionState

PrełActivityGetExecutionState

łConteggioTempo

└

```
┌─ PostłConteggioTempoGetExecutionState
  PostłActivityGetExecutionState
  ΞłConteggioTempo
  PrełConteggioTempoGetExecutionState
└─

┌─ PrełConteggioTempoStop
  łConteggioTempo
└─
┌─ PostłConteggioTempoStop
  ΔłConteggioTempo
  PrełConteggioTempoStop
└─

— section ConteggioTempoRelational parents ConteggioTempoIntensional ,
ConditionRelational , CentralinaFissaRelational
└─

        . . .

┌─ ℝConteggioTempo
  ℝActivity
  łConteggioTempo
  ΩCondition
  ΩCentralinaFissa
  ΩConteggioTempo
|
  centralina ∈ dom ŁCentralinaFissa
  attesaSecondi ∈ dom ŁCondition
```

└

— section ConteggioTempoExtensional parents ConteggioTempoRelational , ConditionExtensional , CentralinaFissaExtensional

└

. . .

┌─ ConteggioTempoInit

ł ConteggioTempoInit

Δ Đ ConteggioTempo

|

ℝ ConteggioTempo ’

ActivityInit

this? ∈ CONTEGGIO_TEMPO

centralina? ∈ dom Ł CentralinaFissa

attesaSecondi? ∈ dom Ł condition

centralina’ = centralina?

attesaSecondi’ = attesaSecondi?

stop’ = False

└

—

ConteggioTempoNew == ∃ ł ConteggioTempo ’ • ƒ ConteggioTempoN ∧ ConteggioTempoInit[oc!/this?]

└

┌─ ConteggioTempoFin

ł ConteggioTempoFin

Δ Đ ConteggioTempo

|

ℝConteggioTempo

ActivityFin

stop = True

⌙

—

ConteggioTempoDelete == ∃ łConteggioTempo • ƒConteggioTempoD ∧ ConteggioTempoFin

⌙

┌─ PreConteggioTempoRunStop

PrełConteggioTempoRun

ÐConteggioTempo

|

ℝConteggioTempo

PreActivityRun

stop = True

⌙

┌─ PostConteggioTempoRunStop

PostłConteggioTempoRun

ΔÐConteggioTempo

|

ΔℝConteggioTempo

PostActivityRun

executionState′ = TERMINATED

centralina′ = centralina

attesaSecondi′ = attesaSecondi

stop′ = stop

⌙

┌─ PreConteggioTempoRunNotStop

PrełConteggioTempoRun

ÐConteggioTempo

|

ℝConteggioTempo

PreActivityRun

stop = False

└

┌─ PostConteggioTempoRunNotStop

PostłConteggioTempoRun

ΔÐConteggioTempo

|

ΔℝConteggioTempo

PostActivityRun

centralina′ = centralina

attesaSecondi′ = attesaSecondi

stop′ = stop

CentralinaFissaTick[centralina/oc?] ⨟

ConditionAwait[attesaSecondi/oc?,this/a?]

└

—

ConteggioTempoRun == ∃ ΔłConteggioTempo • ƒConteggioTempo ∧

((PreConteggioTempoRunStop ∧ PostConteggioTempoRunStop) ∨

(PreConteggioTempoRunNotStop ∧ PostConteggioTempoRunNotStop))

└

Come si vede l'evento Run di tale attività presenta casi di funzionamento distinti in base al valore di una sua variabile di stato, modificabile dall'esterno.

┌─ PreConteggioTempoStart

PrełConteggioTempoStart

ÐConteggioTempo

|

ℝConteggioTempo

PreActivityStart

└

┌─ PostConteggioTempoStart

PostłConteggioTempoStart

ΔÐConteggioTempo

|

ΔℝConteggioTempo

PostActivityStart

centralina′ = centralina

attesaSecondi′ = attesaSecondi

stop′ = stop

└

—

ConteggioTempoStart == ∃ ΔłConteggioTempo • ƒConteggioTempo ∧ PreConteggioTempoStart ∧ PostConteggioTempoStart

└

┌─ PreConteggioTempoSetExecutionState

PrełConteggioTempoSetExecutionState

ÐConteggioTempo

|

ℝConteggioTempo

PreActivitySetExecutionState

└

┌─ PostConteggioTempoSetExecutionState

PostłConteggioTempoSetExecutionState

ΔÐConteggioTempo

|

ΔℝConteggioTempo

PostActivitySetExecutionState

centralina′ = centralina

attesaSecondi′ = attesaSecondi

stop′ = stop

└

—

ConteggioTempoSetExecutionState == ∃ ΔłConteggioTempo •

ƒConteggioTempo ∧ PreConteggioTempoSetExecutionState ∧

PostConteggioTempoSetExecutionState

└

┌─ PreConteggioTempoGetExecutionState

PrełConteggioTempoGetExecutionState

ÐConteggioTempo

|

ℝConteggioTempo

PreActivityGetExecutionState

└

┌─ PostConteggioTempoGetExecutionState

PostłConteggioTempoGetExecutionState

≡ÐConteggioTempo

|

ΔℝConteggioTempo

PostActivityGetExecutionState

└

—

ConteggioTempoGetExecutionState == ∃ Ξ ł ConteggioTempo •

ƒ ConteggioTempo ∧ PreConteggioTempoGetExecutionState ∧

PostConteggioTempoGetExecutionState

└

┌ PreConteggioTempoStop

PrełConteggioTempoStop

ÐConteggioTempo

|

ℝConteggioTempo

└

┌ PostConteggioTempoStop

PostłConteggioTempoStop

ΔÐConteggioTempo

|

ΔℝConteggioTempo

this′ = this

executionState′ = executionState

centralina′ = centralina

attesaSecondi′ = attesaSecondi

stop′ = True

└

—

ConteggioTempoStop == ∃ ΔłConteggioTempo • ƒ ConteggioTempo ∧

PreConteggioTempoStop ∧ PostConteggioTempoStop

⌙

Questo evento consente di chiedere alla attività di terminare.

4.6.4 Discussione

L'approccio seguito per la modellazione di entità attive comporta alcune conseguenze notevoli.
Generalmente l'uso di entità attive complica la realizzazione di un sistema dal punto di vista pratico, poiché molte strutture dati di quest'ultimo possono essere esposte a inconsistenza a causa dell'uso concorrente da parte di più attività.
Si vuole in questo paragrafo delineare come l'approccio seguito permetta un controllo efficace della concorrenza all'interno del sistema.
In primo luogo si noti che i concetti definiti lasciano non specificato tutto ciò che riguarda le politiche di scheduling delle attività e la sincronizzazione esplicita fra esse qualora si trovino ad operare su risorse comuni.
Ciò che interessa in questa fase è infatti specificare quali sono le entità attive e *cosa* fanno nel sistema. Quali sono le altre entità con cui operano, quando cominciano ad eseguire, quando si sospendono e dove. Questo consente di rendere espliciti i flussi di esecuzione all'interno del sistema e le parti del sistema influenzate da essi. Si noti ad esempio come si sia evitato di definire esplicitamente meccanismi di mutua esclusione.
Il costrutto variabile condizione, nonostante possa essere utilizzato assieme a meccanismi di mutua esclusione, per esprimere la sincronizzazione fra attività, è stato qui introdotto per esplicitare i punti del sistema in cui le attività si sospendono.
L'uso consigliato, in fase di analisi, è dunque quello di consentire il recupero esplicito degli oggetti Condition da parte dell'attività tramite eventi appositi messi a disposizione dalle parti del sistema su cui essa opera. Una volta recuperato un riferimento ad un oggetto Condition su cui necessita di sospendersi, l'attività invocherà esplicitamente l'evento Await su di essa.
Questo accorgimento consente di rendere analizzabile un flusso attivo e il suo effetto all'interno del sistema semplicemente esaminando l'evento Run dell'attività.
Note queste informazioni, in un secondo momento (in fase di progetto), sarà possibile incapsulare le variabili condizione dove è più opportuno (ad esempio all'interno di un *monitor* [22]), predisponendo elementi di sincronizzazione solo dove necessario.

Alla possibile obiezione che in tal modo gli oggetti Condition siano esposti all'esterno pur essendo meccanismi interni, il cui cattivo uso potrebbe determinare malfunzionamenti, si risponde allo stesso modo. Nella fase di analisi siamo interessati a specificare esplicitamente cosa fa il sistema, dunque specifichiamo l'uso che deve essere fatto degli oggetti. Sarà poi il progettista, nella fase successiva, a decidere come e dove incapsulare tali meccanismi.
Un “cattivo uso” degli oggetti Condition nel modello di analisi, indica esclusivamente una specifica errata da parte dell'analista.
Verranno dati più avanti in questo capitolo, esempi circa l'uso consigliato delle variabili condizione in fase di analisi.

4.7 Semantica degli Oggetti

Dovrebbe a questo punto essere chiaro quale sia la semantica attribuita al concetto di oggetto dalla notazione Concepts Z. Ogni oggetto istanza di una certa classe possiede un'identità distinta, corrispondente al valore dell'attributo this. Questa consente, al verificarsi di un evento, di modificare unicamente lo stato di tale oggetto. I possibili stati di un oggetto sono definiti dall'intensione della classe di cui è istanza. Le componenti di un'intensione sono utilizzate per descrivere l'effetto di un evento sull'oggetto. Dunque le componenti dell'intensione modellano le variabili di istanza di un oggetto, mentre l'insieme di eventi che può avvenire su di esso rappresenta la sua interfaccia con l'esterno. Gli eventi sono descritti specificando a fronte di un certo stato corrente e di un certo input, quale sia lo stato futuro e quale sia l'output. Dunque la notazione da la *semantica di un oggetto come state machine*.

4.8 Metafore per esprimere l'interazione

Si è detto che alcune delle macro-parti del sistema individuate dall'analista potrebbero essere distribuite, dunque occorre definire come avvenga la comunicazione fra esse. Si è inoltre sottolineato come per specificare completamente un blocco, occorre descrivere la sua struttura, il suo comportamento e l'interazione con l'esterno. Verranno ora mostrati i concetti messi a disposizione da Concepts Z per l'analisi delle interazioni.

4.8.1 Interazione fra entità concentrate

Con il termine entità concentrate si indicano usualmente due entità collocate sullo stesso nodo fisico di elaborazione. L'interazione fra entità concentrate avviene usualmente tramite invocazione di metodo o chiamata di procedura. Sono stati già mostrati degli esempi di questo tipo di interazione in Concepts Z. Quando un evento implica un altro evento all'interno della parte predicativa della sua postcondizione, si sta specificando l'occorrenza di questo tipo di interazioni. L'usuale meccanismo di Z per il *renaming* delle variabili di uno schema [3], può essere utilizzato per specificare il passaggio di parametri fra le invocazioni di eventi. Si ricordi che l'invocazione di un evento di un'altra classe all'interno della postcondizione di un evento implica una dipendenza da quella classe.

4.8.2 Interazione fra entità distribuite

Con il termine entità distribuite si indicano usualmente due entità collocate su nodi di elaborazione distinti, che dunque comunicano in modo remoto. Questo tipo di comunicazione pone problemi aggiuntivi, dovuti all'utilizzo di reti di telecomunicazione per portare a compimento lo scambio di informazione fra i nodi. Alcuni supporti per la comunicazione distribuita, hanno tentato di ricondurre, con risultati non ottimali, questo tipo di interazione al concetto di invocazione di procedura. Questo tipo di interazione è di tipo sincrono bloccante e va sotto il nome di chiamata di procedura remota (RPC). Questo tipo di scelta produce una uguaglianza sintattica a livello di codice fra le interazioni concentrate e distribuite, ma una difformità della semantica sottostante, la quale apre diverse problematiche. Inoltre, ricondurre la comunicazione remota alla chiamata di procedura, risulta sovra vincolante, in quanto la comunicazione distribuita, consente molte più possibilità della sola interazione sincrona bloccante.
In questo paragrafo, si vogliono introdurre una serie di concetti che consentano di trattare efficacemente la comunicazione distribuita in Concepts Z. Si vuole inoltre mostrare l'efficacia della notazione nell'analisi, portando con essa a compimento la specifica del suo stesso supporto per la comunicazione. Gli esempi successivi mostrano inoltre l'uso consigliato delle variabili condizione nell'analisi.

E' noto che un *middleware* è un layer software che innalza il livello della comunicazione fra due entità, mettendo loro a disposizione determinate astrazioni per l'interazione. Profondi studi sui sistemi distribuiti [23], hanno portato alla luce che le possibili interazioni fra due entità remote, a livello concettuale, sono fondamentalmente di 3 tipi:

Broadcast: un'entità emette un messaggio in modo tale che esso raggiunga il maggior numero possibile di entità comunicanti (il messaggio è indirizzato quindi idealmente a tutte).

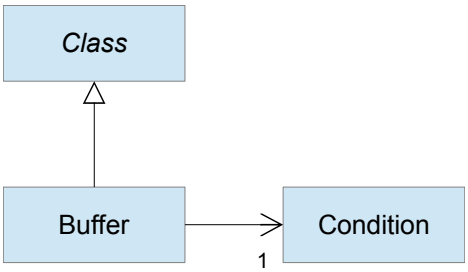
Dispatch: una entità invia un messaggio ad un'altra entità specifica, senza alcuna aspettativa circa la consegna del messaggio o eventuali azioni da parte del ricevente (anche nota come interazione di tipo *fire and forget*).

Request-Reply: una entità invia un messaggio ad un'altra entità specifica con l'aspettativa di ricevere una risposta

Si noti come le RPC sono solo un possibile caso di quest'ultima categoria. Ad esempio non è assolutamente detto che una interazione di questo tipo debba essere bloccante.
Si vuole dunque specificare un middleware che metta a disposizione dei blocchi di un sistema (e dell'analista) queste astrazioni.
Si noti che il middleware ha l'obiettivo di mettere a disposizione i concetti per l'interazione desiderati, non di risolvere problematiche legate alla comunicazione distribuita come ad esempio la perdita di messaggi o la duplicazione di essi. Tale compito è infatti solitamente affidato ad un *protocollo* di comunicazione. La gran parte dei protocolli di comunicazione oggi disponibili (ad esempio TCP) affronta già con relativo successo queste problematiche. Nel condurre l'analisi si farà dunque l'assunzione che il middleware, per la trasmissione dei messaggi, disponga di un canale di comunicazione che utilizza un protocollo adeguato alle necessità applicative.
Una seconda assunzione relativa alla analisi che si andrà a condurre, è la correttezza d'uso delle astrazioni fornite dal middleware. In altre parole si assume che le entità comunicanti utilizzino consistentemente le astrazioni di comunicazione messe a disposizione. Quindi date due entità comunicanti, una effettuerà azioni specifiche di una delle categorie dette sopra e l'altra le operazioni duali corrette della categoria, per portare

a termine l'interazione (non è ad esempio possibile rispondere ad una request con un dispaccio). Diversamente da ciò infatti, l'analista avrebbe prodotto un modello che esprime in modo errato la dimensione interazione.

4.8.2.1 La nozione di Buffer



Poiché si vuol essere in grado di scambiare informazione fra nodi distinti, la prima astrazione di cui si necessita è senza dubbio quella di un *buffer*, dove sia possibile deporre l'informazione in attesa di essere inviata e l'informazione in attesa di essere recuperata da un'entità ricevente.

```
— section AnalysisConcepts parents BaseConcepts
└
        . . .
|
  Buffer : CLASS
  BUFFER : ℙ INSTANCE
|
  isA(Buffer) = Class
  extension⦇{Buffer}⦈ = BUFFER
  ¬ abstract Buffer
└
```

— section BufferIntensional parents ClassIntensional

└

Un Buffer è definito come una classe parametrica sfruttando il supporto di Z per i tipi generici. Riferisce una Condition dove sarà possibile, per le attività che usano il buffer, sospendersi.

┌═ łBuffer [X]
 łClass
 dataWait : CONDITION
 dataQueue : seq X
|
 this ∈ BUFFER
└

┌═ łBufferInit [X]
 łClassInit
 łBuffer[X]’
 condition? : CONDITION
└

┌═ łBufferFin [X]
 łClassFin
 łBuffer[X]
└

GetDataWait consente alle attività di recuperare l'oggetto Condition associato al buffer.

┌═ PrełBufferGetDataWait [X]
 łBuffer[X]
└

PostłBufferGetDataWait [X]

ΞłBuffer[X]

PrełBufferGetDataWait

dataWait! : CONDITION

PrełBufferGetDataQueue [X]

łBuffer[X]

PostłBufferGetDataQueue [X]

ΞłBuffer[X]

PrełBufferGetDataQueue

dataQueue! : seq X

Enqueue immette informazione nel buffer ed implica il risveglio di una attività in attesa di dati.

PrełBufferEnqueue [X]

łBuffer[X]

data? : X

PostłBufferEnqueue [X]

ΔłBuffer[X]

PrełBufferEnqueue

Dequeue è l'operazione di estrazione di un dato dal buffer. Restituisce un insieme vuoto se non ci sono dati o la testa della lista nel caso ve ne siano. Ciò significa che l'operazione di estrazione è sempre eseguibile ed ha sempre successo, qualunque sia lo stato del buffer.

PrełBufferDequeue [X]

łBuffer[X]

PostłBufferDequeue [X]

ΔłBuffer[X]

PrełBufferDequeue

data! : X

— section BufferRelational parents BufferIntensional , ConditionRelational

. . .

ℝBuffer [X]

ℝClass

łBuffer[X]

ΩCondition

ΩBuffer[X]

dataWait ∈ dom ŁCondition

— section BufferExtensional parents BufferRelational , ConditionExtensional

. . .

ĐBuffer [X]

ÐCondition

ÆBuffer[X]

|

BufferIsClass

└

. . .

┌═ BufferInit [X]

łBufferInit[X]

ΔÐBuffer

|

ℝBuffer ’

ClassInit

this? ∈ BUFFER

condition? ∈ dom ŁCondition

dataWait’ = condition?

dataQueue’ = ⟨ ⟩

└

—

BufferNew [X] == ∃ łBuffer[X] ’ • ƒBufferN ∧ BufferInit[ob!/this?]

└

┌═ BufferFin [X]

łBufferFin[X]

ΔÐBuffer

|

ℝBuffer

ClassFin

#(ŁCondition dataWait).activityQueue = 0

└

—

BufferDelete [X] == ∃ łBuffer[X] • ƒBufferD ∧ BufferFin

└

┌═ PreBufferGetDataWait [X]

PrełBufferGetDataWait[X]

ĐBuffer

|

ℝBuffer

└

┌═ PostBufferGetDataWait [X]

PostłBufferGetDataWait[X]

ΞĐBuffer

|

ΔℝBuffer

dataWait! = dataWait

└

—

BufferGetDataWait [X] == ∃ ΞłBuffer[X] • ƒBuffer ∧ PreBufferGetDataWait ∧ PostBufferGetDataWait

└

┌═ PreBufferGetDataQueue [X]

PrełBufferGetDataQueue[X]

ĐBuffer

|

ℝBuffer

└

┌═ PostBufferGetDataQueue [X]

PostłBufferGetDataQueue[X]

ΞŧÐBuffer

|

ΔℝBuffer

dataQueue! = dataQueue

└

—

BufferGetDataQueue [X] == ∃ ΞŧłBuffer[X] • ƒBuffer ∧ PreBufferGetDataQueue ∧ PostBufferGetDataQueue

└

┌═ PreBufferEnqueue [X]

PrełBufferEnqueue[X]

ÐBuffer

|

ℝBuffer

└

┌═ PostBufferEnqueue [X]

PostłBufferEnqueue[X]

ΔÐBuffer

|

ΔℝBuffer

this′ = this

dataWait′ = dataWait

dataQueue′ = dataQueue ⌢ ⟨ data? ⟩

ConditionSignal[dataWait/oc?]

└

—

BufferEnqueue [X] == ∃ ΔłBuffer[X] • ƒBuffer ∧ PreBufferEnqueue ∧ PostBufferEnqueue

└

┌═ PreBufferDequeue [X]

PrełBufferDequeue[X]

ĐBuffer

|

ℝBuffer

└

┌═ PostBufferDequeue [X]

PostłBufferDequeue[X]

ΔĐBuffer

|

ΔℝBuffer

this’ = this

dataWait’ = dataWait

dataQueue = ⟨ ⟩ ⇒ dataQueue’ = dataQueue ∧ data! = Ø

dataQueue ≠ ⟨ ⟩ ⇒ dataQueue’ = tail dataQueue ∧ data! = {head dataQueue}

└

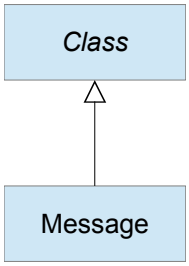
—

BufferDequeue [X] == ∃ ΔłBuffer[X] • ƒBuffer ∧ PreBufferDequeue ∧ PostBufferDequeue

└

4.8.2.2 La nozione di Message

L'informazione scambiata fra i nodi sarà contenuta in un *messaggio*



— section AnalysisConcepts parents BaseConcepts

└

. . .

Si vuole poter inviare messaggi sia a entità specifiche, sia a tutte le entità. Si definisce dunque l'universo dei possibili identificatori di entità e in base ad esso l'universo degli indirizzi, che comprende gli identificatori di entità più un indirizzo speciale che le denota tutte.

— [ENTITY_ID_SPACE] └

—

ENTITYID ::= eAll | eId 《 ENTITY_ID_SPACE 》

└

Oltre a voler indirizzare entità, si vogliono poter denotare diversi punti di accesso all'interno di esse (*end point*). Dunque un messaggio sarà indirizzato ad una certa entità e ad un certo *end number,* un numero naturale che identifica un punto di accesso ad una entità. Su un certo end point può essere in ascolto una sola entità, dal quale essa invia e riceve messaggi. L'unica l'eccezione è l'end point da cui vengono emessi e a cui vengono recapitati i messaggi broadcast, che per convenzione sarà indicato con il numero 0.

Analogamente definiamo l'universo dei simboli che possono codificare il contenuto di un

messaggio.

```
— [ SYMBOL ] └─
```

```
┌
  Message : CLASS
  MESSAGE : ℙ INSTANCE
├
  isA(Message) = Class
  extension⦇{Message}⦈ = MESSAGE
  ¬ abstract Message
└─
```

```
— section MessageIntensional parents ClassIntensional
└─
```

Un messaggio contiene informazioni circa il mittente e l'end point di origine, il numero progressivo dei messaggi inviati tramite quell'end point a una certa destinazione e un identificatore di richiesta. Analogamente vi sono informazioni circa il ricevente e l'end point di destinazione a cui consegnare il messaggio. Infine compare il contenuto del messaggio, inteso come una sequenza di simboli appartenenti all'alfabeto SYMBOL.
Fra gli invarianti viene asserito che l'indirizzo del mittente non può essere l'indirizzo che denota tutte le entità (eAll) e che qualora l'end point di origine sia quello adibito all'emissione dei messaggi broadcast (ossia 0), ricevente ed end point di destinazione sono necessariamente tutte le entità e lo stesso end point di broadcast.

```
┌─ ⦇Message
  ⦇Class
  sender : ENTITYID
  senderEnd : ℕ
  sequenceNumber : ℕ
```

reqId : ℕ

receiver : ENTITYID

receiverEnd : ℕ

data : seq SYMBOL

|

this ∈ MESSAGE

sender ≠ eAll

senderEnd = 0 ⇒ receiver = eAll ∧ receiverEnd = 0

└

Un messaggio una volta costruito non è più modificabile. Esso mette quindi a disposizione un costruttore che consente di specificare il valore di tutti i campi ed i relativi eventi di lettura per poterli recuperare.

┌ ł MessageInit

ł ClassInit

ł Message ’

sender? : ENTITYID

senderEnd? : ℕ

sequenceNumber? : ℕ

reqId? : ℕ

receiver? : ENTITYID

receiverEnd? : ℕ

data? : seq SYMBOL

└

┌ ł MessageFin

ł ClassFin

ł Message

└

┌─ PrełMessageGetSender
 łMessage
└

┌─ PostłMessageGetSender
 ΞłMessage
 PrełMessageGetSender
 sender! : ENTITYID
└

┌─ PrełMessageGetSenderEnd
 łMessage
└

┌─ PostłMessageGetSenderEnd
 ΞłMessage
 PrełMessageGetSenderEnd
 senderEnd! : ℕ
└

┌─ PrełMessageGetSequenceNumber
 łMessage
└

┌─ PostłMessageGetSequenceNumber
 ΞłMessage
 PrełMessageGetSequenceNumber
 sequenceNumber! : ℕ
└

┌─ PrełMessageGetReqId
 łMessage
└

┌─ PostłMessageGetReqId
 ΞłMessage
 PrełMessageGetReqId
 reqId! : ℕ
└

┌─ PrełMessageGetReceiver
 łMessage
└

┌─ PostłMessageGetReceiver
 ΞłMessage
 PrełMessageGetReceiver
 receiver! : ENTITYID
└

┌─ PrełMessageGetReceiverEnd
 łMessage
└

┌─ PostłMessageGetReceiverEnd
 ΞłMessage
 PrełMessageGetReceiverEnd
 receiverEnd! : ℕ
└

┌─ PrełMessageGetData

```
 łMessage
└
┌─ PostłMessageGetData
 ΞłMessage
 PrełMessageGetData
 data! : seq SYMBOL
└

— section MessageRelational parents MessageIntensional , ClassRelational
└
        . . .

┌─ ℝMessage
 ℝClass
 łMessage
 ΩMessage
└

— section MessageExtensional parents MessageRelational , ClassExtensional
└
        . . .

┌─ ÐMessage
 ÐClass
 ÆMessage
|
 MessageIsClass
```

└

. . .

┌ MessageInit

łMessageInit

ΔĐMessage

|

ℝMessage ’

ClassInit

this? ∈ MESSAGE

sender? ≠ eAll

senderEnd? = 0 ⇒ receiver? = eAll ∧ receiverEnd? = 0

sender’ = sender?

senderEnd’ = senderEnd?

sequenceNumber’ = sequenceNumber?

reqId’ = reqId?

receiver’ = receiver?

receiverEnd’ = receiverEnd?

data’ = data?

└

—

MessageNew == ∃ łMessage ’ • ƒMessageN ∧ MessageInit[om!/this?]

└

┌ MessageFin

łMessageFin

ΔĐMessage

|

ℝMessage

ClassFin

└

—

MessageDelete == ∃ łMessage • ƒMessageD ∧ MessageFin

└

┌ PreMessageGetSender

PrełMessageGetSender

ÐMessage

|

ℝMessage

└

┌ PostMessageGetSender

PostłMessageGetSender

ΞÐMessage

|

ΔℝMessage

sender! = sender

└

—

MessageGetSender == ∃ ΞłMessage • ƒMessage ∧ PreMessageGetSender ∧ PostMessageGetSender

└

┌ PreMessageGetSenderEnd

PrełMessageGetSenderEnd

ĐMessage

|

ℝMessage

└

┌─ PostMessageGetSenderEnd

PostłMessageGetSenderEnd

ΞĐMessage

|

ΔℝMessage

senderEnd! = senderEnd

└

—

MessageGetSenderEnd == ∃ ΞłMessage • ƒMessage ∧ PreMessageGetSenderEnd ∧ PostMessageGetSenderEnd

└

┌─ PreMessageGetSequenceNumber

PrełMessageGetSequenceNumber

ĐMessage

|

ℝMessage

└

┌─ PostMessageGetSequenceNumber

PostłMessageGetSequenceNumber

ΞĐMessage

|

ΔℝMessage

sequenceNumber! = sequenceNumber

└

—

MessageGetSequenceNumber == ∃ ΞłMessage • ƒMessage ∧ PreMessageGetSequenceNumber ∧ PostMessageGetSequenceNumber

└

┌ PreMessageGetReqId

PrełMessageGetReqId

ĐMessage

|

ℝMessage

└

┌ PostMessageGetReqId

PostłMessageGetReqId

ΞĐMessage

|

ΔℝMessage

reqId! = reqId

└

—

MessageGetReqId == ∃ ΞłMessage • ƒMessage ∧ PreMessageGetReqId ∧ PostMessageGetReqId

└

┌ PreMessageGetReceiver

PrełMessageGetReceiver

ĐMessage

|

ℝMessage

└

┌ PostMessageGetReceiver

PostłMessageGetReceiver

ΞŧĐMessage

|

ΔℝMessage

receiver! = receiver

└

—

MessageGetReceiver == ∃ ΞłMessage • ƒMessage ∧ PreMessageGetReceiver

∧ PostMessageGetReceiver

└

┌ PreMessageGetReceiverEnd

PrełMessageGetReceiverEnd

ĐMessage

|

ℝMessage

└

┌ PostMessageGetReceiverEnd

PostłMessageGetReceiverEnd

ΞŧĐMessage

|

ΔℝMessage

receiverEnd! = receiverEnd

└

—

MessageGetReceiverEnd == ∃ Ξ‡Message • ƒMessage ∧
PreMessageGetReceiverEnd ∧ PostMessageGetReceiverEnd

└

┌ PreMessageGetData

PrełMessageGetData

ĐMessage

|

ℝMessage

└

┌ PostMessageGetData

PostłMessageGetData

Ξ‡ĐMessage

|

ΔℝMessage

data! = data

└

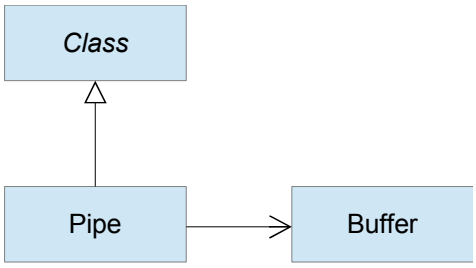
—

MessageGetData == ∃ Ξ‡Message • ƒMessage ∧ PreMessageGetData ∧
PostMessageGetData

└

4.8.2.3 La nozione di Pipe

Una pipe rappresenta un contenitore sequenziale di informazioni che può essere chiuso. Una volta chiusa una pipe. Non è più possibile depositare informazioni all'interno di essa.



— section AnalysisConcepts parents BaseConcepts

└

. . .

|

Pipe : CLASS

PIPE : ℙ INSTANCE

|

isA(Pipe) = Class

extension(⦇{Pipe}⦈) = PIPE

¬ abstract Pipe

└

— section PipeIntensional parents ClassIntensional

└

┌ ł Pipe

łClass

isClosed : □□

buffer : BUFFER

|

this ∈ PIPE

└

┌─ łPipeInit
 łClassInit
 łPipe ’
└

┌─ łPipeFin
 łClassFin
 łPipe
└

┌─ PrełPipeIsClosed
 łPipe
└
┌─ PostłPipeIsClosed
 ≡łPipe
 PrełPipeIsClosed
 isClosed! : □□
└

┌─ PrełPipeGetDataWait
 łPipe
└
┌─ PostłPipeGetDataWait
 ≡łPipe
 PrełPipeGetDataWait
 dataWait! : CONDITION
└

PrełPipeGetDataQueue

łPipe

PostłPipeGetDataQueue [X]

Ξ łPipe

PrełPipeGetDataQueue

dataQueue! : seq X

PrełPipeEnqueue [X]

łPipe

data? : X

PostłPipeEnqueue [X]

Δ łPipe

PrełPipeEnqueue[X]

PrełPipeDequeue

łPipe

PostłPipeDequeue [X]

Δ łPipe

PrełPipeDequeue

data! : □□X

```
┌─ PrełPipeClose
 łPipe
└─
┌─ PostłPipeClose
 ΔłPipe
 PrełPipeClose
└─
```

```
— section PipeRelational parents PipeIntensional , BufferRelational
└─
```

. . .

```
┌═ ℝPipe [ X ]
 ℝClass
 łPipe
 ΩBuffer[X]
 ΩPipe
|
 buffer ∈ dom ŁBuffer
└─
```

```
— section PipeExtensional parents PipeRelational , BufferExtensional
└─
```

. . .

```
┌═ ĐPipe [ X ]
 ĐBuffer[X]
 ÆPipe
```

|

PipeIsClass

└

. . .

Una pipe appena creata con l'evento PipeNew è aperta.

┌═ PipeInit [X]

łPipeInit

ΔĐPipe[X]

|

ℝPipe ’

ClassInit

this? ∈ PIPE

isClosed’ = False

∃ oc! : CONDITION

• ConditionNew ⨟

BufferNew[buffer’/ob!,oc!/condition?]

└

—

PipeNew [X] == ∃ łPipe ’ • ƒPipeN[X] ∧ PipeInit[op!/this?]

└

┌═ PipeFin [X]

łPipeFin

ΔĐPipe[X]

|

ℝPipe

ClassFin

isClosed = True

#(ŁCondition (ŁBuffer buffer).dataWait).activityQueue = 0

└

—

PipeDelete [X] == ∃ łPipe • ƒPipeD[X] ∧ PipeFin

└

┌═ PrePipeIsClosed [X]

PrełPipeIsClosed

ĐPipe[X]

|

ℝPipe

└

┌═ PostPipeIsClosed [X]

PostłPipeIsClosed

ΞĐPipe[X]

|

ΔℝPipe

isClosed! = isClosed

└

—

PipeIsClosed [X] == ∃ ΞłPipe • ƒPipe[X] ∧ PrePipeIsClosed ∧ PostPipeIsClosed

└

┌═ PrePipeGetDataWait [X]

PrełPipeGetDataWait

ĐPipe[X]

|

ℝPipe

└

┌═ PostPipeGetDataWait [X]

PostłPipeGetDataWait

ΞĐPipe[X]

|

ΔℝPipe

BufferGetDataWait[buffer/ob?]

└

—

PipeGetDataWait [X] == ∃ ΞłPipe • ƒPipe[X] ∧ PrePipeGetDataWait ∧ PostPipeGetDataWait

└

┌═ PrePipeGetDataQueue [X]

PrełPipeGetDataQueue

ĐPipe[X]

|

ℝPipe

└

┌═ PostPipeGetDataQueue [X]

PostłPipeGetDataQueue[X]

ΞĐPipe

|

ΔℝPipe

BufferGetDataQueue[buffer/ob?]

└

—

PipeGetDataQueue [X] == ∃ ΞłPipe • ƒPipe[X] ∧ PrePipeGetDataQueue ∧ PostPipeGetDataQueue

└

L'operazione di immissione di dati nella pipe Enqueue è possibile fin tanto che la Pipe resta aperta.

┌═ PrePipeEnqueue [X]

PrełPipeEnqueue[X]

ĐPipe[X]

|

ℝPipe

isClosed = False

└

┌═ PostPipeEnqueue [X]

PostłPipeEnqueue[X]

ΔĐPipe

|

ΔℝPipe

this′ = this

isClosed′ = isClosed

buffer′ = buffer

BufferEnqueue[buffer/ob?]

└

—

PipeEnqueue [X] == ∃ ΔłPipe • ƒPipe[X] ∧ PrePipeEnqueue ∧ PostPipeEnqueue

└

L'operazione di prelievo di un dato dalla pipe Dequeue, è sempre eseguibile ma può restituire un insieme vuoto qualora non vi sia informazione in essa.

┌═ PrePipeDequeue [X]

PrełPipeDequeue

ĐPipe[X]

|

ℝPipe

└

┌═ PostPipeDequeue [X]

PostłPipeDequeue[X]

ΔĐPipe[X]

|

ΔℝPipe

this′ = this

isClosed′ = isClosed

buffer′ = buffer

#data! ∈ 0 .. 1

BufferDequeue[buffer/ob?]

└

—

PipeDequeue [X] == ∃ ΔłPipe • ƒPipe[X] ∧ PrePipeDequeue ∧ PostPipeDequeue

└

Si noti come l'operazione di chiusura Close svegli tutti coloro che sono in attesa di acquisire informazioni dalla pipe.

┌═ PrePipeClose [X]

PrełPipeClose

ĐPipe[X]

|

ℝPipe

└

┌═ PostPipeClose [X]

PostłPipeClose

ΔĐPipe[X]

|

ΔℝPipe

this′ = this

isClosed′ = True

buffer′ = buffer

∃ dataWait! : CONDITION

• BufferGetDataWait[buffer/ob?] ⨟

ConditionSignalAll[dataWait!/oc?]

└

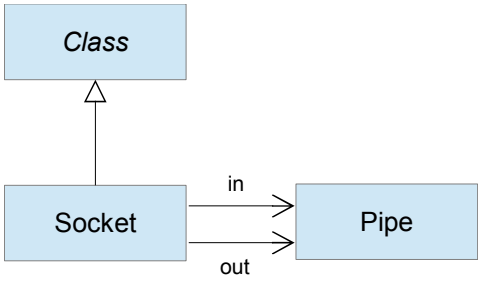
—

PipeClose [X] == ∃ ΔłPipe • ƒPipe[X] ∧ PrePipeClose ∧ PostPipeClose

└

4.8.2.4 La nozione di Socket

Una socket rappresenta un end point di comunicazione di una entità, da cui vengono inviati e ricevuti messaggi. Una entità comunicante utilizza la socket tramite gli eventi Send e Receive, mentre il supporto che si occupa della trasmissione estrae i messaggi dal canale di uscita tramite l'evento Get e deposita messaggi nel canale di ingresso tramite l'evento Put.



— section AnalysisConcepts parents BaseConcepts

└

. . .

|

Socket : CLASS

SOCKET : ℙ INSTANCE

|

isA(Socket) = Class

extension(⦇{Socket}⦈) = SOCKET

¬ abstract Socket

└

— section SocketIntensional parents ClassIntensional

└

Una Socket è associata ad un'entità e ad un certo end number e utilizza due Pipe per i suoi canali di invio e ricezione.

┌─ łSocket

łClass

owner : ENTITYID

endNumber : ℕ

in : PIPE

out : PIPE

|

this ∈ SOCKET

in ≠ out

owner ≠ eAll

└

┌ łSocketInit

łClassInit

łSocket ’

owner? : ENTITYID

endNumber? : ℕ

└

┌ łSocketFin

łClassFin

łSocket

└

┌ PrełSocketSend

łSocket

m? : MESSAGE

└

┌ PostłSocketSend

ΔłSocket

PrełSocketSend

└

┌ PrełSocketReceive

łSocket

└

┌─ PostłSocketReceive

ΔłSocket

PrełSocketReceive

m! : □□MESSAGE

└

┌─ PrełSocketGet

łSocket

└

┌─ PostłSocketGet

ΔłSocket

PrełSocketGet

m! : □□MESSAGE

└

┌─ PrełSocketPut

łSocket

m? : MESSAGE

└

┌─ PostłSocketPut

ΔłSocket

PrełSocketPut

└

┌─ PrełSocketGetOwner

łSocket

```
└
┌─ PostłSocketGetOwner
  ΞłSocket
  PrełSocketGetOwner
  owner! : ENTITYID
└

┌─ PrełSocketGetEndNumber
  łSocket
└
┌─ PostłSocketGetEndNumber
  ΞłSocket
  PrełSocketGetEndNumber
  endNumber! : ℕ
└

┌─ PrełSocketGetInWait
  łSocket
└
┌─ PostłSocketGetInWait
  ΞłSocket
  PrełSocketGetInWait
  inWait! : CONDITION
└

┌─ PrełSocketGetOutWait
  łSocket
└
```

┌─ PostłSocketGetOutWait
  ΞłSocket
  PrełSocketGetOutWait
  outWait! : CONDITION
└─

┌─ PrełSocketIsInClosed
  łSocket
└─
┌─ PostłSocketIsInClosed
  ΞłSocket
  PrełSocketIsInClosed
  isInClosed! : □□
└─

┌─ PrełSocketIsOutClosed
  łSocket
└─
┌─ PostłSocketIsOutClosed
  ΞłSocket
  PrełSocketIsOutClosed
  isOutClosed! : □□
└─

┌─ PrełSocketGetInQueue
  łSocket
└─
┌─ PostłSocketGetInQueue

Ξ łSocket
PrełSocketGetInQueue
inQueue! : iseq MESSAGE
└

┌ PrełSocketGetOutQueue
łSocket
└
┌ PostłSocketGetOutQueue
Ξ łSocket
PrełSocketGetOutQueue
outQueue! : iseq MESSAGE
└

┌ PrełSocketCloseIn
łSocket
└
┌ PostłSocketCloseIn
Δ łSocket
PrełSocketCloseIn
└

┌ PrełSocketCloseOut
łSocket
└
┌ PostłSocketCloseOut
Δ łSocket
PrełSocketCloseOut

└

— section SocketRelational parents SocketIntensional , PipeRelational , MessageRelational

└

. . .

Negli invarianti relazionali viene asserito che le due pipe non sono mai nulle e che lo stesso vale per il loro contenuto. Inoltre si dice che la stessa istanza di messaggio può essere presente una volta sola in una pipe e che le due pipe non hanno messaggi in comune. Si dice infine che i messaggi inviati e ricevuti dalle pipe deve essere coerenti, per quanto riguarda l'indirizzamento, con l'entità e l'end point a cui la Socket è associata.

┌─ ℝSocket

ℝClass

łSocket

ΩBuffer[MESSAGE]

ΩMessage

ΩPipe

ΩSocket

|

in ∈ dom łPipe

out ∈ dom łPipe

ran (łBuffer (łPipe in).buffer).dataQueue ∪ ran (łBuffer (łPipe out).buffer).dataQueue ⊆ dom łMessage

(łBuffer (łPipe in).buffer).dataQueue ∈ iseq MESSAGE

(łBuffer (łPipe out).buffer).dataQueue ∈ iseq MESSAGE

ran (łBuffer (łPipe in).buffer).dataQueue ∩ ran (łBuffer (łPipe out).buffer).dataQueue = Ø

∀ m : ran (łBuffer (łPipe in).buffer).dataQueue

• (endNumber > 0 ⇒ (ŁMessage m).receiver = owner) ∧
(endNumber = 0 ⇒ (ŁMessage m).senderEnd = 0) ∧
(ŁMessage m).receiverEnd = endNumber
∀ m : ran (ŁBuffer (ŁPipe out).buffer).dataQueue • (ŁMessage m).sender = owner ∧ (ŁMessage m).senderEnd = endNumber
└

— section SocketExtensional parents SocketRelational , PipeExtensional , MessageExtensional
└

. . .

┌ ĐSocket
ĐMessage
ĐPipe[MESSAGE]
ÆSocket
|
SocketIsClass
└

. . .

┌ SocketInit
łSocketInit
ΔĐSocket
|
ℝSocket ’
ClassInit

this? ∈ SOCKET

owner? ≠ eAll

owner′ = owner?

endNumber′ = endNumber?

PipeNew[in′/op!] ⨟

PipeNew[out′/op!]

└

—

SocketNew == ∃ łSocket ′ • ƒSocketN ∧ SocketInit[os!/this?]

└

┌─ SocketFin

łSocketFin

ΔĐSocket

|

ℝSocket

ClassFin

(ŁPipe in).isClosed = True

#(ŁCondition (ŁBuffer (ŁPipe in).buffer).dataWait).activityQueue = 0

(ŁPipe out).isClosed = True

#(ŁCondition (ŁBuffer (ŁPipe out).buffer).dataWait).activityQueue = 0

└

—

SocketDelete == ∃ łSocket • ƒSocketD ∧ SocketFin

└

Come si vede dalla precondizione del metodo Send, non è possibile inviare messaggi da una socket il cui canale d'uscita è stato chiuso.

┌─ PreSocketSend

PreŁSocketSend

ĐSocket

|

ℝSocket

m? ∈ dom ŁMessage

m? ∉ ran (ŁBuffer (ŁPipe out).buffer).dataQueue ∪ ran (ŁBuffer (ŁPipe in).buffer).dataQueue

(ŁPipe out).isClosed = False

(ŁMessage m?).sender = owner

(ŁMessage m?).senderEnd = endNumber

└

┌ PostSocketSend

PostŁSocketSend

ΔĐSocket

|

ΔℝSocket

this′ = this

owner′ = owner

endNumber′ = endNumber

in′ = in

out′ = out

PipeEnqueue[out/op?,m?/data?]

└

—

SocketSend == ∃ ΔŁSocket • ƒSocket ∧ PreSocketSend ∧ PostSocketSend

└

┌ PreSocketReceive

```
  PrełSocketReceive
  ÐSocket
|
  ℝSocket
└
┌─ PostSocketReceive
  PostłSocketReceive
  ΔÐSocket
|
  ΔℝSocket
  this′ = this
  owner′ = owner
  endNumber′ = endNumber
  in′ = in
  out′ = out
  #m! ∈ 0 .. 1
  PipeDequeue[in/op?,m!/data!]
└
—
  SocketReceive == ∃ ΔłSocket • ƒSocket ∧ PreSocketReceive ∧
PostSocketReceive
└

┌─ PreSocketGet
  PrełSocketGet
  ÐSocket
|
  ℝSocket
```

└

┌─ PostSocketGet

PostłSocketGet

ΔĐSocket

|

ΔℝSocket

this′ = this

owner′ = owner

endNumber′ = endNumber

in′ = in

out′ = out

#m! ∈ 0 .. 1

PipeDequeue[out/op?,m!/data!]

└

—

SocketGet == ∃ ΔłSocket • ƒSocket ∧ PreSocketGet ∧ PostSocketGet

└

Analogamente non è possibile inserire messaggi nel canale di ingresso di una Socket (Put) dopo che esso è stato chiuso.

┌─ PreSocketPut

PrełSocketPut

ĐSocket

|

ℝSocket

m? ∈ dom ŁMessage

m? ∉ ran (ŁBuffer (ŁPipe in).buffer).dataQueue ∪ ran (ŁBuffer (ŁPipe out).buffer).dataQueue

(ŁPipe in).isClosed = False

```
  endNumber > 0 ⇒ (ŁMessage m?).receiver = owner
  endNumber = 0 ⇒ (ŁMessage m?).senderEnd = 0
  (ŁMessage m?).receiverEnd = endNumber
└
┌─ PostSocketPut
  PostłSocketPut
  ΔĐSocket
|
  ΔℝSocket
  this' = this
  owner' = owner
  endNumber' = endNumber
  in' = in
  out' = out
  PipeEnqueue[in/op?,m?/data?]
└
—
  SocketPut == ∃ ΔłSocket • ƒSocket ∧ PreSocketPut ∧ PostSocketPut
└


┌─ PreSocketGetOwner
  PrełSocketGetOwner
  ĐSocket
|
  ℝSocket
└
┌─ PostSocketGetOwner
  PostłSocketGetOwner
```

Ξ ĐSocket

|

ΔℝSocket

owner! = owner

└

—

SocketGetOwner == ∃ Ξ łSocket • ƒSocket ∧ PreSocketGetOwner ∧ PostSocketGetOwner

└

┌ PreSocketGetEndNumber

PrełSocketGetEndNumber

ĐSocket

|

ℝSocket

└

┌ PostSocketGetEndNumber

PostłSocketGetEndNumber

Ξ ĐSocket

|

ΔℝSocket

endNumber! = endNumber

└

—

SocketGetEndNumber == ∃ Ξ łSocket • ƒSocket ∧ PreSocketGetEndNumber ∧ PostSocketGetEndNumber

└

I due eventi successivi GetInWait e GetOutWait, consentono di recuperare le variabili condizione associate alla Socket.

```
┌─ PreSocketGetInWait
  PrełSocketGetInWait
  ĐSocket
|
  ℝSocket
└─
┌─ PostSocketGetInWait
  PostłSocketGetInWait
  ΞĐSocket
|
  ΔℝSocket
  PipeGetDataWait[in/op?,inWait!/dataWait!]
└─
─
  SocketGetInWait == ∃ ΞłSocket • ƒSocket ∧ PreSocketGetInWait ∧
PostSocketGetInWait
└─


┌─ PreSocketGetOutWait
  PrełSocketGetOutWait
  ĐSocket
|
  ℝSocket
└─
┌─ PostSocketGetOutWait
  PostłSocketGetOutWait
  ΞĐSocket
```

|

ΔℝSocket

PipeGetDataWait[out/op?,outWait!/dataWait!]

└

—

SocketGetOutWait == ∃ ΞłSocket • ƒSocket ∧ PreSocketGetOutWait ∧ PostSocketGetOutWait

└

┌─ PreSocketIsInClosed

PrełSocketIsInClosed

ĐSocket

|

ℝSocket

└

┌─ PostSocketIsInClosed

PostłSocketIsInClosed

ΞĐSocket

|

ΔℝSocket

PipeIsClosed[in/op?,isInClosed!/isClosed!]

└

—

SocketIsInClosed == ∃ ΞłSocket • ƒSocket ∧ PreSocketIsInClosed ∧ PostSocketIsInClosed

└

┌─ PreSocketIsOutClosed

PrełSocketIsOutClosed

ĐSocket

|

ℝSocket

└

┌─ PostSocketIsOutClosed

PostłSocketIsOutClosed

ΞĐSocket

|

ΔℝSocket

PipeIsClosed[out/op?,isOutClosed!/isClosed!]

└

—

SocketIsOutClosed == ∃ ΞłSocket • ƒSocket ∧ PreSocketIsOutClosed ∧ PostSocketIsOutClosed

└

┌─ PreSocketGetInQueue

PrełSocketGetInQueue

ĐSocket

|

ℝSocket

└

┌─ PostSocketGetInQueue

PostłSocketGetInQueue

ΞĐSocket

|

ΔℝSocket

PipeGetDataQueue[in/op?,inQueue!/dataQueue!]

└

—

SocketGetInQueue == ∃ Ξ'Socket • ƒSocket ∧ PreSocketGetInQueue ∧ PostSocketGetInQueue

└

┌ PreSocketGetOutQueue

PreSocketGetOutQueue

ĐSocket

|

ℝSocket

└

┌ PostSocketGetOutQueue

PostSocketGetOutQueue

Ξ'ĐSocket

|

Δ'ℝSocket

PipeGetDataQueue[out/op?,outQueue!/dataQueue!]

└

—

SocketGetOutQueue == ∃ Ξ'Socket • ƒSocket ∧ PreSocketGetOutQueue ∧ PostSocketGetOutQueue

└

┌ PreSocketCloseIn

PreSocketCloseIn

ĐSocket

|

ℝSocket

└

┌ PostSocketCloseIn

PostłSocketCloseIn

ΔÐSocket

|

ΔℝSocket

this′ = this

owner′ = owner

endNumber′ = endNumber

in′ = in

out′ = out

PipeClose[in/op?]

└

—

SocketCloseIn == ∃ ΞłSocket • ƒSocket ∧ PreSocketCloseIn ∧

PostSocketCloseIn

└

┌ PreSocketCloseOut

PrełSocketCloseOut

ÐSocket

|

ℝSocket

└

┌ PostSocketCloseOut

PostłSocketCloseOut

ΔÐSocket

|

ΔℝSocket

this′ = this

owner′ = owner

endNumber′ = endNumber

in′ = in

out′ = out

PipeClose[out/op?]

⌙

—

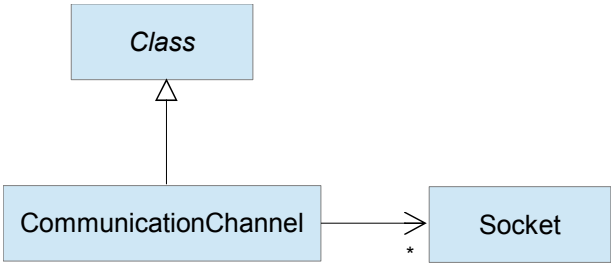
SocketCloseOut == ∃ ☲łSocket • ƒSocket ∧ PreSocketCloseOut ∧ PostSocketCloseOut

⌙

Questi due metodi consentono di chiudere selettivamente il canale d'ingresso e di uscita di una socket.

4.8.2.5 La nozione di CommunicationChannel

Un canale di comunicazione rappresenta un supporto in grado di spostare messaggi dalle code di uscita delle socket alle loro code di ingresso. Per non complicare inutilmente la trattazione, si darà una specifica molto astratta di questo componente, specificando semplicemente che il comportamento atteso da esso è di tipo *best-effort*, ossia se il ricevente di un messaggio da recapitare non è disponibile, oppure la coda di ingresso della rispettiva socket risulta chiusa, il messaggio viene gettato dal canale. Si noti che la scelta di una specifica così astratta non risulta vincolante, in quanto qualunque realizzazione di questo componente che soddisfi quanto specificato, è adatta a fungere da supporto a tutti i concetti fin'ora definiti e che verranno definiti in seguito. Attualmente molti middleware per la comunicazione soddisfano già questa specifica[23].




```
— section AnalysisConcepts parents BaseConcepts
└
        . . .
|
  CommunicationChannel : CLASS
  COMMUNICATION_CHANNEL : ℙ INSTANCE
|
  isA(CommunicationChannel) = Class
  extension(⦇{CommunicationChannel}⦈) = COMMUNICATION_CHANNEL
  ¬ abstract CommunicationChannel
└
```

```
— section CommunicationChannelIntensional parents ClassIntensional
└
```

Nello schema di stato si asserisce che un canale di comunicazione mantiene due funzioni iniettive. Una dai naturali non nulli a delle Socket, l'altra tra indirizzi e Socket. Le funzioni sono necessariamente delle iniezioni per quanto detto prima, ovvero che su un dato end point può essere in ascolto una entità alla volta, mentre tutte possono ascoltare l'end point di broadcast (e dunque le loro Socket vengono distinte in base all'indirizzo dell'entità).
Si asserisce inoltre che una Socket può essere impiegata in una sola delle modalità bradcast o uno a uno (*unicast*).

```
┌─ ł CommunicationChannel
```

```
 łClass
 socket : ℕ↘1↖ ⤖ SOCKET
 broadcast : ENTITYID ⤖ SOCKET
|
 this ∈ COMMUNICATION_CHANNEL
 ran socket ∩ ran broadcast = ∅
└_

┌─ łCommunicationChannelInit
 łClassInit
 łCommunicationChannel ’
└_

┌─ łCommunicationChannelFin
 łClassFin
 łCommunicationChannel
└_

┌─ PrełCommunicationChannelBind
 łCommunicationChannel
 s? : SOCKET
└_
┌─ PostłCommunicationChannelBind
 ΔłCommunicationChannel
 PrełCommunicationChannelBind
└_

┌─ PrełCommunicationChannelGetBroadcastSocket
```

łCommunicationChannel

e? : ENTITYID

└

┌─ PostłCommunicationChannelGetBroadcastSocket

Ξ łCommunicationChannel

PrełCommunicationChannelGetBroadcastSocket

s! : □□SOCKET

└

┌─ PrełCommunicationChannelGetSocket

łCommunicationChannel

p? : ℕ

└

┌─ PostłCommunicationChannelGetSocket

Ξ łCommunicationChannel

PrełCommunicationChannelGetSocket

s! : □□SOCKET

└

┌─ PrełCommunicationChannelSelectSockets

łCommunicationChannel

└

┌─ PostłCommunicationChannelSelectSockets

Ξ łCommunicationChannel

PrełCommunicationChannelSelectSockets

s! : □□SOCKET

└

┌─ PrełCommunicationChannelTransmitOneToOne
 łCommunicationChannel
 s? : SOCKET
└

┌─ PostłCommunicationChannelTransmitOneToOne
 ΔłCommunicationChannel
 PrełCommunicationChannelTransmitOneToOne
└

┌─ PrełCommunicationChannelTransmitBroadcast
 łCommunicationChannel
 s? : SOCKET
└

┌─ PostłCommunicationChannelTransmitBroadcast
 ΔłCommunicationChannel
 PrełCommunicationChannelTransmitBroadcast
└

┌─ PrełCommunicationChannelTransmit
 łCommunicationChannel
└

┌─ PostłCommunicationChannelTransmit
 ΔłCommunicationChannel
 PrełCommunicationChannelTransmit
└

— section CommunicationChannelRelational parents
CommunicationChannelIntensional , SocketRelational

└

. . .

Qui si asserisce che le Socket mantenute da un CommunicationChannel non sono mai nulle, che sono associate in modo consistente con i valori presenti negli attributi di ciascuna e che un messaggio non può essere contemporaneamente presente in una coda di uscita e di ingresso.

┌─ ℝCommunicationChannel

ℝClass

łCommunicationChannel

ΩBuffer[MESSAGE]

ΩMessage

ΩPipe

ΩSocket

ΩCommunicationChannel

|

ran socket ⊆ dom łSocket

ran broadcast ⊆ dom łSocket

∀ n : dom socket • (łSocket (socket n)).endNumber = n

∀ e : dom broadcast • (łSocket (broadcast e)).owner = e ∧ (łSocket (broadcast e)).endNumber = 0

⋃{s : ran socket ∪ ran broadcast • ran (łBuffer (łPipe (łSocket s).in).buffer).dataQueue } ∩

⋃{s : ran socket ∪ ran broadcast • ran (łBuffer (łPipe (łSocket s).out).buffer).dataQueue } = ∅

└

— section CommunicationChannelExtensional parents CommunicationChannelRelational , SocketExtensional

└

. . .

┌─ ÐCommunicationChannel
ÐSocket
ÆCommunicationChannel
|
CommunicationChannelIsClass
└

. . .

┌─ CommunicationChannelInit
łCommunicationChannelInit
ΔÐCommunicationChannel
|
ℝCommunicationChannel ′
ClassInit
this? ∈ COMMUNICATION_CHANNEL
socket′ = ∅
broadcast′ = ∅
└
—
CommunicationChannelNew == ∃ łCommunicationChannel ′ •
ƒCommunicationChannelN ∧ CommunicationChannelInit[oc!/this?]
└

┌─ CommunicationChannelFin
łCommunicationChannelFin

ΔĐCommunicationChannel

|

ℝCommunicationChannel

ClassFin

socket = Ø

broadcast = Ø

└

—

CommunicationChannelDelete == ∃ łCommunicationChannel •

ƒCommunicationChannelD ∧ CommunicationChannelFin

└

Bind consente ad una entità di registrare una Socket.

┌ PreCommunicationChannelBind

PrełCommunicationChannelBind

ĐCommunicationChannel

|

ℝCommunicationChannel

s? ∉ ran socket ∪ ran broadcast

s? ∈ dom ŁSocket

(ŁSocket s?).endNumber = 0 ⇒ (ŁSocket s?).owner ∉ dom broadcast

(ŁSocket s?).endNumber > 0 ⇒ (ŁSocket s?).endNumber ∉ dom socket

(ŁBuffer (ŁPipe (ŁSocket s?).in).buffer).dataQueue = ⟨ ⟩

(ŁBuffer (ŁPipe (ŁSocket s?).out).buffer).dataQueue = ⟨ ⟩

└

┌ PostCommunicationChannelBind

PostłCommunicationChannelBind

ΔĐCommunicationChannel

|

ΔℝCommunicationChannel

this′ = this

∃ owner! : ENTITYID ; endNumber! : ℕ

• SocketGetOwner[s?/os?] ⨟

SocketGetEndNumber[s?/os?] ∧

[| endNumber! > 0 ⇒ socket′ = socket ∪ {endNumber! ↦ s?} ∧

broadcast′ = broadcast

endNumber! = 0 ⇒ socket′ = socket ∧

broadcast′ = broadcast ∪ {owner! ↦ s?}]

└

—

CommunicationChannelBind == ∃ ΔłCommunicationChannel •

ƒCommunicationChannel ∧ PreCommunicationChannelBind ∧

PostCommunicationChannelBind

└

GetBroadcastSocket consente di recuperare l'eventuale Socket di broadcast di una certa entità.

┌ PreCommunicationChannelGetBroadcastSocket

PrełCommunicationChannelGetBroadcastSocket

ĐCommunicationChannel

|

ℝCommunicationChannel

└

┌ PostCommunicationChannelGetBroadcastSocket

PostłCommunicationChannelGetBroadcastSocket

≡ĐCommunicationChannel

|

ΔℝCommunicationChannel

e? ∈ dom broadcast ⇒ s! = {broadcast(e?)}

e? ∉ dom broadcast ⇒ s! = ∅

└

—

CommunicationChannelGetBroadcastSocket == ∃ Ξ łCommunicationChannel • ƒCommunicationChannel ∧ PreCommunicationChannelGetBroadcastSocket ∧ PostCommunicationChannelGetBroadcastSocket

└

GetSocket consente di recuperare l'eventuale Socket unicast associata ad un certo end number.

┌ PreCommunicationChannelGetSocket

PrełCommunicationChannelGetSocket

ĐCommunicationChannel

|

ℝCommunicationChannel

└

┌ PostCommunicationChannelGetSocket

PostłCommunicationChannelGetSocket

Ξ ĐCommunicationChannel

|

ΔℝCommunicationChannel

p? ∈ dom socket ⇒ s! = {socket(p?)}

p? ∉ dom socket ⇒ s! = ∅

└

—

CommunicationChannelGetSocket == ∃ Ξ łCommunicationChannel

• ƒCommunicationChannel ∧ PreCommunicationChannelGetSocket ∧ PostCommunicationChannelGetSocket
└

SelectSockets restituisce un insieme contenente le Socket registrate al canale che hanno messaggi da inviare.

┌ PreCommunicationChannelSelectSockets
 PrełCommunicationChannelSelectSockets
 ĐCommunicationChannel
|
 ℝCommunicationChannel
└

┌ PostCommunicationChannelSelectSockets
 PostłCommunicationChannelSelectSockets
 ∃ĐCommunicationChannel
|
 ΔℝCommunicationChannel
 s! ⊆ ran socket ∪ ran broadcast
 s! = { s : ran socket ∪ ran broadcast ; outQueue! : iseq MESSAGE | SocketGetOutQueue[s/os?] ∧ outQueue! ≠ ⟨ ⟩ • s }
└

—
 CommunicationChannelSelectSockets == ∃ ∃łCommunicationChannel • ƒCommunicationChannel ∧ PreCommunicationChannelSelectSockets ∧ PostCommunicationChannelSelectSockets
└

TransmitOneToOne trasmette il messaggio in cima alla coda di uscita di una Socket unicast con semantica best-effort.

```
┌─ PreCommunicationChannelTransmitOneToOneSuccess
  PreŁCommunicationChannelTransmitOneToOne
  ĐCommunicationChannel
|
  ℝCommunicationChannel
  s? ∈ ran socket
  (ŁBuffer (ŁPipe (ŁSocket s?).out).buffer).dataQueue ≠ ⟨ ⟩
  let msg == head (ŁBuffer (ŁPipe (ŁSocket s?).out).buffer).dataQueue
  • [| (ŁMessage msg).receiverEnd ∈ dom socket
       (ŁSocket (socket (ŁMessage msg).receiverEnd)).owner = (ŁMessage
msg).receiver
       (ŁPipe (ŁSocket (socket (ŁMessage msg).receiverEnd)).in).isClosed = False
]
└─
┌─ PostCommunicationChannelTransmitOneToOneSuccess
  PostŁCommunicationChannelTransmitOneToOne
  ΔĐCommunicationChannel
|
  ΔℝCommunicationChannel
  this′ = this
  socket′ = socket
  broadcast′ = broadcast
  ∃ m! : □□MESSAGE ; m : MESSAGE ; receiver! , owner! : ENTITYID ;
receiverEnd! : ℕ ; s! : □□SOCKET ; dest : SOCKET ; isInClosed! : □ □
  | m! = {m} ∧ s! = {dest} ∧ owner! = receiver! ∧ isInClosed! = False
  • SocketGet[s?/os?] ⨾
    MessageGetReceiver[m/om?] ⨾
    MessageGetReceiverEnd[m/om?] ⨾
```

CommunicationChannelGetSocket[this/oc?,receiverEnd!/p?] ⨟
SocketGetOwner[dest/os?] ⨟
SocketIsInClosed[dest/os?] ⨟
SocketPut[dest/os?,m/m?]

└

┌─ PreCommunicationChannelTransmitOneToOneNoDelivery
PrełCommunicationChannelTransmitOneToOne
ÐCommunicationChannel
|
ℝCommunicationChannel
s? ∈ ran socket
(ŁBuffer (ŁPipe (ŁSocket s?).out).buffer).dataQueue ≠ ⟨ ⟩
let msg == head (ŁBuffer (ŁPipe (ŁSocket s?).out).buffer).dataQueue
• ¬ [| (ŁMessage msg).receiverEnd ∈ dom socket
(ŁSocket (socket (ŁMessage msg).receiverEnd)).owner = (ŁMessage msg).receiver
(ŁPipe (ŁSocket (socket (ŁMessage msg).receiverEnd)).in).isClosed = False]

└

┌─ PostCommunicationChannelTransmitOneToOneNoDelivery
PostłCommunicationChannelTransmitOneToOne
ΔÐCommunicationChannel
|
ΔℝCommunicationChannel
this′ = this
socket′ = socket
broadcast′ = broadcast
∃ m! : □□MESSAGE ; m : MESSAGE

| m! = {m}
• SocketGet[s?/os?] ⨟
MessageDelete[m/om?]
└

—

CommunicationChannelTransmitOneToOne ==
∃ ΔłCommunicationChannel
• ƒCommunicationChannel ∧
((PreCommunicationChannelTransmitOneToOneSuccess ∧
PostCommunicationChannelTransmitOneToOneSuccess) ∨
(PreCommunicationChannelTransmitOneToOneNoDelivery ∧
PostCommunicationChannelTransmitOneToOneNoDelivery))
└

TransmitBroadcast trasmette il messaggio in cima alla coda di uscita di una Socket di broadcast con semantica best effort.

┌─ PreCommunicationChannelTransmitBroadcast
PrełCommunicationChannelTransmitBroadcast
ĐCommunicationChannel
|
ℝCommunicationChannel
s? ∈ ran broadcast
(ŁBuffer (ŁPipe (ŁSocket s?).out).buffer).dataQueue ≠ ⟨ ⟩
└
┌─ DeliverBroadcast
ΔĐSocket
sSet? : □□SOCKET
m? : MESSAGE
|

m? ∈ dom ŁMessage

m? ∉ ⋃{ s : sSet? • ran (ŁBuffer (ŁPipe (ŁSocket s).in).buffer).dataQueue ∪ ran (ŁBuffer (ŁPipe (ŁSocket s).out).buffer).dataQueue }

∀ s : sSet? • (ŁSocket s).endNumber = 0

(ŁMessage m?).senderEnd = 0

sSet? = ∅ ⇒ ΞĐSocket

sSet? ≠ ∅ ⇒ ∃ s : sSet? ; newSSet : ℙ SOCKET ; isInClosed! : 𝔹

| newSSet = sSet? ∖ {s}

• SocketIsInClosed[s/os?] ⨾

(([| isInClosed! = False] ⇒ SocketPut[s/os?]) ∧

([| isInClosed! = True] ⇒ ΞĐSocket)) ⨾

DeliverBroadcast[newSSet/sSet?]

└

┌ PostCommunicationChannelTransmitBroadcast

PostŁCommunicationChannelTransmitBroadcast

ΔĐCommunicationChannel

|

ΔℝCommunicationChannel

this′ = this

socket′ = socket

broadcast′ = broadcast

∃ m! : ℙ MESSAGE ; m : MESSAGE ; sSet? : ℙ SOCKET

| m! = {m} ∧ sSet? = ran broadcast

• SocketGet[s?/os?] ⨾

DeliverBroadcast[m/m?]

└

—

CommunicationChannelTransmitBroadcast ==

∃ ΔłCommunicationChannel

• ƒCommunicationChannel ∧ PreCommunicationChannelTransmitBroadcast ∧ PostCommunicationChannelTransmitBroadcast

└

Quando l'evento Transmit avviene al canale alcuni messaggi vengono trasmessi.

┌─ PreCommunicationChannelTransmit

PrełCommunicationChannelTransmit

ĐCommunicationChannel

|

ℝCommunicationChannel

∃ s : ran socket ∪ ran broadcast • (ŁBuffer (ŁPipe (ŁSocket s).out).buffer).dataQueue ≠ 〈 〉

└

┌─ Transmit

ΔĐCommunicationChannel

sSet? : □□SOCKET

c? : COMMUNICATION_CHANNEL

|

c? ∈ dom ŁCommunicationChannel

sSet? ⊆ ran (ŁCommunicationChannel c?).socket ∪ ran (ŁCommunicationChannel c?).broadcast

∀ s : sSet? • (ŁBuffer (ŁPipe (ŁSocket s).out).buffer).dataQueue ≠ 〈 〉

sSet? = ∅ ⇒ ΞĐCommunicationChannel

sSet? ≠ ∅ ⇒ ∃ s : sSet? ; newSSet : □□SOCKET ; endNumber! : ℕ

| newSSet = sSet? ∖ {s}

• SocketGetEndNumber[s/os?] ⨾

(([| endNumber! = 0] ⇒

CommunicationChannelTransmitBroadcast[c?/oc?,s/s?]) ∧

([| endNumber! > 0] ⇒

CommunicationChannelTransmitOneToOne[c?/oc?,s/s?])) ⨟

Transmit[newSSet/sSet?]

└

┌─ PostCommunicationChannelTransmit

PostłCommunicationChannelTransmit

ΔÐCommunicationChannel

|

ΔℝCommunicationChannel

this′ = this

socket′ = socket

broadcast′ = broadcast

∃ s! : □□SOCKET

| s! ≠ Ø

• CommunicationChannelSelectSockets[this/oc?] ⨟

Transmit[this/c?,s!/sSet?]

└

—

CommunicationChannelTransmit ==

∃ ΔłCommunicationChannel • ƒCommunicationChannel ∧

PreCommunicationChannelTransmit ∧ PostCommunicationChannelTransmit

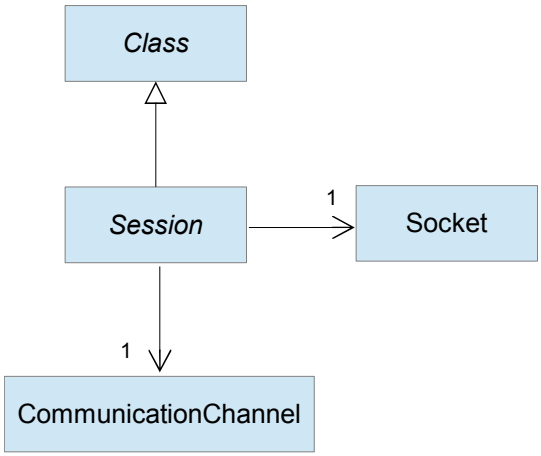
└

4.8.2.6 La nozione di Session

I concetti sin'ora definiti sono tutti assimilabili a meccanismi. Tramite essi si vogliono ora specificare le metafore desiderate per l'interazione.

Una sessione di comunicazione viene aperta da una entità per effettuare azioni di

comunicazione di un certo tipo. Una volta aperta una sessione, è possibile effettuare azioni di quella tipologia con diverse entità comunicanti fino alla sua chiusura.




```
— section AnalysisConcepts parents BaseConcepts
└
        . . .
|
  Session : CLASS
  SESSION : ℙ INSTANCE
|
  isA(Session) = Class
  extension(⦇{Session}⦈) = SESSION
  abstract Session
└
```

```
— section SessionIntensional parents ClassIntensional
└
```

Una Sessione si avvale di una Socket e di un canale di comunicazione dove dovrà registrarla. Inoltre mantiene una funzione che associa ad ogni destinazione il successivo numero di sequenza da utilizzare per il messaggio.

┌─ łSession
 łClass
 endPoint : SOCKET
 channel : COMMUNICATION_CHANNEL
 sequenceNumbers : ENTITYID × ℕ ⇸ ℕ
|
 this ∈ SESSION
└─

┌─ łSessionInit
 łClassInit
 łSession ’
 owner? : ENTITYID
 endNumber? : ℕ
 channel? : COMMUNICATION_CHANNEL
 sequenceNumbers? : ENTITYID × ℕ ⇸ ℕ
└─

┌─ łSessionFin
 łClassFin
 łSession
└─

┌─ PrełSessionGetOwner
 łSession
└─
┌─ PostłSessionGetOwner
 ≡łSession

PrełSessionGetOwner

owner! : ENTITYID

└

┌ PrełSessionGetEndNumber

łSession

└

┌ PostłSessionGetEndNumber

Ξ łSession

PrełSessionGetEndNumber

endNumber! : ℕ

└

— section SessionRelational parents SessionIntensional , CommunicationChannelRelational

└

. . .

Qui si asserisce che la Socket di una sessione è sempre registrata in modo consistente col canale di comunicazione riferito. Inoltre viene affermato che due messaggi in uscita successivi, verso la stessa destinazione, sono sempre contraddistinti da numeri di sequenza crescenti. La funzione sequence number tiene traccia del prossimo numero da utilizzare per ogni destinazione a cui si sono inviati messaggi. Si noti che in tal modo è possibile, tramite la tripla (receiver,receiverEnd,sequenceNumber), identificare univocamente un messaggio inviato all'interno di una sessione.

┌ ℝSession

ℝClass

łSession

ΩBuffer[MESSAGE]

```
  ΩMessage
  ΩPipe
  ΩSocket
  ΩCommunicationChannel
  ΩSession
|
  endPoint ∈ dom ŁSocket
  channel ∈ dom ŁCommunicationChannel
  (ŁSocket endPoint).owner ≠ eAll
  (ŁSocket endPoint).endNumber = 0 ⇒ (ŁSocket endPoint).owner ∈ dom
(ŁCommunicationChannel channel).broadcast ∧
                                                (ŁCommunicationChannel
channel).broadcast((ŁSocket endPoint).owner) = endPoint
  (ŁSocket endPoint).endNumber > 0 ⇒ (ŁSocket endPoint).endNumber ∈ dom
(ŁCommunicationChannel channel).socket ∧
                                                (ŁCommunicationChannel
channel).socket((ŁSocket endPoint).endNumber) = endPoint
  let outgoing == (ŁBuffer (ŁPipe (ŁSocket endPoint).out).buffer).dataQueue
  • [| ∀ i , j : 1 .. #outgoing
        | i < j ∧
          (ŁMessage (outgoing i)).receiver = (ŁMessage (outgoing j)).receiver ∧
          (ŁMessage (outgoing i)).receiverEnd = (ŁMessage (outgoing
j)).receiverEnd
        • (ŁMessage (outgoing i)).sequenceNumber < (ŁMessage (outgoing
j)).sequenceNumber
        ∀ m : ran outgoing • ((ŁMessage m).receiver,(ŁMessage m).receiverEnd) ∈
dom sequenceNumbers
        ∀ d : {m : ran outgoing • ((ŁMessage m).receiver,(ŁMessage
```

m).receiverEnd)}

- (ŁMessage (outgoing (max {i : dom outgoing
| (ŁMessage (outgoing i)).receiver = d.1 ∧
(ŁMessage (outgoing i)).receiverEnd =
d.2}))).sequenceNumber = sequenceNumbers(d.1,d.2) − 1]

⌙

— section SessionExtensional parents SessionRelational ,
CommunicationChannelExtensional

⌙

. . .

┌─ ĐSession
ĐCommunicationChannel
ÆSession
|
SessionIsClass
⌙

. . .

Il costruttore SessionInit con i dati forniti, crea una Socket e la registra al canale indicato.

┌─ SessionInit
łSessionInit
ΔĐSession
|
ℝSession ’
ClassInit

this? ∈ SESSION

owner? ≠ eAll

channel? ∈ dom ŁCommunicationChannel

endNumber? = 0 ⇒ owner? ∉ dom (ŁCommunicationChannel channel?).broadcast

endNumber? > 0 ⇒ endNumber? ∉ dom (ŁCommunicationChannel channel?).socket

channel' = channel?

sequenceNumbers' = sequenceNumbers?

SocketNew[endPoint'/os!] ⨟

CommunicationChannelBind[channel?/oc?,endPoint'/s?]

└

Una sessione può essere finalizzata solo dopo che la sua Socket è stata chiusa.

┌ SessionFin

łSessionFin

ΔÐSession

|

ℝSession

ClassFin

(ŁPipe (ŁSocket endPoint).in).isClosed = True

(ŁPipe (ŁSocket endPoint).out).isClosed = True

└

┌ PreSessionGetOwner

PrełSessionGetOwner

ÐSession

|

ℝSession

└

┌─ PostSessionGetOwner

PostłSessionGetOwner

ΞĐSession

|

ΔℝSession

SocketGetOwner[endPoint/os?]

└

—

SessionGetOwner == ∃ ΞłSession • ƒSession ∧ PreSessionGetOwner ∧ PostSessionGetOwner

└

┌─ PreSessionGetEndNumber

PrełSessionGetEndNumber

ĐSession

|

ℝSession

└

┌─ PostSessionGetEndNumber

PostłSessionGetEndNumber

ΞĐSession

|

ΔℝSession

SocketGetEndNumber[endPoint/os?]

└

—

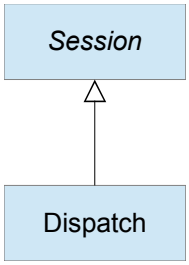
SessionGetEndNumber == ∃ ⨋ł Session • ƒSession ∧

PreSessionGetEndNumber ∧ PostSessionGetEndNumber

└

4.8.2.7 La nozione di Dispatch

E' ora possibile definire la prima delle tipologie desiderate di interazione. Come si è detto per dispaccio si intende l'invio asincrono di un messaggio, senza alcuna attesa successiva su di esso (fire and forget). Per farlo occorre aprire una sessione di questa categoria. Successivamente è possibile inviare dispacci (evento Forward) o riceverli (evento Serve). La ricezione è possibile sia in modalità non bloccante che bloccante. L'invio di un messaggio è invece per sua natura un'operazione non bloccante.



— section AnalysisConcepts parents BaseConcepts

└

. . .

|

Dispatch : CLASS

DISPATCH : ℙ SESSION

|

isA(Dispatch) = Session

extension(⦇{Dispatch}⦈) = DISPATCH

¬ abstract Dispatch

└

— section DispatchIntensional parents SessionIntensional

└

Nell'intensione di stato viene aggiunto un invariante, il quale afferma che l'end point di destinazione di un dispaccio non può essere quello di broadcast. Come detto infatti i dispacci si effettuano verso entità specifiche.

┌ ł Dispatch

ł Session

|

this ∈ DISPATCH

∀ d : dom sequenceNumbers • d.2 > 0

└

┌ ł DispatchInit

ł SessionInit

ł Dispatch ’

└

┌ ł DispatchFin

ł SessionFin

ł Dispatch

└

┌ Preł DispatchGetOwner

Preł SessionGetOwner

ł Dispatch

└

┌─ PostłDispatchGetOwner
PostłSessionGetOwner
ΞłDispatch
PrełDispatchGetOwner
└

┌─ PrełDispatchGetEndNumber
PrełSessionGetEndNumber
łDispatch
└
┌─ PostłDispatchGetEndNumber
PostłSessionGetEndNumber
ΞłDispatch
PrełDispatchGetEndNumber
└

┌─ PrełDispatchForward
łDispatch
receiver? : ENTITYID
receiverEnd? : ℕ↘1↖
data? : seq SYMBOL
└
┌─ PostłDispatchForward
ΔłDispatch
PrełDispatchForward
└

┌─ PrełDispatchServe

łDispatch

└

┌─ PostłDispatchServe

ΔłDispatch

PrełDispatchServe

m! : □□MESSAGE

└

┌─ PrełDispatchGetServeWait

łDispatch

└

┌─ PostłDispatchGetServeWait

≡łDispatch

PrełDispatchGetServeWait

serveWait! : CONDITION

└

┌─ PrełDispatchCloseIn

łDispatch

└

┌─ PostłDispatchCloseIn

ΔłDispatch

PrełDispatchCloseIn

└

┌─ PrełDispatchCloseOut

łDispatch

└

┌─ PostłDispatchCloseOut
ΔłDispatch
PrełDispatchCloseOut
└─

┌─ PrełDispatchIsInClosed
łDispatch
└─

┌─ PostłDispatchIsInClosed
ΞłDispatch
PrełDispatchIsInClosed
isInClosed! : □□
└─

┌─ PrełDispatchIsOutClosed
łDispatch
└─

┌─ PostłDispatchIsOutClosed
ΞłDispatch
PrełDispatchIsOutClosed
isOutClosed! : □□
└─

— section DispatchRelational parents DispatchIntensional , SessionRelational
└─

. . .

Gli invarianti relazionali affermano che un dispaccio non può essere inviato dall'end point di broadcast e che poiché non si attende alcuna risposta, i numeri di sequenza e gli identificatori di richiesta dei messaggi in uscita coincidono.

```
┌─ ℝDispatch
  ℝSession
  łDispatch
  ΩDispatch
|
  (ŁSocket endPoint).endNumber > 0
  let outgoing == (ŁBuffer (ŁPipe (ŁSocket endPoint).out).buffer).dataQueue
  • [| ∀ m : ran outgoing • (ŁMessage m).sequenceNumber = (ŁMessage
m).reqId ]
└
```

```
— section DispatchExtensional parents DispatchRelational , SessionExtensional
└
        . . .

┌─ ÐDispatch
  ÐSession
  ÆDispatch
|
  DispatchIsSession
└
        . . .
```

Il costruttore DispatchNew consente di specificare, se desiderato, oltre alle informazioni riguardanti il mittente dei messaggi, i numeri di sequenza iniziali di alcune destinazioni.

```
┌─ DispatchInit
```

łDispatchInit

ΔĐDispatch

|

ℝDispatch ’

SessionInit

this? ∈ DISPATCH

endNumber? > 0

∀ d : dom sequenceNumbers? • d.2 > 0

└

—

DispatchNew == ∃ łDispatch ’ • ƒDispatchN ∧ DispatchInit[od!/this?]

└

┌ DispatchFin

łDispatchFin

ΔĐDispatch

|

ℝDispatch

SessionFin

#(ŁCondition (ŁBuffer (ŁPipe (ŁSocket endPoint).in).buffer).dataWait).activityQueue = 0

└

—

DispatchDelete == ∃ łDispatch • ƒDispatchD ∧ DispatchFin

└

┌ PreDispatchGetOwner

PrełDispatchGetOwner

ÐDispatch

|

ℝDispatch

PreSessionGetOwner

└

┌ PostDispatchGetOwner

PostłDispatchGetOwner

ΞłÐDispatch

|

ΔℝDispatch

PostSessionGetOwner

└

—

DispatchGetOwner == ∃ ΞłDispatch • ƒDispatch ∧ PreDispatchGetOwner ∧ PostDispatchGetOwner

└

┌ PreDispatchGetEndNumber

PrełDispatchGetEndNumber

ÐDispatch

|

ℝDispatch

PreSessionGetEndNumber

└

┌ PostDispatchGetEndNumber

PostłDispatchGetEndNumber

ΞłÐDispatch

|

ΔℝDispatch

PostSessionGetEndNumber

└

—

DispatchGetEndNumber == ∃ ΞłDispatch • ƒDispatch ∧ PreDispatchGetEndNumber ∧ PostDispatchGetEndNumber

└

┌ PreDispatchForward

PrełDispatchForward

ĐDispatch

|

ℝDispatch

receiver? ≠ eAll

(ŁPipe (ŁSocket endPoint).out).isClosed = False

└

┌ PostDispatchForward

PostłDispatchForward

ΔĐDispatch

|

ΔℝDispatch

this′ = this

endPoint′ = endPoint

channel′ = channel

(receiver?,receiverEnd?) ∈ dom sequenceNumbers

⇒ sequenceNumbers′ = sequenceNumbers ⊕ {(receiver?,receiverEnd?) ↦ sequenceNumbers(receiver?,receiverEnd?) + 1}

(receiver?,receiverEnd?) ∉ dom sequenceNumbers

⇒ sequenceNumbers′ = sequenceNumbers ∪ {(receiver?,receiverEnd?) ↦ 1}

∃ owner! : ENTITYID ; endNumber! : ℕ↘1↖ ; sequenceNumber? : ℕ ; om! : MESSAGE

| ((receiver?,receiverEnd?) ∈ dom sequenceNumbers ⇒ sequenceNumber? = sequenceNumbers(receiver?,receiverEnd?)) ∧

((receiver?,receiverEnd?) ∉ dom sequenceNumbers ⇒ sequenceNumber? = 0)

• SocketGetOwner[endPoint/os?] ⨟

SocketGetEndNumber[endPoint/os?] ⨟ MessageNew[owner!/sender?,endNumber!/senderEnd?,sequenceNumber?/reqId?] ⨟

SocketSend[endPoint/os?,om!/m?]

└

—

DispatchForward == ∃ Δ‡Dispatch • ƒDispatch ∧ PreDispatchForward ∧ PostDispatchForward

└

┌ PreDispatchServe

Pre‡DispatchServe

Đ Dispatch

|

ℝDispatch

└

┌ PostDispatchServe

Post‡DispatchServe

ΔĐDispatch

|

```
  ΔℝDispatch
  this′ = this
  endPoint′ = endPoint
  channel′ = channel
  sequenceNumbers′ = sequenceNumbers
  #m! ∈ 0 .. 1
  SocketReceive[endPoint/os?]
└
─
  DispatchServe == ∃ ΔłDispatch • ƒDispatch ∧ PreDispatchServe ∧
PostDispatchServe
└


┌─ PreDispatchGetServeWait
  PrełDispatchGetServeWait
  ĐDispatch
|
  ℝDispatch
└
┌─ PostDispatchGetServeWait
  PostłDispatchGetServeWait
  ΞŧĐDispatch
|
  ΔℝDispatch
  SocketGetInWait[endPoint/os?,serveWait!/inWait!]
└
─
  DispatchGetServeWait == ∃ ΞłDispatch • ƒDispatch ∧
```

PreDispatchGetServeWait ∧ PostDispatchGetServeWait

└

Come è possibile vedere, l'operazione di ricezione Serve ha sempre successo e nel caso non vi sia informazione da ritirare, restituisce un insieme vuoto.

Qualora una attività decidesse di bloccarsi nell'attesa di un messaggio, può utilizzare l'evento GetServeWait per recuperare la variabile condizione adatta e in seguito invocare l'evento Await su di essa.

Alla ricezione di un messaggio sull'end point associato a questa sessione, l'attività verrà posta nuovamente in stato Runnable e potrà ritentare la ricezione.

Come spiegato nel paragrafo 4.6.4, questa modalità di utilizzo delle variabili condizione è orientata ad esplicitare *cosa* fa una attività, consentendo di analizzare il suo effetto sul sistema da un unico punto (il suo evento Run).

In *fase di progetto*, una scelta appropriata per trattare efficacemente la realizzazione dell'evento di ricezione, potrebbe essere quella di non fornire l'evento GetServeWait nell'interfaccia concreta del blocco, ma dare la possibilità all'attività invocante l'evento Serve, di fornire un parametro (ad esempio booleano) in ingresso, che dichiari la volontà dell'attività di bloccarsi o meno.

In tal modo essa, una volta risvegliata da una ricezione bloccante, constatando di aver ricevuto un insieme vuoto dall'evento, potrà concludere con sicurezza che la sessione è stata chiusa.

Ovviamente per seguire questa strada sarebbe appropriato proteggere l'invocazione di Serve con meccanismi di mutua esclusione fino alla sua sospensione o conclusione.

┌ PreDispatchCloseIn

PreDispatchCloseIn

ĐDispatch

|

ℝDispatch

└

┌ PostDispatchCloseIn

PostDispatchCloseIn

ΔĐDispatch

|

ΔℝDispatch

this′ = this

endPoint′ = endPoint

channel′ = channel

sequenceNumbers′ = sequenceNumbers

SocketCloseIn[endPoint/os?]

└

—

DispatchCloseIn == ∃ ΔłDispatch • ƒDispatch ∧ PreDispatchCloseIn ∧ PostDispatchCloseIn

└

Si ricorda che la chiusura di uno dei canali sveglia tutte le attività in attesa sul canale.

┌ PreDispatchCloseOut

PrełDispatchCloseOut

ĐDispatch

|

ℝDispatch

└

┌ PostDispatchCloseOut

PostłDispatchCloseOut

ΔĐDispatch

|

ΔℝDispatch

this′ = this

endPoint′ = endPoint

channel′ = channel

sequenceNumbers′ = sequenceNumbers

SocketCloseOut[endPoint/os?]

└

—

DispatchCloseOut == ∃ ΔłDispatch • ƒDispatch ∧ PreDispatchCloseOut ∧ PostDispatchCloseOut

└

┌─ PreDispatchIsInClosed

PrełDispatchIsInClosed

ĐDispatch

|

ℝDispatch

└

┌─ PostDispatchIsInClosed

PostłDispatchIsInClosed

ΞĐDispatch

|

ΔℝDispatch

SocketIsInClosed[endPoint/os?]

└

—

DispatchIsInClosed == ∃ ΞłDispatch • ƒDispatch ∧ PreDispatchIsInClosed ∧ PostDispatchIsInClosed

└

┌─ PreDispatchIsOutClosed

PrełDispatchIsOutClosed

ĐDispatch

|

ℝDispatch

└

┌─ PostDispatchIsOutClosed

PostłDispatchIsOutClosed

ΞĐDispatch

|

ΔℝDispatch

SocketIsOutClosed[endPoint/os?]

└

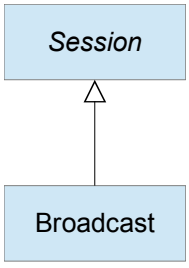
—

DispatchIsOutClosed == ∃ ΞłDispatch • ƒDispatch ∧ PreDispatchIsOutClosed ∧ PostDispatchIsOutClosed

└

4.8.2.8 La nozione di Broadcast

Come si è detto un broadcast consente di rendere noto un messaggio al maggior numero possibile di entità comunicanti. Dopo aver creato una sessione di questo tipo è possibile emettere broadcast (evento Emit) o porsi all'ascolto di essi in modalità bloccante o non bloccante (evento Sense), tramite l'end point associato alla sessione.



— section AnalysisConcepts parents BaseConcepts

└

. . .

|

Broadcast : CLASS

BROADCAST : ℙ SESSION

|

isA(Broadcast) = Session

extension(⦇{Broadcast}⦈) = BROADCAST

¬ abstract Broadcast

└

— section BroadcastIntensional parents SessionIntensional

└

L'invariante dell'intensione di stato afferma che l'unica destinazione possibile di un broadcast sono tutte le entità e l'end point di broadcast.

┌ ł Broadcast

ł Session

|

this ∈ BROADCAST

dom sequenceNumbers = {(eAll,0)}

└

┌ ł BroadcastInit

ł SessionInit

ł Broadcast ’

└

┌─ łBroadcastFin
 łSessionFin
 łBroadcast
└

┌─ PrełBroadcastGetOwner
 PrełSessionGetOwner
 łBroadcast
└
┌─ PostłBroadcastGetOwner
 PostłSessionGetOwner
 ≡łBroadcast
 PrełBroadcastGetOwner
└

┌─ PrełBroadcastGetEndNumber
 PrełSessionGetEndNumber
 łBroadcast
└
┌─ PostłBroadcastGetEndNumber
 PostłSessionGetEndNumber
 ≡łBroadcast
 PrełBroadcastGetEndNumber
└

┌─ PrełBroadcastEmit
 łBroadcast

data? : seq SYMBOL

└

┌─ PostłBroadcastEmit

ΔłBroadcast

PrełBroadcastEmit

└

┌─ PrełBroadcastSense

łBroadcast

└

┌─ PostłBroadcastSense

ΔłBroadcast

PrełBroadcastSense

m! : □□MESSAGE

└

┌─ PrełBroadcastGetSenseWait

łBroadcast

└

┌─ PostłBroadcastGetSenseWait

ΞłBroadcast

PrełBroadcastGetSenseWait

senseWait! : CONDITION

└

┌─ PrełBroadcastCloseIn

łBroadcast

└

┌─ PostłBroadcastCloseIn
  ΔłBroadcast
  PrełBroadcastCloseIn
└─

┌─ PrełBroadcastCloseOut
  łBroadcast
└─
┌─ PostłBroadcastCloseOut
  ΔłBroadcast
  PrełBroadcastCloseOut
└─

┌─ PrełBroadcastIsInClosed
  łBroadcast
└─
┌─ PostłBroadcastIsInClosed
  ΞłBroadcast
  PrełBroadcastIsInClosed
  isInClosed! : □□
└─

┌─ PrełBroadcastIsOutClosed
  łBroadcast
└─
┌─ PostłBroadcastIsOutClosed
  ΞłBroadcast
  PrełBroadcastIsOutClosed

```
  isOutClosed! : □□
└
```

```
— section BroadcastRelational parents BroadcastIntensional , SessionRelational
└
```

. . .

Qui si afferma che in un broadcast l'end number è necessariamente 0 (e dunque non occorre lo si fornisca alla creazione di una tale sessione) e che i numeri di sequenza e gli identificatori di richiesta coincidono.

```
┌─ ℝBroadcast
  ℝSession
  łBroadcast
  ΩBroadcast
|
  (ŁSocket endPoint).endNumber = 0
  let outgoing == (ŁBuffer (ŁPipe (ŁSocket endPoint).out).buffer).dataQueue
  • [| ∀ m : ran outgoing • (ŁMessage m).sequenceNumber = (ŁMessage
m).reqId ]
└
```

```
— section BroadcastExtensional parents BroadcastRelational ,
SessionExtensional
└
```

. . .

```
┌─ ÐBroadcast
  ÐSession
  ÆBroadcast
```

|

BroadcastIsSession

└

. . .

Si noti come il costruttore di un Broadcast (BroadcastNew) non richieda l'end number al contrario di quello previsto per una generica Session. Dunque c'è una variazione nella signature, che nel caso dei costruttori è permessa, poiché essi, essendo specifici per una certa classe, fanno eccezione.

┌ BroadcastInit

łBroadcastInit

ΔĐBroadcast

|

ℝBroadcast’

SessionInit

this? ∈ BROADCAST

endNumber? = 0

dom sequenceNumbers? = {(eAll,0)}

└

—

BroadcastNew == ∃ łBroadcast’ ; endNumber? : ℕ | endNumber? = 0 •

ƒBroadcastN ∧ BroadcastInit[ob!/this?]

└

┌ BroadcastFin

łBroadcastFin

ΔĐBroadcast

|

ℝBroadcast

SessionFin

#(ŁCondition (ŁBuffer (ŁPipe (ŁSocket endPoint).in).buffer).dataWait).activityQueue = 0

└

—

BroadcastDelete == ∃ łBroadcast • ƒBroadcastD ∧ BroadcastFin

└

┌─ PreBroadcastGetOwner

PrełBroadcastGetOwner

ĐBroadcast

|

ℝBroadcast

PreSessionGetOwner

└

┌─ PostBroadcastGetOwner

PostłBroadcastGetOwner

ΞĐBroadcast

|

ΔℝBroadcast

PostSessionGetOwner

└

—

BroadcastGetOwner == ∃ ΞłBroadcast • ƒBroadcast ∧ PreBroadcastGetOwner ∧ PostBroadcastGetOwner

└

┌─ PreBroadcastGetEndNumber

PrełBroadcastGetEndNumber

ĐBroadcast

|

ℝBroadcast

PreSessionGetEndNumber

└

┌ PostBroadcastGetEndNumber

PostłBroadcastGetEndNumber

ΞĐBroadcast

|

ΔℝBroadcast

PostSessionGetEndNumber

└

—

BroadcastGetEndNumber == ∃ ΞłBroadcast • ƒBroadcast ∧ PreBroadcastGetEndNumber ∧ PostBroadcastGetEndNumber

└

┌ PreBroadcastEmit

PrełBroadcastEmit

ĐBroadcast

|

ℝBroadcast

(ŁPipe (ŁSocket endPoint).out).isClosed = False

└

┌ PostBroadcastEmit

PostłBroadcastEmit

ΔĐBroadcast

|

ΔℝBroadcast

this′ = this

endPoint′ = endPoint

channel′ = channel

sequenceNumbers′ = sequenceNumbers ⊕ {(eAll,0) ↦ sequenceNumbers(eAll,0) + 1}

∃ owner! , receiver? : ENTITYID ; senderEnd? , sequenceNumber? : ℕ ; om! : MESSAGE

| {(receiver?,senderEnd?)} = dom sequenceNumbers ∧

sequenceNumber? = sequenceNumbers(receiver?,senderEnd?)

• SocketGetOwner[endPoint/os?] ⨟

MessageNew[owner!/sender?,sequenceNumber?/reqId?,senderEnd?/receiverEnd?] ⨟

SocketSend[endPoint/os?,om!/m?]

└

—

BroadcastEmit == ∃ ΔłBroadcast • ƒBroadcast ∧ PreBroadcastEmit ∧ PostBroadcastEmit

└

┌ PreBroadcastSense

PrełBroadcastSense

ĐBroadcast

|

ℝBroadcast

└

┌ PostBroadcastSense

PostłBroadcastSense

ΔÐBroadcast

|

ΔℝBroadcast

this′ = this

endPoint′ = endPoint

channel′ = channel

sequenceNumbers′ = sequenceNumbers

#m! ∈ 0 .. 1

SocketReceive[endPoint/os?]

└

—

BroadcastSense == ∃ ΔłBroadcast • ƒBroadcast ∧ PreBroadcastSense ∧ PostBroadcastSense

└

┌ PreBroadcastGetSenseWait

PrełBroadcastGetSenseWait

ÐBroadcast

|

ℝBroadcast

└

┌ PostBroadcastGetSenseWait

PostłBroadcastGetSenseWait

ΞÐBroadcast

|

ΔℝBroadcast

SocketGetInWait[endPoint/os?,senseWait!/inWait!]

└

—

BroadcastGetSenseWait == ∃ Ξ‡Broadcast • ƒBroadcast ∧ PreBroadcastGetSenseWait ∧ PostBroadcastGetSenseWait

└

┌─ PreBroadcastCloseIn

Pre‡BroadcastCloseIn

ÐBroadcast

|

ℝBroadcast

└

┌─ PostBroadcastCloseIn

Post‡BroadcastCloseIn

ΔÐBroadcast

|

ΔℝBroadcast

this′ = this

endPoint′ = endPoint

channel′ = channel

sequenceNumbers′ = sequenceNumbers

SocketCloseIn[endPoint/os?]

└

—

BroadcastCloseIn == ∃ Δ‡Broadcast • ƒBroadcast ∧ PreBroadcastCloseIn ∧ PostBroadcastCloseIn

└

```
┌─ PreBroadcastCloseOut
  PrełBroadcastCloseOut
  ÐBroadcast
|
  ℝBroadcast
└─
┌─ PostBroadcastCloseOut
  PostłBroadcastCloseOut
  ΔÐBroadcast
|
  ΔℝBroadcast
  this′ = this
  endPoint′ = endPoint
  channel′ = channel
  sequenceNumbers′ = sequenceNumbers
  SocketCloseOut[endPoint/os?]
└─
──
  BroadcastCloseOut == ∃ ΔłBroadcast • ƒBroadcast ∧ PreBroadcastCloseOut
∧ PostBroadcastCloseOut
└─


┌─ PreBroadcastIsInClosed
  PrełBroadcastIsInClosed
  ÐBroadcast
|
  ℝBroadcast
└─
```

┌─ PostBroadcastIsInClosed

PostłBroadcastIsInClosed

Ξ ÐBroadcast

|

ΔℝBroadcast

SocketIsInClosed[endPoint/os?]

└

—

BroadcastIsInClosed == ∃ ΞłBroadcast • ƒBroadcast ∧ PreBroadcastIsInClosed ∧ PostBroadcastIsInClosed

└

┌─ PreBroadcastIsOutClosed

PrełBroadcastIsOutClosed

ÐBroadcast

|

ℝBroadcast

└

┌─ PostBroadcastIsOutClosed

PostłBroadcastIsOutClosed

Ξ ÐBroadcast

|

ΔℝBroadcast

SocketIsOutClosed[endPoint/os?]

└

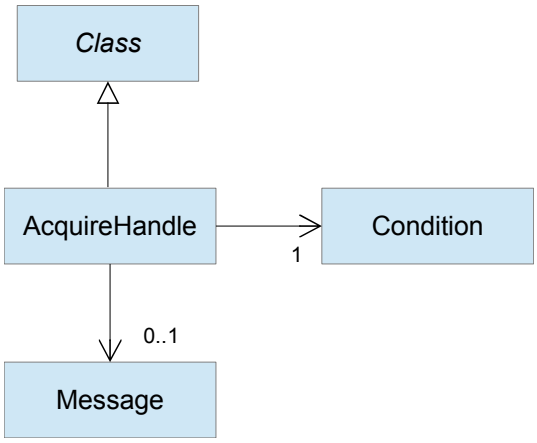
—

BroadcastIsOutClosed == ∃ ΞłBroadcast • ƒBroadcast ∧ PreBroadcastIsOutClosed ∧ PostBroadcastIsOutClosed

└

4.8.2.9 La nozione di AcquireHandle

Per poter precisare il concetto di request-reply, occorre introdurre alcuni meccanismi aggiuntivi. Poiché infatti dopo l'invio di una richiesta, si ha l'aspettativa di una risposta ed è in generale possibile inviare più richieste analoghe ad una stessa destinazione, occorre associare ogni risposta alla specifica richiesta che l'ha generata. A tal fine si definisce la classe AcquireHandle, la quale contiene i dati di una richiesta effettuata, la risposta se già pervenuta, e mette a disposizione la possibilità di sospendersi nell'attesa della risposta a quella specifica richiesta.



— section AnalysisConcepts parents BaseConcepts

└

. . .

|

AcquireHandle : CLASS

ACQUIRE_HANDLE : ℙ INSTANCE

|

isA(AcquireHandle) = Class

extension(⦇{AcquireHandle}⦈) = ACQUIRE_HANDLE

¬ abstract AcquireHandle

└

— section AcquireHandleIntensional parents ClassIntensional

└

┌─ łAcquireHandle

łClass

reqId : ℕ

receiver : ENTITYID

receiverEnd : ℕ↘1↖

replyWait : CONDITION

reply : □□MESSAGE

|

this ∈ ACQUIRE_HANDLE

#reply ∈ 0 .. 1

└

┌─ łAcquireHandleInit

łClassInit

łAcquireHandle ’

reqId? : ℕ

receiver? : ENTITYID

receiverEnd? : ℕ↘1↖

└

┌─ łAcquireHandleFin

łClassFin

łAcquireHandle

└

┌─ PrełAcquireHandleGetReply
łAcquireHandle
└

┌─ PostłAcquireHandleGetReply
ΞłAcquireHandle
PrełAcquireHandleGetReply
reply! : □□MESSAGE
└

┌─ PrełAcquireHandlePutReply
łAcquireHandle
r? : MESSAGE
└

┌─ PostłAcquireHandlePutReply
ΔłAcquireHandle
PrełAcquireHandlePutReply
└

┌─ PrełAcquireHandleGetReplyWait
łAcquireHandle
└

┌─ PostłAcquireHandleGetReplyWait
ΞłAcquireHandle
PrełAcquireHandleGetReplyWait
replyWait! : CONDITION
└

┌─ PrełAcquireHandleGetReqId
 łAcquireHandle
└─

┌─ PostłAcquireHandleGetReqId
 ΞłAcquireHandle
 PrełAcquireHandleGetReqId
 reqId! : ℕ
└─

┌─ PrełAcquireHandleGetReceiver
 łAcquireHandle
└─

┌─ PostłAcquireHandleGetReceiver
 ΞłAcquireHandle
 PrełAcquireHandleGetReceiver
 receiver! : ENTITYID
└─

┌─ PrełAcquireHandleGetReceiverEnd
 łAcquireHandle
└─

┌─ PostłAcquireHandleGetReceiverEnd
 ΞłAcquireHandle
 PrełAcquireHandleGetReceiverEnd
 receiverEnd! : ℕ↘1↖
└─

— section AcquireHandleRelational parents AcquireHandleIntensional , ConditionRelational , MessageRelational

└

. . .

Qui si asserisce che è possibile inserire nella handle solo una ben determinata risposta.

┌ ℝAcquireHandle

ℝClass

łAcquireHandle

ΩCondition

ΩMessage

ΩAcquireHandle

|

replyWait ∈ dom ŁCondition

reply ⊆ dom ŁMessage

∀ m : reply • (ŁMessage m).reqId = reqId ∧ (ŁMessage m).sender = receiver ∧ (ŁMessage m).senderEnd = receiverEnd

└

— section AcquireHandleExtensional parents AcquireHandleRelational , ConditionExtensional , MessageExtensional

└

. . .

┌ ĐAcquireHandle

ĐCondition

ĐMessage

ÆAcquireHandle

|

AcquireHandleIsClass

└

. . .

┌─ AcquireHandleInit

łAcquireHandleInit

ΔĐAcquireHandle

|

ℝAcquireHandle ’

ClassInit

this? ∈ ACQUIRE_HANDLE

reqId’ = reqId?

receiver’ = receiver?

receiverEnd’ = receiverEnd?

reply’ = ∅

ConditionNew[replyWait’/oc!]

└

—

AcquireHandleNew == ∃ łAcquireHandle ’ • ƒAcquireHandleN ∧ AcquireHandleInit[oa!/this?]

└

┌─ AcquireHandleFin

łAcquireHandleFin

ΔĐAcquireHandle

|

ℝAcquireHandle

ClassFin

```
  #(ŁCondition replyWait).activityQueue = 0
└
—
  AcquireHandleDelete == ∃ łAcquireHandle • ƒAcquireHandleD ∧
AcquireHandleFin
└

┌ PreAcquireHandleGetReply
  PrełAcquireHandleGetReply
  ĐAcquireHandle
|
  ℝAcquireHandle
└
┌ PostAcquireHandleGetReply
  PostłAcquireHandleGetReply
  ΞĐAcquireHandle
|
  ΔℝAcquireHandle
  reply! = reply
└
—
  AcquireHandleGetReply == ∃ ΞłAcquireHandle • ƒAcquireHandle ∧
PreAcquireHandleGetReply ∧ PostAcquireHandleGetReply
└
```

Come si nota nelle precondizioni di PutReply, è possibile deporre una sola risposta nella handle.

```
┌ PreAcquireHandlePutReply
  PrełAcquireHandlePutReply
```

ĐAcquireHandle

|

ℝAcquireHandle

reply = Ø

r? ∈ dom ŁMessage

(ŁMessage r?).sender = receiver

(ŁMessage r?).senderEnd = receiverEnd

(ŁMessage r?).reqId = reqId

└

┌─ PostAcquireHandlePutReply

PostłAcquireHandlePutReply

ΔĐAcquireHandle

|

ΔℝAcquireHandle

this′ = this

reqId′ = reqId

receiver′ = receiver

receiverEnd′ = receiverEnd

replyWait′ = replyWait

reply′ = {r?}

ConditionSignal[replyWait/oc?]

└

—

AcquireHandlePutReply == ∃ ΔłAcquireHandle • ƒAcquireHandle ∧ PreAcquireHandlePutReply ∧ PostAcquireHandlePutReply

└

┌─ PreAcquireHandleGetReplyWait

PrełAcquireHandleGetReplyWait

ĐAcquireHandle

|

ℝAcquireHandle

└

┌ PostAcquireHandleGetReplyWait

PostłAcquireHandleGetReplyWait

ΞĐAcquireHandle

|

ΔℝAcquireHandle

replyWait! = replyWait

└

—

AcquireHandleGetReplyWait == ∃ ΞłAcquireHandle • ƒAcquireHandle ∧ PreAcquireHandleGetReplyWait ∧ PostAcquireHandleGetReplyWait

└

┌ PreAcquireHandleGetReqId

PrełAcquireHandleGetReqId

ĐAcquireHandle

|

ℝAcquireHandle

└

┌ PostAcquireHandleGetReqId

PostłAcquireHandleGetReqId

ΞĐAcquireHandle

|

ΔℝAcquireHandle

reqId! = reqId

└

—

AcquireHandleGetReqId == ∃ Ξ ł AcquireHandle • ƒ AcquireHandle ∧ PreAcquireHandleGetReqId ∧ PostAcquireHandleGetReqId

└

┌ PreAcquireHandleGetReceiver

PrełAcquireHandleGetReceiver

ĐAcquireHandle

|

ℝAcquireHandle

└

┌ PostAcquireHandleGetReceiver

PostłAcquireHandleGetReceiver

Ξ ĐAcquireHandle

|

ΔℝAcquireHandle

receiver! = receiver

└

—

AcquireHandleGetReceiver == ∃ Ξ ł AcquireHandle • ƒ AcquireHandle ∧ PreAcquireHandleGetReceiver ∧ PostAcquireHandleGetReceiver

└

┌ PreAcquireHandleGetReceiverEnd

PrełAcquireHandleGetReceiverEnd

ĐAcquireHandle

|

ℝAcquireHandle

└

┌ PostAcquireHandleGetReceiverEnd

PostłAcquireHandleGetReceiverEnd

ΞĐAcquireHandle

|

ΔℝAcquireHandle

receiverEnd! = receiverEnd

└

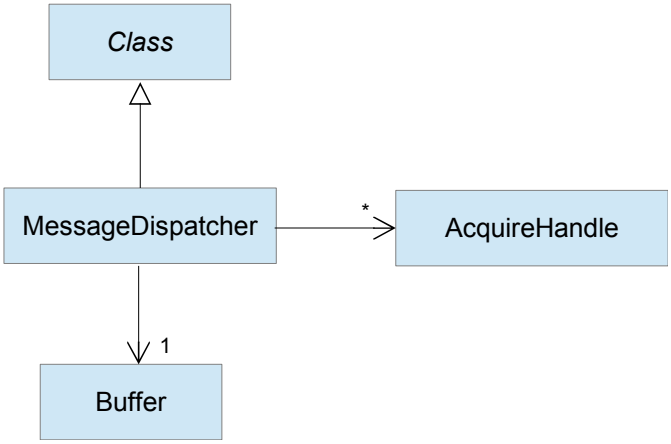
—

AcquireHandleGetReceiverEnd == ∃ ΞłAcquireHandle • ƒAcquireHandle ∧ PreAcquireHandleGetReceiverEnd ∧ PostAcquireHandleGetReceiverEnd

└

4.8.2.10 La nozione di MessageDispatcher

Un'altra problematica presente nella realizzazione dell'astrazione request-reply, è che dal canale di ingresso dell'end point associato a tale sessione, è possibile ricevere sia richieste, sia risposte a richieste effettuate. A tal fine si definisce l'ulteriore meccanismo MessageDispatcher, il quale consente di dividere le richieste ricevute dalle risposte, memorizzando le prime in un buffer e riponendo le seconde all'interno delle corrette AcquireHandle, predisposte a seguito delle corrispondenti richieste.



— section AnalysisConcepts parents BaseConcepts

⌙

. . .

⌈

MESSAGE_DISPATCHER : ℙ INSTANCE

⌋

isA(MessageDispatcher) = Class

extension(⦇{MessageDispatcher}⦈) = MESSAGE_DISPATCHER

¬ abstract MessageDispatcher

⌙

— section MessageDispatcherIntensional parents ClassIntensional

⌙

Questa classe utilizza una funzione “handle” all'interno dell'intensione di stato ⨍MessageDispatcher per tenere traccia delle richieste effettuate in attesa di risposta, in base alla destinazione (indirizzo + end number) ed al numero di richiesta. E' inoltre presente un buffer per ospitare le richieste ricevute in attesa di essere consegnate al livello applicativo.

┌─ łMessageDispatcher
 łClass
 handle : ENTITYID × ℕ↘1↖ × ℕ ↣ ACQUIRE_HANDLE
 requests : BUFFER
|
 this ∈ MESSAGE_DISPATCHER
└

┌─ łMessageDispatcherInit
 łClassInit
 łMessageDispatcher’
└

┌─ łMessageDispatcherFin
 łClassFin
 łMessageDispatcher
└

L'evento DispatcherDispatch classifica un messaggio come richiesta o come risposta a una richiesta effettuata ed aggiorna le strutture dati della classe corrispondentemente.

┌─ PrełMessageDispatcherDispatch
 łMessageDispatcher
 m? : MESSAGE
└

┌─ PostłMessageDispatcherDispatch
 ΔłMessageDispatcher
 PrełMessageDispatcherDispatch
└

GetRequest consente l'accesso alle richieste ricevute da processare.

```
┌─ PrełMessageDispatcherGetRequests
  łMessageDispatcher
└─
┌─ PostłMessageDispatcherGetRequests
  ΞłMessageDispatcher
  PrełMessageDispatcherGetRequests
  requests! : BUFFER
└─
```

PutHandle consente di registrare una AcquireHandle nel Dispatcher.

```
┌─ PrełMessageDispatcherPutHandle
  łMessageDispatcher
  h? : ACQUIRE_HANDLE
└─
┌─ PostłMessageDispatcherPutHandle
  ΔłMessageDispatcher
  PrełMessageDispatcherPutHandle
└─
```

```
— section MessageDispatcherRelational parents
MessageDispatcherIntensional , BufferRelational , AcquireHandleRelational
└─
```

. . .

Qui si asserisce che gli oggetti su cui opera il MessageDispatcher non sono mai nulli. Si afferma inoltre che il buffer requests contiene tutte istanze distinte di messaggi e che le uniche handle di cui si tiene traccia sono quelle relative a richieste ancora in attesa di risposta. Le AcquireHandle sono mappate in modo consistente ai loro dati identificativi dalla funzione handle.

```
┌─ ℝMessageDispatcher
```

```
  ℝClass
  łMessageDispatcher
  ΩBuffer[MESSAGE]
  ΩMessage
  ΩAcquireHandle
  ΩMessageDispatcher
|
  ran handle ⊆ dom ŁAcquireHandle
  requests ∈ dom ŁBuffer
  ran (ŁBuffer requests).dataQueue ⊆ dom ŁMessage
  (ŁBuffer requests).dataQueue ∈ iseq MESSAGE
  ∀ h : ran handle • (ŁAcquireHandle h).reply = ∅
  ∀ entry : dom handle • (ŁAcquireHandle (handle entry)).receiver = entry.1 ∧
                         (ŁAcquireHandle (handle entry)).receiverEnd = entry.2
∧
                         (ŁAcquireHandle (handle entry)).reqId = entry.3
└
```

```
— section MessageDispatcherExtensional parents MessageDispatcherRelational
, BufferExtensional , AcquireHandleExtensional
└
```

. . .

```
┌─ ÐMessageDispatcher
  ÐBuffer[MESSAGE]
  ÐAcquireHandle
  ÆMessageDispatcher
```

|

MessageDispatcherIsClass

└

. . .

┌─ MessageDispatcherInit

łMessageDispatcherInit

ΔĐMessageDispatcher

|

ℝMessageDispatcher’

ClassInit

this? ∈ MESSAGE_DISPATCHER

handle’ = ∅

∃ oc! : CONDITION

• ConditionNew ⨟

BufferNew[requests’/ob!,oc!/condition?]

└

—

MessageDispatcherNew == ∃ łMessageDispatcher’ • ƒMessageDispatcherN

∧ MessageDispatcherInit[om!/this?]

└

┌─ MessageDispatcherFin

łMessageDispatcherFin

ΔĐMessageDispatcher

|

ℝMessageDispatcher

ClassFin

handle = ∅

#(ŁCondition (ŁBuffer requests).dataWait).activityQueue = 0

└

—

MessageDispatcherDelete == ∃ łMessageDispatcher • ƒMessageDispatcherD

∧ MessageDispatcherFin

└

┌─ PreMessageDispatcherDispatchReply

PrełMessageDispatcherDispatch

ĐMessageDispatcher

|

ℝMessageDispatcher

m? ∈ dom ŁMessage

m? ∉ ran (ŁBuffer requests).dataQueue

((ŁMessage m?).sender,(ŁMessage m?).senderEnd,(ŁMessage m?).reqId) ∈

dom handle

└

┌─ PostMessageDispatcherDispatchReply

PostłMessageDispatcherDispatch

ΔĐMessageDispatcher

|

ΔℝMessageDispatcher

this′ = this

requests′ = requests

∃ sender! : ENTITYID ; senderEnd! : ℕ↘1↖ ; reqId! : ℕ ; h : ACQUIRE_HANDLE

| h = handle(sender!,senderEnd!,reqId!)

• MessageGetSender[m?/om?] ⨟

MessageGetSenderEnd[m?/om?] ⨟
MessageGetReqId[m?/om?] ⨟
AcquireHandlePutReply[h/oa?,m?/r?] ∧
[| handle′ = {(sender!,senderEnd!,reqId!)} ⩤ handle]

└

┌─ PreMessageDispatcherDispatchRequest
PreŁMessageDispatcherDispatch
ĐMessageDispatcher
|
ℝMessageDispatcher
m? ∈ dom ŁMessage
m? ∉ ran (ŁBuffer requests).dataQueue
((ŁMessage m?).sender,(ŁMessage m?).senderEnd,(ŁMessage m?).reqId) ∉ dom handle

└

┌─ PostMessageDispatcherDispatchRequest
PostŁMessageDispatcherDispatch
ΔĐMessageDispatcher
|
ΔℝMessageDispatcher
this′ = this
requests′ = requests
handle′ = handle
BufferEnqueue[requests/ob?,m?/data?]

└

—

MessageDispatcherDispatch ==
∃ ΔŁMessageDispatcher

• ƒMessageDispatcher ∧ ((PreMessageDispatcherDispatchReply ∧ PostMessageDispatcherDispatchReply) ∨
(PreMessageDispatcherDispatchRequest ∧ PostMessageDispatcherDispatchRequest))
└

┌─ PreMessageDispatcherGetRequests
PrełMessageDispatcherGetRequests
ĐMessageDispatcher
|
ℝMessageDispatcher
└

┌─ PostMessageDispatcherGetRequests
PostłMessageDispatcherGetRequests
ΞĐMessageDispatcher
|
ΔℝMessageDispatcher
requests! = requests
└

—
MessageDispatcherGetRequests ==
∃ ΞłMessageDispatcher • ƒMessageDispatcher ∧ PreMessageDispatcherGetRequests ∧ PostMessageDispatcherGetRequests
└

┌─ PreMessageDispatcherPutHandle
PrełMessageDispatcherPutHandle
ĐMessageDispatcher

|

ℝMessageDispatcher

h? ∈ dom ŁAcquireHandle

(ŁAcquireHandle h?).reply = Ø

((ŁAcquireHandle h?).receiver,(ŁAcquireHandle h?).receiverEnd,

(ŁAcquireHandle h?).reqId) ∉ dom handle

└

┌ PostMessageDispatcherPutHandle

PostłMessageDispatcherPutHandle

ΔĐMessageDispatcher

|

ΔℝMessageDispatcher

this′ = this

requests′ = requests

∃ receiver! : ENTITYID ; receiverEnd! : ℕ↘1↖ ; reqId! : ℕ

• AcquireHandleGetReceiver[h?/oa?] ⨟

AcquireHandleGetReceiverEnd[h?/oa?] ⨟

AcquireHandleGetReqId[h?/oa?] ∧

[| handle′ = handle ∪ {(receiver!,receiverEnd!,reqId!) ↦ h?}]

└

—

MessageDispatcherPutHandle == ∃ ΔłMessageDispatcher •

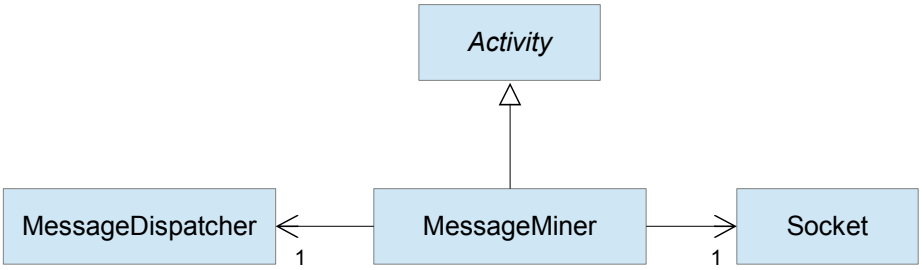
ƒMessageDispatcher ∧ PreMessageDispatcherPutHandle ∧

PostMessageDispatcherPutHandle

└

4.8.2.11 La nozione di MessageMiner

Si vuole dare la possibilità di ricevere richieste in modo bloccante e non bloccante. Allo stesso tempo occorre essere in grado di recapitare le risposte alle richieste precedenti nelle rispettive acquire handle, dove potrebbero esserci altre attività in attesa. Al fine di disaccoppiare il flusso di ricezione di messaggi provenienti dall'end point, da quello delle attività che desiderano ritirare tali messaggi, si definisce un'attivita (MessageMiner), incaricata di estrarre messaggi dal canale di ingresso di un end point e consegnarli ad un MessageDispatcher, indipendentemente dalle attività che utilizzano la sessione.
Un'attività di tipo MessageMiner, se non sono disponibili messaggi, si sospende in attesa di essi. Se la sessione viene chiusa (e di conseguenza tutte le attività in attesa svegliate), il MessageMiner termina l'esecuzione (si veda l'evento Run).




```
— section AnalysisConcepts parents BaseConcepts
└
        . . .
|
  MessageMiner : CLASS
  MESSAGE_MINER : ℙ ACTIVITY
|
  isA(MessageMiner) = Activity
  extension(⦇{MessageMiner}⦈) = MESSAGE_MINER
  ¬ abstract MessageMiner
└
```

— section MessageMinerIntensional parents ActivityIntensional

└

┌ łMessageMiner

łActivity

s : SOCKET

d : MESSAGE_DISPATCHER

|

this ∈ MESSAGE_MINER

└

┌ łMessageMinerInit

łActivityInit

łMessageMiner ’

s? : SOCKET

d? : MESSAGE_DISPATCHER

└

┌ łMessageMinerFin

łActivityFin

łMessageMiner

└

┌ PrełMessageMinerRun

PrełActivityRun

łMessageMiner

└

┌─ PostłMessageMinerRun

PostłActivityRun

ΔłMessageMiner

PrełMessageMinerRun

└

┌─ PrełMessageMinerStart

PrełActivityStart

łMessageMiner

└

┌─ PostłMessageMinerStart

PostłActivityStart

ΔłMessageMiner

PrełMessageMinerStart

└

┌─ PrełMessageMinerSetExecutionState

PrełActivitySetExecutionState

łMessageMiner

└

┌─ PostłMessageMinerSetExecutionState

PostłActivitySetExecutionState

ΔłMessageMiner

PrełMessageMinerSetExecutionState

└

┌─ PrełMessageMinerGetExecutionState
PrełActivityGetExecutionState
łMessageMiner
└

┌─ PostłMessageMinerGetExecutionState
PostłActivityGetExecutionState
≡łMessageMiner
PrełMessageMinerGetExecutionState
└

— section MessageMinerRelational parents MessageMinerIntensional , SocketRelational , MessageDispatcherRelational
└

. . .

┌─ ℝMessageMiner
ℝActivity
łMessageMiner
ΩBuffer[MESSAGE]
ΩPipe
ΩSocket
ΩMessageDispatcher
ΩMessageMiner
|
s ∈ dom ŁSocket
d ∈ dom ŁMessageDispatcher
└

— section MessageMinerExtensional parents MessageMinerRelational , SocketExtensional , MessageDispatcherExtensional

└

. . .

┌─ ĐMessageMiner
ĐSocket
ĐMessageDispatcher
ÆMessageMiner
|
MessageMinerIsActivity
└

. . .

┌─ MessageMinerInit
łMessageMinerInit
ΔĐMessageMiner
|
ℝMessageMiner ’
ActivityInit
this? ∈ MESSAGE_MINER
s? ∈ dom ŁSocket
d? ∈ dom ŁMessageDispatcher
s’ = s?
d’ = d?
└

—

MessageMinerNew == ∃ łMessageMiner′ • ƒMessageMinerN ∧
MessageMinerInit[om!/this?]
└

┌─ MessageMinerFin
łMessageMinerFin
ΔĐMessageMiner
|
ℝMessageMiner
ActivityFin
└
—
MessageMinerDelete == ∃ łMessageMiner • ƒMessageMinerD ∧
MessageMinerFin
└

┌─ PreMessageMinerRunSuccess
PrełMessageMinerRun
ĐMessageMiner
|
ℝMessageMiner
PreActivityRun
(ŁBuffer (ŁPipe (ŁSocket s).in).buffer).dataQueue ≠ ⟨ ⟩
└
┌─ PostMessageMinerRunSuccess
PostłMessageMinerRun
ΔĐMessageMiner
|

ΔℝMessageMiner

PostActivityRun

s′ = s

d′ = d

∃ m! : □□MESSAGE ; m : MESSAGE

| m! = {m}

• SocketReceive[s/os?] ⨟

MessageDispatcherDispatch[d/om?,m/m?] ∧

[| executionState′ = RUNNABLE]

└

┌─ PreMessageMinerRunNoMessageSocketOpen

PreŁMessageMinerRun

ÐMessageMiner

|

ℝMessageMiner

PreActivityRun

(ŁBuffer (ŁPipe (ŁSocket s).in).buffer).dataQueue = ⟨ ⟩

(ŁPipe (ŁSocket s).in).isClosed = False

└

┌─ PostMessageMinerRunNoMessageSocketOpen

PostŁMessageMinerRun

ΔÐMessageMiner

|

ΔℝMessageMiner

PostActivityRun

s′ = s

d′ = d

∃ m! : □□MESSAGE ; isInClosed! : □□; inWait! : CONDITION

| m! = Ø ∧ isInClosed! = False

• SocketReceive[s/os?] ⨟

SocketIsInClosed[s/os?] ⨟

SocketGetInWait[s/os?] ⨟

ConditionAwait[inWait!/oc?,this/a?]

└

┌─ PreMessageMinerRunNoMessageSocketClosed

PrełMessageMinerRun

ÐMessageMiner

|

ℝMessageMiner

PreActivityRun

(ŁBuffer (ŁPipe (ŁSocket s).in).buffer).dataQueue = ⟨ ⟩

(ŁPipe (ŁSocket s).in).isClosed = True

└

┌─ PostMessageMinerRunNoMessageSocketClosed

PostłMessageMinerRun

ΔÐMessageMiner

|

ΔℝMessageMiner

PostActivityRun

s′ = s

d′ = d

∃ m! : □□MESSAGE ; isInClosed! : □□

| m! = Ø ∧ isInClosed! = True

• SocketReceive[s/os?] ⨟

SocketIsInClosed[s/os?] ∧

[| executionState′ = TERMINATED]

└

—

MessageMinerRun == ∃ ΔłMessageMiner

• ƒMessageMiner ∧ ((PreMessageMinerRunSuccess ∧

PostMessageMinerRunSuccess) ∨

(PreMessageMinerRunNoMessageSocketOpen ∧

PostMessageMinerRunNoMessageSocketOpen) ∨

(PreMessageMinerRunNoMessageSocketClosed ∧

PostMessageMinerRunNoMessageSocketClosed))

└

┌─ PreMessageMinerStart

PrełMessageMinerStart

ĐMessageMiner

|

ℝMessageMiner

PreActivityStart

└

┌─ PostMessageMinerStart

PostłMessageMinerStart

ΔĐMessageMiner

|

ΔℝMessageMiner

PostActivityStart

s′ = s

d′ = d

└

—

MessageMinerStart == ∃ ΔłMessageMiner • ƒMessageMiner ∧
PreMessageMinerStart ∧ PostMessageMinerStart
└

┌ PreMessageMinerSetExecutionState
PrełMessageMinerSetExecutionState
ĐMessageMiner
|
ℝMessageMiner
PreActivitySetExecutionState
└
┌ PostMessageMinerSetExecutionState
PostłMessageMinerSetExecutionState
ΔĐMessageMiner
|
ΔℝMessageMiner
PostActivitySetExecutionState
s′ = s
d′ = d
└
—
MessageMinerSetExecutionState == ∃ ΔłMessageMiner • ƒMessageMiner ∧
PreMessageMinerSetExecutionState ∧ PostMessageMinerSetExecutionState
└

┌ PreMessageMinerGetExecutionState
PrełMessageMinerGetExecutionState
ĐMessageMiner

|

ℝMessageMiner

PreActivityGetExecutionState

└

┌ PostMessageMinerGetExecutionState

PostłMessageMinerGetExecutionState

ΞŁĐMessageMiner

|

ΔℝMessageMiner

PostActivityGetExecutionState

└

—

MessageMinerGetExecutionState == ∃ ΞŁłMessageMiner • ƒMessageMiner ∧ PreMessageMinerGetExecutionState ∧ PostMessageMinerGetExecutionState

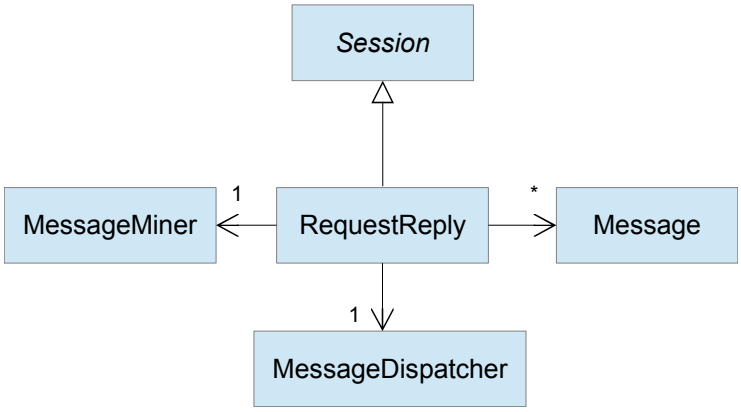
└

4.8.2.12 La nozione di RequestReply

Siamo ora in grado di definire la terza categoria di interazione in modo preciso. Come detto a causa delle necessità di tale categoria, si possono identificare tre diversi “serbatoi” di messaggi: le AcquireHandle predisposte ad ospitare le risposte alle richieste effettuate, le richieste ricevute in attesa di essere ritirate dal livello applicativo, e le richieste consegnate al livello applicativo a cui esso non ha ancora risposto. Oltre a questi ci sono ovviamente i canali di ingresso e uscita della Socket associata alla sessione, che come per le altre due categorie di interazione, vengono gestiti in modo trasparente, anche grazie ad un MessageMiner. Dopo aver creato una sessione di tipo RequestReply, è dunque possibile inviare richieste (evento Ask) e ricevere come risultato una AcquireHandle, che consente di effettuare il polling di una risposta o bloccarsi nell'attesa di essa. E' possibile ricevere una richiesta in modo bloccante o non bloccante (eventi Grant e GetGrantWait), rispondere a una richiesta ricevuta (evento Answer) e chiudere la sessione (evento Close).

E' inoltre possibile dichiarare la volontà di non rispondere a una certa richiesta (evento ClearRequest).

Si noti che in questa categoria di sessione, gli oggetti Message vengono gestiti trasparentemente come nelle altre due. Tuttavia mentre in fase di invio di una richiesta, il sequence number e l'identificatore di richiesta (reqId) di un messaggio coincidono, il messaggio di risposta ad una richiesta, avrà il reqId della richiesta che l'ha generata. Il suo sequence number invece sarà progressivo e dipendente dal numero di messaggi già inviati verso quella destinazione.




```
— section AnalysisConcepts parents BaseConcepts
└
        . . .
|
  RequestReply : CLASS
  REQUEST_REPLY : ℙ SESSION
|
  isA(RequestReply) = Session
  extension(⦇{RequestReply}⦈) = REQUEST_REPLY
  ¬ abstract RequestReply
└

— section RequestReplyIntensional parents SessionIntensional
```

└

┌─ łRequestReply
 łSession
 dispatcher : MESSAGE_DISPATCHER
 pendingReceivedRequests : □□MESSAGE
 miner : MESSAGE_MINER
|
 this ∈ REQUEST_REPLY
 ∀ d : dom sequenceNumbers • d.2 > 0
└

┌─ łRequestReplyInit
 łSessionInit
 łRequestReply ’
└

┌─ łRequestReplyFin
 łSessionFin
 łRequestReply
└

┌─ PrełRequestReplyGetOwner
 PrełSessionGetOwner
 łRequestReply
└
┌─ PostłRequestReplyGetOwner
 PostłSessionGetOwner

Ξ łRequestReply

PrełRequestReplyGetOwner

└

┌ PrełRequestReplyGetEndNumber

PrełSessionGetEndNumber

łRequestReply

└

┌ PostłRequestReplyGetEndNumber

PostłSessionGetEndNumber

Ξ łRequestReply

PrełRequestReplyGetEndNumber

└

┌ PrełRequestReplyAsk

łRequestReply

receiver? : ENTITYID

receiverEnd? : ℕ↘1↖

data? : seq SYMBOL

└

┌ PostłRequestReplyAsk

ΔłRequestReply

PrełRequestReplyAsk

h! : ACQUIRE_HANDLE

└

┌ PrełRequestReplyGrant

łRequestReply

└

┌─ PostłRequestReplyGrant
  ΔłRequestReply
  PrełRequestReplyGrant
  m! : □□MESSAGE
└

┌─ PrełRequestReplyGetGrantWait
  łRequestReply
└

┌─ PostłRequestReplyGetGrantWait
  ΞłRequestReply
  PrełRequestReplyGetGrantWait
  grantWait! : CONDITION
└

┌─ PrełRequestReplyAnswer
  łRequestReply
  request? : MESSAGE
  data? : seq SYMBOL
└

┌─ PostłRequestReplyAnswer
  ΔłRequestReply
  PrełRequestReplyAnswer
└

┌─ PrełRequestReplyClearRequest
  łRequestReply

request? : MESSAGE

└

┌─ Post‡RequestReplyClearRequest

Δ‡RequestReply

Pre‡RequestReplyClearRequest

└

┌─ Pre‡RequestReplyClose

‡RequestReply

└

┌─ Post‡RequestReplyClose

Δ‡RequestReply

Pre‡RequestReplyClose

└

┌─ Pre‡RequestReplyIsClosed

‡RequestReply

└

┌─ Post‡RequestReplyIsClosed

Ξ‡RequestReply

Pre‡RequestReplyIsClosed

isClosed! : □□

└

— section RequestReplyRelational parents RequestReplyIntensional , SessionRelational , MessageMinerRelational

└

. . .

┌─ ℝRequestReply
ℝSession
łRequestReply
ΩMessageDispatcher
ΩMessageMiner
ΩRequestReply
|
dispatcher ∈ dom ŁMessageDispatcher
pendingReceivedRequests ⊆ dom ŁMessage
miner ∈ dom ŁMessageMiner
(ŁSocket endPoint).endNumber > 0
(ŁMessageMiner miner).s = endPoint
(ŁMessageMiner miner).d = dispatcher
pendingReceivedRequests ∩ ran (ŁBuffer (ŁMessageDispatcher dispatcher).requests).dataQueue = Ø
└─

— section RequestReplyExtensional parents RequestReplyRelational , SessionExtensional , MessageMinerExtensional
└─

. . .

┌─ ÐRequestReply
ÐSession
ÐMessageMiner
ÆRequestReply

|

RequestReplyIsSession

└

. . .

┌ RequestReplyInit

łRequestReplyInit

ΔĐRequestReply

|

ℝRequestReply ′

SessionInit

this? ∈ REQUEST_REPLY

endNumber? > 0

∀ d : dom sequenceNumbers? • d.2 > 0

pendingReceivedRequests′ = ∅

MessageDispatcherNew[dispatcher′/om!] ⨟

MessageMinerNew[miner′/om!,endPoint′/s?,dispatcher′/d?] ⨟

MessageMinerStart[miner′/om?]

└

—

RequestReplyNew == ∃ łRequestReply ′ • ƒRequestReplyN ∧

RequestReplyInit[or!/this?]

└

┌ RequestReplyFin

łRequestReplyFin

ΔĐRequestReply

|

ℝRequestReply

SessionFin

#(ŁCondition (ŁBuffer (ŁMessageDispatcher dispatcher).requests).dataWait).activityQueue = 0

└

—

RequestReplyDelete == ∃ łRequestReply • ƒRequestReplyD ∧ RequestReplyFin

└

┌ PreRequestReplyGetOwner

PrełRequestReplyGetOwner

ĐRequestReply

|

ℝRequestReply

PreSessionGetOwner

└

┌ PostRequestReplyGetOwner

PostłRequestReplyGetOwner

ΞĐRequestReply

|

ΔℝRequestReply

PostSessionGetOwner

└

—

RequestReplyGetOwner == ∃ ΞłRequestReply • ƒRequestReply ∧ PreRequestReplyGetOwner ∧ PostRequestReplyGetOwner

└

```
┌─ PreRequestReplyGetEndNumber
  PrełRequestReplyGetEndNumber
  ĐRequestReply
|
  ℝRequestReply
  PreSessionGetEndNumber
└
┌─ PostRequestReplyGetEndNumber
  PostłRequestReplyGetEndNumber
  ΞłĐRequestReply
|
  ΔℝRequestReply
  PostSessionGetEndNumber
└
—
  RequestReplyGetEndNumber == ∃ ΞłRequestReply • ƒRequestReply ∧
PreRequestReplyGetEndNumber ∧ PostRequestReplyGetEndNumber
└


┌─ PreRequestReplyAsk
  PrełRequestReplyAsk
  ĐRequestReply
|
  ℝRequestReply
  receiver? ≠ eAll
  (ŁPipe (ŁSocket endPoint).out).isClosed = False
└
```

┌─ PostRequestReplyAsk

PostłRequestReplyAsk

ΔÐRequestReply

|

ΔℝRequestReply

this′ = this

endPoint′ = endPoint

channel′ = channel

(receiver?,receiverEnd?) ∈ dom sequenceNumbers

⇒ sequenceNumbers′ = sequenceNumbers ⊕ {(receiver?,receiverEnd?) ↦ sequenceNumbers(receiver?,receiverEnd?) + 1}

(receiver?,receiverEnd?) ∉ dom sequenceNumbers

⇒ sequenceNumbers′ = sequenceNumbers ∪ {(receiver?,receiverEnd?) ↦ 1}

dispatcher′ = dispatcher

pendingReceivedRequests′ = pendingReceivedRequests

miner′ = miner

∃ owner! : ENTITYID ; endNumber! : ℕ↘1↖ ; sequenceNumber? : ℕ ; om! : MESSAGE

| ((receiver?,receiverEnd?) ∈ dom sequenceNumbers ⇒ sequenceNumber? = sequenceNumbers(receiver?,receiverEnd?)) ∧

((receiver?,receiverEnd?) ∉ dom sequenceNumbers ⇒ sequenceNumber? = 0)

• SocketGetOwner[endPoint/os?] ⨟

SocketGetEndNumber[endPoint/os?] ⨟ MessageNew[owner!/sender?,endNumber!/senderEnd?,sequenceNumber?/reqId?] ⨟

AcquireHandleNew[h!/oa!,sequenceNumber?/reqId?] ⨟

MessageDispatcherPutHandle[dispatcher/om?,h!/h?] ⨟

SocketSend[endPoint/os?,om!/m?]

└

—

RequestReplyAsk == ∃ ΔłRequestReply • ƒRequestReply ∧ PreRequestReplyAsk ∧ PostRequestReplyAsk

└

┌ PreRequestReplyGrant

PrełRequestReplyGrant

ĐRequestReply

|

ℝRequestReply

└

┌ PostRequestReplyGrant

PostłRequestReplyGrant

ΔĐRequestReply

|

ΔℝRequestReply

this′ = this

endPoint′ = endPoint

channel′ = channel

sequenceNumbers′ = sequenceNumbers

dispatcher′ = dispatcher

pendingReceivedRequests′ = pendingReceivedRequests ∪ m!

miner′ = miner

#m! ∈ 0 .. 1

∃ requests! : BUFFER

• MessageDispatcherGetRequests[dispatcher/om?] ⨟

BufferDequeue[requests!/ob?,m!/data!]

└

—

RequestReplyGrant == ∃ ΔłRequestReply • ƒRequestReply ∧ PreRequestReplyGrant ∧ PostRequestReplyGrant

└

┌─ PreRequestReplyGetGrantWait

PrełRequestReplyGetGrantWait

ĐRequestReply

|

ℝRequestReply

└

┌─ PostRequestReplyGetGrantWait

PostłRequestReplyGetGrantWait

ΞĐRequestReply

|

ΔℝRequestReply

∃ requests! : BUFFER

• MessageDispatcherGetRequests[dispatcher/om?] ⨟

BufferGetDataWait[requests!/ob?,grantWait!/dataWait!]

└

—

RequestReplyGetGrantWait == ∃ ΞłRequestReply • ƒRequestReply ∧ PreRequestReplyGetGrantWait ∧ PostRequestReplyGetGrantWait

└

┌─ PreRequestReplyAnswer

PreŁRequestReplyAnswer

ĐRequestReply

|

ℝRequestReply

request? ∈ pendingReceivedRequests

(ŁPipe (ŁSocket endPoint).out).isClosed = False

└

┌─ PostRequestReplyAnswer

PostŁRequestReplyAnswer

ΔĐRequestReply

|

ΔℝRequestReply

this′ = this

endPoint′ = endPoint

channel′ = channel

dispatcher′ = dispatcher

pendingReceivedRequests′ = pendingReceivedRequests ⧹ {request?}

miner′ = miner

∃ owner! , receiver? : ENTITYID ; endNumber! , receiverEnd? : ℕ↘1↖ ; sequenceNumber? , reqId? : ℕ ; om! : MESSAGE

| ((receiver?,receiverEnd?) ∈ dom sequenceNumbers ⇒ sequenceNumber? = sequenceNumbers(receiver?,receiverEnd?)) ∧

((receiver?,receiverEnd?) ∉ dom sequenceNumbers ⇒ sequenceNumber? = 0)

• SocketGetOwner[endPoint/os?] ⨟

SocketGetEndNumber[endPoint/os?] ⨟

MessageGetSender[request?/om?,receiver?/sender!] ⨟

MessageGetSenderEnd[request?/om?,receiverEnd?/senderEnd!] ⨟

MessageGetReqId[request?/om?,reqId?/reqId!] ⨟

MessageNew[owner!/sender?,endNumber!/senderEnd?] ⨟

SocketSend[endPoint/os?,om!/m?] ∧

[| (receiver?,receiverEnd?) ∈ dom sequenceNumbers

⇒ sequenceNumbers′ = sequenceNumbers ⊕ {(receiver?,receiverEnd?) ↦ sequenceNumbers(receiver?,receiverEnd?) + 1}

(receiver?,receiverEnd?) ∉ dom sequenceNumbers

⇒ sequenceNumbers′ = sequenceNumbers ∪ {(receiver?,receiverEnd?) ↦ 1}]

└

—

RequestReplyAnswer == ∃ ΔłRequestReply • ƒRequestReply ∧ PreRequestReplyAnswer ∧ PostRequestReplyAnswer

└

┌─ PreRequestReplyClearRequest

PrełRequestReplyClearRequest

ĐRequestReply

|

ℝRequestReply

└

┌─ PostRequestReplyClearRequest

PostłRequestReplyClearRequest

ΔĐRequestReply

|

ΔℝRequestReply

this′ = this

endPoint′ = endPoint

channel′ = channel

sequenceNumbers′ = sequenceNumbers

dispatcher′ = dispatcher

pendingReceivedRequests′ = pendingReceivedRequests ⧵ {request?}

miner′ = miner

└

—

RequestReplyClearRequest == ∃ ΔłRequestReply • ƒRequestReply ∧ PreRequestReplyClearRequest ∧ PostRequestReplyClearRequest

└

┌ PreRequestReplyClose

PrełRequestReplyClose

ĐRequestReply

|

ℝRequestReply

pendingReceivedRequests = ∅

└

┌ PostRequestReplyClose

PostłRequestReplyClose

ΔĐRequestReply

|

ΔℝRequestReply

this′ = this

endPoint′ = endPoint

channel′ = channel

sequenceNumbers′ = sequenceNumbers

dispatcher′ = dispatcher

pendingReceivedRequests′ = pendingReceivedRequests

miner′ = miner

SocketCloseIn[endPoint/os?] ⨟

SocketCloseOut[endPoint/os?]

└

—

RequestReplyClose == ∃ ΔłRequestReply • ƒRequestReply ∧ PreRequestReplyClose ∧ PostRequestReplyClose

└

┌ PreRequestReplyIsClosed

PrełRequestReplyIsClosed

ĐRequestReply

|

ℝRequestReply

└

┌ PostRequestReplyIsClosed

PostłRequestReplyIsClosed

ΞĐRequestReply

|

ΔℝRequestReply

SocketIsOutClosed[endPoint/os?,isClosed!/isOutClosed!]

└

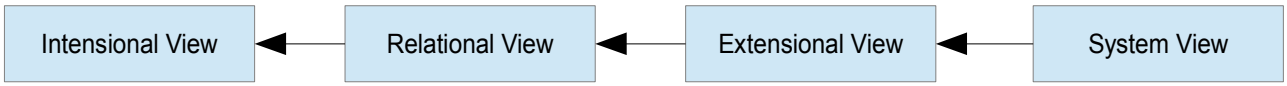
—

RequestReplyIsClosed == ∃ ΞłRequestReply • ƒRequestReply ∧ PreRequestReplyIsClosed ∧ PostRequestReplyIsClosed

└

4.9 Vista di Sistema

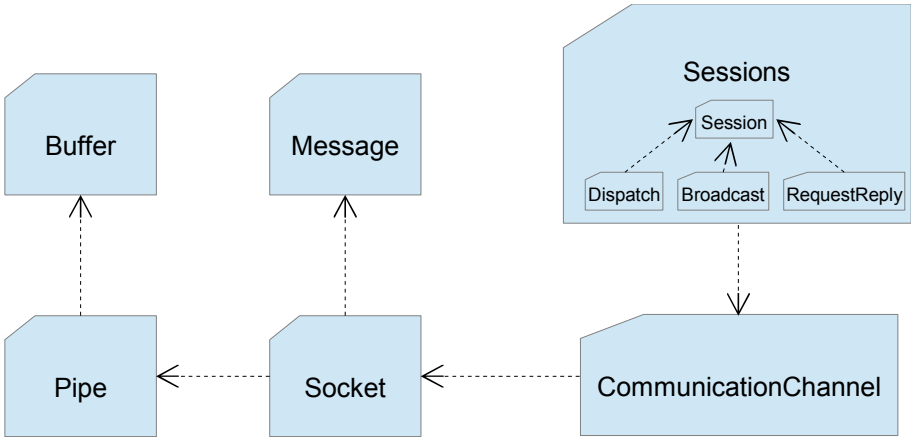
Oltre a poter specificare blocchi in modo preciso, sarebbe desiderabile avere a disposizione un modo non ambiguo per esprimere l'architettura del sistema e la sua corretta inizializzazione. A tal fine è possibile predisporre una ulteriore vista (*System View*), la quale consente di esplicitare quali sono le macro-parti del sistema,in che relazione sono e quali sono i blocchi componenti di ognuna di esse. Nella vista di sistema è inoltre possibile specificare quali sono gli eventi che avvengono al sistema nel suo complesso,in che modo si mappano sui blocchi definiti e qual'è l'evento di inizializzazione del sistema. L'architettura complessiva di una specifica in Concepts-Z è dunque la seguente (dove le frecce indicano la relazione di dipendenza):



La system view denota il modello del sistema nel suo complesso e rappresenta quindi l'artefatto dell'analisi nella sua interezza (modello di analisi). Al suo interno sono espressi i vincoli architetturali che dovranno essere preservati dalla successiva fase del processo di sviluppo. Questo consente inoltre, in linea di principio, di stabilire relazioni con il precedente modello del sistema (in questa fase il modello dei requisiti).

Per modellare le macro-parti del sistema si utilizzerà il concetto di *package*.
Si definisce package un contenitore in grado di ospitare classi oppure altri package, ma non entrambe le cose contemporaneamente. Si dice che un package *dipende* da un altro, se almeno una delle classi in esso contenute, dipende da una classe ospitata nell'altro. Un package che contiene altri package, può quindi finire per avere una dipendenza verso un package esterno, a causa di uno di quelli che contiene. Ogni package dipende da se stesso.

A titolo di esempio, si introdurrà ora una possibile architettura per il middleware per le interazioni definito in precedenza (nella figura i riquadri a forma di scheda indicano i package e le frecce tratteggiate indicano le relazioni di dipendenza fra essi).



— section MiddlewareSystem parents DispatchExtensional , BroadcastExtensional , RequestReplyExtensional

⌙

Un package si indica con uno schema avente nome ℙ seguito dal nome del package. Tale schema include direttamente i dependency schema di ciascuna delle entità contenute.

┌═ ℙBuffer [X]

ÐBuffer[X] (11)

⌙

┌═ ÐℙBuffer [X]

ℙBuffer[X] (12)

⌙

┌═ ℙBufferInit [X]

ÐBufferInit[X] (13)

⌙

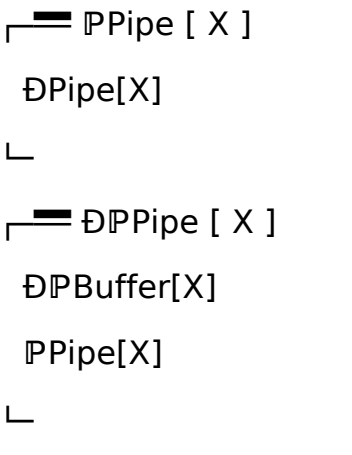
┌═ ÐℙBufferInit [X]

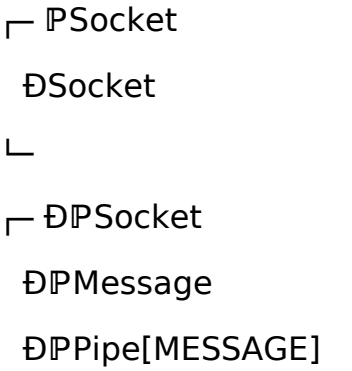
ℙBufferInit[X] (14)

⌙

Una volta definito un package occorre esplicitare il suo dependency schema, il quale

segue un meccanismo di definizione analogo al dependency schema utilizzato per le classi. Infine occorre esplicitare lo stato iniziale di questi schema in modo analogo a quanto si faceva per le classi. Ancora una volta si noti come dati (11) e (12), sia possibile ottenere automaticamente (13) e (14) (che dunque nel resto della trattazione verranno omessi).





```
 ℙSocket
└─
      . . .

┌─ ℙCommunicationChannel
 ÐCommunicationChannel
└─
┌─ ÐℙCommunicationChannel
 ÐℙSocket
 ℙCommunicationChannel
└─
      . . .

┌─ ℙSession
 ÐSession
└─
┌─ ÐℙSession
 ÐℙCommunicationChannel
 ℙSession
└─
      . . .

┌─ ℙDispatch
 ÐDispatch
└─
┌─ ÐℙDispatch
 ÐℙSession
 ℙDispatch
```

└

. . .

┌─ ℙBroadcast
 ÐBroadcast
└

┌─ ÐℙBroadcast
 ÐℙSession
 ℙBroadcast
└

. . .

┌─ ℙRequestReply
 ÐAcquireHandle
 ÐMessageDispatcher
 ÐMessageMiner
 ÐRequestReply
└

┌─ ÐℙRequestReply
 ÐℙSession
 ℙRequestReply
└

. . .

┌─ ℙSessions
 ÐℙSession
 ÐℙDispatch
 ÐℙBroadcast

ĐℙRequestReply

└

┌─ ĐℙSessions

ĐℙCommunicationChannel

ℙSessions

└

. . .

┌─ Middleware

ℙBuffer

ℙMessage

ℙPipe

ℙSocket

ℙCommunicationChannel

ℙSessions

|

ĐℙBuffer

ĐℙMessage

ĐℙPipe

ĐℙSocket

ĐℙCommunicationChannel

ĐℙSessions

└

Questo schema rappresenta il sistema nel suo complesso. Esso include direttamente i package che lo compongono. Si noti come nella parte predicativa vengano asseriti i dependency schema dei package. Uno schema utilizzato come predicato in Z, impone che tutte le variabili che esso elenca siano state definite in una dichiarazione precedente. Dunque in questo modo si sta asserendo che se un package fa parte del sistema, allora devono essere presenti anche tutti gli altri package da cui esso dipende (e per costruzione

anche tutti i blocchi di ognuno di tali package). *Si sta dunque sfruttando il parser del linguaggio Z per verificare i vincoli architetturali imposti sul sistema*.

┌─ MiddlewareInit
Middleware'
|
ĐℙBufferInit
ĐℙMessageInit
ĐℙPipeInit
ĐℙSocketInit
ĐℙCommunicationChannelInit
ĐℙSessionsInit
└─

Schema che specifica lo stato iniziale del sistema.

┌─ MiddlewareMain
ΔMiddleware
|
communicationChannels = ∅
∃ oc! : COMMUNICATION_CHANNEL • CommunicationChannelNew
└─

Questo è l'evento di inizializzazione del sistema. Nel caso in esame si limita ad instanziare il canale di comunicazione che si occuperà della trasmissione dei messaggi.

┌─ MiddlewareGetDispatch
ΔMiddleware
owner? : ENTITYID
endNumber? : ℕ
sequenceNumbers? : ENTITYID × ℕ ⇸ ℕ

od! : DISPATCH

|

communicationChannels ≠ ∅

∃ channel? : COMMUNICATION_CHANNEL

| channel? ∈ dom ŁCommunicationChannel

• DispatchNew

└

┌─ MiddlewareGetBroadcast

ΔMiddleware

owner? : ENTITYID

sequenceNumbers? : ENTITYID × ℕ ⇸ ℕ

ob! : BROADCAST

|

communicationChannels ≠ ∅

∃ channel? : COMMUNICATION_CHANNEL

| channel? ∈ dom ŁCommunicationChannel

• BroadcastNew

└

┌─ MiddlewareGetRequestReply

ΔMiddleware

owner? : ENTITYID

endNumber? : ℕ

sequenceNumbers? : ENTITYID × ℕ ⇸ ℕ

or! : REQUEST_REPLY

|

communicationChannels ≠ ∅

∃ channel? : COMMUNICATION_CHANNEL

| channel? ∈ dom ŁCommunicationChannel

• RequestReplyNew

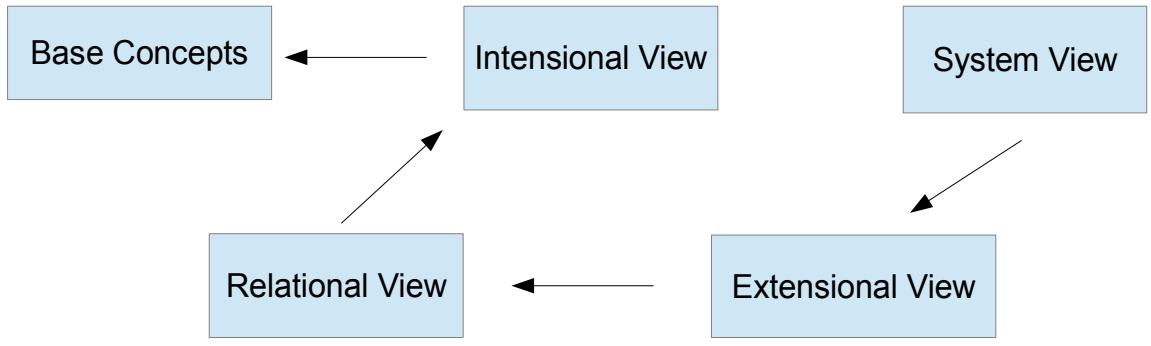
∟

Il sistema consente di ottenere degli oggetti sessione adatti ad effettuare le interazioni introdotte in precedenza. Si noti come gli eventi di servizio possano avvenire solo dopo il verificarsi dell'evento Main.

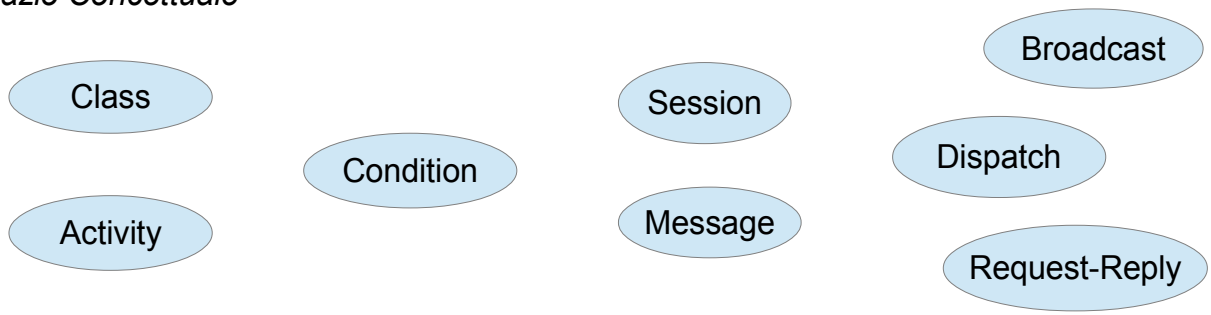
4.10 Sommario dei concetti introdotti

In questo capitolo si è introdotta una notazione basata sul linguaggio Z, che consente in modo agevole l'utilizzo di tecniche object oriented per l'analisi precisa di un sistema software. Si è poi utilizzata questa notazione per definire lo spazio concettuale messo a disposizione dell'analista da Concepts Z. Si vogliono in questo paragrafo ricapitolare i concetti introdotti, in modo da rendere chiaro ciò che si è delineato.

Architettura di una specifica in Concepts Z



Spazio Concettuale



Si è visto come la notazione, attraverso una ben determinata architettura, metta a disposizione i mezzi e una metodologia sistematica per la definizione di concetti. E' inoltre stata definita tramite essa, una ampia base concettuale, molto generale, che consente all'analista, oltre alla definizione di astrazioni, di caratterizzare precisamente le entità attive nel sistema, le interazioni fra le parti del sistema, sia concentrate che distribuite e l'architettura complessiva completa di vincoli strutturali.

Si vuole ora analizzare tutto ciò alla luce di quanto stabilito nel paragrafo 2.4. In quella sede si era affermato che un sistema è definito compiutamente quando sono specificate la sua struttura, l'interazione e il comportamento. In Concepts Z, in fase di analisi è possibile specificare la struttura del sistema tramite la sua architettura e le viste intensionali dei blocchi, l'interazione tramite l'implicazione fra eventi e le metafore introdotte per l'interazione distribuita ed il comportamento tramite le pre e post condizioni degli eventi.

Quello che si è delineato è dunque un vero e proprio "modello computazionale" di un linguaggio di modellazione generale, adeguato all'analisi precisa di sistemi software.

5
Progetto in Concepts Z

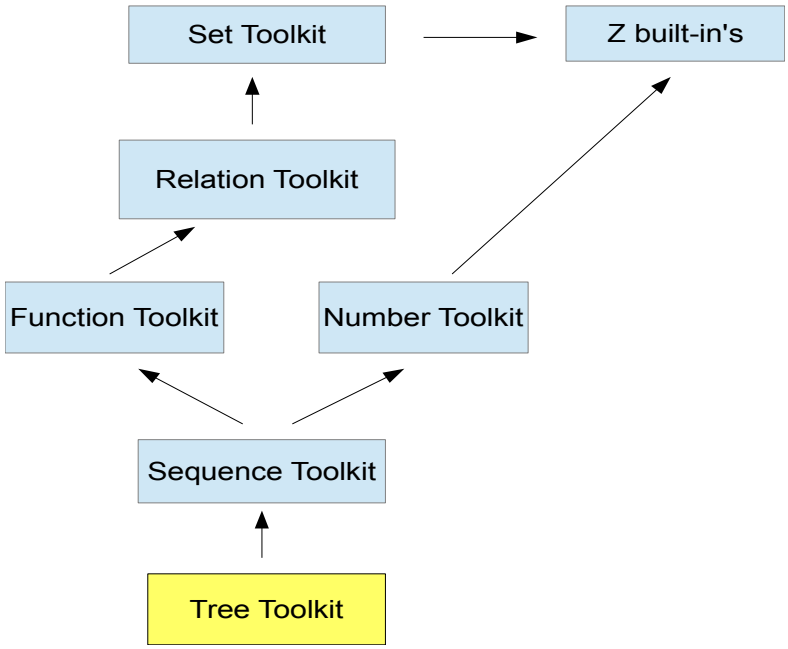
5.1 Obiettivi

Si è visto come la notazione Concepts Z costituisca un valido strumento per la conduzione dell'analisi in un processo di sviluppo. Terminata tale fase occorre raffinare il modello di analisi, introducendo informazioni aggiuntive circa la realizzazione più adeguata dei blocchi per portare a compimento quanto richiesto dall'analista.
Il comportamento di ciascun blocco è stato espresso in termini di precondizioni su uno stato prima del verificarsi di un evento e postcondizioni su uno stato dopo il verificarsi di esso.
Poiché si è detto di voler generare automaticamente il codice del sistema al termine della fase di progetto, il progettista deve, per ogni evento individuato dall'analista, stabilire la sequenza di passi più conveniente per portare a compimento quanto atteso dalla specifica di analisi dell'evento.
Inoltre, sempre per ragioni di convenienza pratica, il progettista può scegliere di adottare una differente signature per gli eventi concreti, introdurre nuovi eventi, una struttura più articolata per ciascun blocco in accordo a comprovate soluzioni efficaci, nuovi blocchi ausiliari ecc. sempre preservando i vincoli imposti dal modello di analisi.
Per portare a termine il suo compito, il progettista necessita di ulteriori concetti oltre a quelli già introdotti. Ad esempio il solo concetto di classe non è sufficiente per esprimere in generale la soluzione progettuale adottata per un blocco. Lo stesso vale per le strutture dati direttamente disponibili per esprimere la rappresentazione di stato. Inoltre la notazione introdotta nel capitolo precedente offre scarso supporto alla determinazione della realizzazione pratica di ciascun evento.
In questo capitolo estenderemo quindi la notazione Concepts Z introducendo una serie di concetti e costrutti che la rendano capace di supportare completamente anche la fase di progetto. Gli obiettivi di tale estensione sono:

- introdurre a livello primitivo strutture dati aggiuntive che riflettano precise scelte realizzative per le rappresentazioni di stato
- Introdurre concetti che consentano di esprimere in modo particolareggiato ed efficace la soluzione progettuale di un blocco
- introdurre la possibilità di esprimere la sequenza di passi più appropriata per la realizzazione di un evento

5.2 Estensione del Mathematical Toolkit di Z

Come si è detto nel paragrafo 3.1 il linguaggio Z è accompagnato da un toolkit (*standard toolkit*) il quale introduce a livello primitivo, le usuali metafore della matematica (numeri, relazioni, funzioni, sequenze ecc.) come elementi del linguaggio. Tali concetti primitivi, costituiscono un insieme sufficiente per esprimere le componenti di stato in fase di analisi. Tuttavia in fase di progetto occorrono costrutti aggiuntivi. In particolare non vi è alcun supporto diretto, praticamente utilizzabile, per esprimere la rappresentazione di una componente di stato sotto forma di *albero* e poterla manipolare.
In questo paragrafo si estenderà dunque lo standard toolkit di Z, per introdurre a livello primitivo tale struttura dati di ampio uso.
Si cercherà di rendere tale estensione (*TreeToolkit*) la più omogenea possibile con il resto dello standard toolkit e la più conforme possibile alla filosofia di definizione di quest'ultimo, in modo che l'utilizzo di un albero differisca in modo minimo da come avviene quello degli altri costrutti.



Lo standard toolkit è definito partendo da alcuni concetti “built-in” nel linguaggio Z. Con essi si definiscono gli usuali operatori dell'algebra degli insiemi, le relazioni come insiemi di coppie ordinate, le funzioni come relazioni e le sequenze come funzioni dai naturali agli elementi di un certo insieme. Si noti come la manipolazione di tali strutture segua la *semantica per valore* del linguaggio Z: un operatore che interviene su una di queste strutture sintetizza un nuovo valore. Si tratta spesso di una funzione, che associa alla struttura iniziale e ad eventuali altri argomenti, un nuovo valore che rappresenta il risultato della trasformazione voluta. *Ciò implica che gli alberi* che si andrà ad introdurre *dovranno essere dei valori*, al pari di una coppia di insiemi la cui intersezione produce un nuovo insieme. In particolare si desidera fornire supporto al concetto di grafo, di albero generico e di albero binario di ricerca.

— section TreeToolkit parents BaseConcepts

⌙

Un grafo è definito come una coppia avente come prima componente un insieme di nodi e come seconda componente un insieme di archi (ossia una relazione fra nodi dell'insieme).

— generic (graph _)

⌙

—

graph X == { c : (□□X) × (X ↔ X) | dom c.2 ⊆ c.1 ∧ ran c.2 ⊆ c.1 }

└

Le due funzioni successive, dato un grafo e un nodo appartenente ad esso, restituiscono l'insieme di archi entranti (ossia che hanno come seconda componente della coppia corrispondente all'arco il nodo) e l'insieme di archi uscenti (in cui cioè il nodo oggetto è la prima componente) di un nodo.

┌═[X]

⊙ : graph X × X ⇸ X ↔ X

⊚ : graph X × X ⇸ X ↔ X

|

dom ⊙ = dom ⊚ = { p : graph X × X | p.2 ∈ p.1.1 }

∀ p : dom ⊙ • ⊙(p) = p.1.2 ⊳ {p.2}

∀ p : dom ⊚ • ⊚(p) = {p.2} ⊲ p.1.2

└

Si tratta della *struttura ricorsiva* utilizzata per rappresentare gli alberi. Un albero è rappresentato da una funzione, con un unico mapping, da un elemento a un insieme di figli (aventi la stessa struttura) non vuoti.

— generic (treeRep _)

└

—

treeRep X == { f : X ⇻ □□treeRep X | #dom f ≤ 1 ∧ ∀ c1 , c2 : ⋃ (ran f) • c1 ≠ ∅ ∧ c2 ≠ ∅ ∧ (dom c1 ∩ dom c2 ≠ ∅ ⇔ c1 = c2) }

└

nodes e arcs sono funzioni di utilità che consentono di ottenere le componenti del grafo equivalente a una certa rappresentazione ricorsiva.
Un arco (a,b) indica che a è il padre di b (figlio).

┌═[X]

nodes : treeRep X ⟶ ℙℙX

arcs : treeRep X ⟶ X ↔ X

|

∀ t : treeRep X • (t = ∅ ⇒ nodes(t) = ∅) ∧

(t ≠ ∅ ∧ ⋃ (ran t) = ∅ ⇒ nodes(t) = dom t) ∧

(t ≠ ∅ ∧ ⋃ (ran t) ≠ ∅ ⇒ nodes(t) = dom t ∪ ⋃{ tc : ⋃ (ran t) • nodes(tc) })

∀ t : treeRep X • (t = ∅ ⇒ arcs(t) = ∅) ∧

(t ≠ ∅ ∧ ⋃ (ran t) = ∅ ⇒ arcs(t) = ∅) ∧

(t ≠ ∅ ∧ ⋃ (ran t) ≠ ∅ ⇒ arcs(t) = { r : dom t ; tc : ⋃ (ran t) ; rtc : X | dom tc = {rtc} • (r,rtc) } ∪ ⋃{ tc : ⋃ (ran t) • arcs(tc) })

└

Un albero è definito come un grafo orientato aciclico con un'unico nodo radice (nessun padre) e tutti gli altri nodi che possono avere al più un padre.

— generic (tree _)

└

—

tree X == { tr : treeRep X | arcs(tr) ↗+↙ ∩ id nodes(tr) = ∅ ∧

(tr ≠ ∅ ⇒ (∃ r : dom tr | #(⊙(nodes(tr)↦arcs(tr),r)) = 0 • ∀ n : nodes(tr) ⧵ {r} • #(⊙(nodes(tr)↦arcs(tr),n)) = 1)) }

└

L'abbreviazione seguente rappresenta gli alberi non vuoti.

—

tree↘1↖ X == tree X ⧵ {∅}

└

— relation (_ contains _)

└

La funzione *root* dato un albero consente di ottenerne la radice.
La relazione *contains* è un predicato che assume valore vero se il suo argomento sinistro (un albero), contiene il suo argomento destro (un nodo).
La funzione *parent*, dato un albero non vuoto e un nodo che non sia la sua radice, restituisce il padre di quel nodo nell'albero.
La funzione *subtree*, dato un albero non vuoto ed un nodo dell'albero, restituisce il sotto-albero avente il nodo indicato come radice.
La funzione *consTree*, dato un nodo e un insieme di alberi, costruisce un nuovo albero avente il nodo indicato come radice e gli alberi indicati come suoi figli.
La funzione *size*, dato un albero, restituisce il numero di nodi in esso contenuti.

┌═[X]

root : tree↘1↖ X ⟶ X

_ contains _ : tree↘1↖ X ↔ X

parent : tree↘1↖ X × X ⇸ X

subtree : tree↘1↖ X × X ⇸ tree↘1↖ X

consTree : X × □□tree↘1↖ X ⇸ tree↘1↖ X

size : tree X ⟶ ℕ

|

∀ t : tree↘1↖ X • root(t) ∈ dom t

∀ t : tree↘1↖ X ; e : X • t contains e ⇔ e = root(t) ∨ ∃ tc : ⋃ (ran t) • tc contains e

dom parent = { p : tree↘1↖ X × X | p.1 contains p.2 ∧ p.2 ≠ root(p.1) }

∀ p : dom parent • (p.2 ∈ { tc : ⋃ (ran p.1) • root(tc) } ⇒ parent(p) = root(p.1)) ∧

(p.2 ∉ { tc : ⋃ (ran p.1) • root(tc) } ⇒ ∃ tc : ⋃ (ran p.1) |

tc contains p.2 • parent(p) = parent(tc,p.2))

dom subtree = { p : tree↘1↖ X × X | p.1 contains p.2 }

∀ p : dom subtree • (p.2 = root(p.1) ⇒ subtree(p) = p.1) ∧

(p.2 ≠ root(p.1) ⇒ ∃ tc : ⋃ (ran p.1) | tc contains p.2 •

subtree(p) = subtree(tc,p.2))

dom consTree = { p : X × □□tree↘1↖ X | (∀ t : p.2 • ¬ t contains p.1) ∧ (∀ t1 ,

t2 : p.2 • nodes(t1) ∩ nodes(t2) = ∅) }

∀ p : dom consTree • consTree(p) = { (p.1,p.2) }

∀ t : tree X • (t = ∅ ⇒ size(t) = 0) ∧

(t ≠ ∅ ⇒ size(t) = 1 + ∑ { tc : ⋃ (ran t) • tc ↦ size(tc) })

∟

La seguente è una funzione *parzialmente specificata* che collega una label generica ad una relazione d'ordine.

La relazione d'ordine, per essere tale, deve essere riflessiva, antisimmetrica e transitiva.

Si dice che a *precede* b se (a,b) appartiene alla relazione d'ordine.

¦═[L , X]

ℂ : L ⟶ X ↔ X

|

∀ r : ran ℂ • ∀ a : dom r ; b , c : X • (a,a) ∈ r ∧

((a,b) ∈ r ∧ (b,a) ∈ r ⇒ a = b) ∧

((a,b) ∈ r ∧ (b,c) ∈ r ⇒ (a,c) ∈ r)

∟

La funzione *childs*, dato un albero, un nodo ed un ordinamento, restituisce la lista dei sotto-alberi figli di un dato nodo, ordinata in senso crescente in base all'ordinamento

fornito in ingresso.

La funzione *preorder*, dato un albero ed un ordinamento, restituisce l'unica sequenza dei nodi dell'albero corrispondente ad una visita in preordine dell'albero stesso (ossia prima la radice, poi i sotto-alberi figli in ordine in base all'ordinamento fornito).

La funzione *postorder*, dato un albero ed un ordinamento, restituisce l'unica sequenza dei nodi dell'albero corrispondente ad una visita in postordine dell'albero stesso (ossia prima i sotto-alberi figli in ordine in base all'ordinamento fornito, poi la radice).

¡═[L , X]

childs : tree↘1↖ X × X × L ⇸ iseq tree↘1↖ X

preorder : tree X × L ⤖ iseq X

postorder : tree X × L ⤖ iseq X

|

dom childs = { p : tree↘1↖ X × X × L | p.1 contains p.2 ∧

∀ c1 , c2 : {t : ⋃ (ran(subtree(p.1,p.2))) • root(t)} • (c1,c2) ∈ ℂ(p.3) ∨ (c2,c1) ∈ ℂ(p.3) }

∀ p : dom childs • childs(p) ∈ { s : iseq tree↘1↖ X | ran s = ⋃ (ran(subtree(p.1,p.2))) ∧ ∀ i : 1 .. #s−1 • root(s(i)) ↦ root(s(i+1)) ∈ ℂ(p.3) }

dom preorder = dom postorder = { p : tree X × L | ∀ n : nodes(p.1) • (p.1,n,p.2) ∈ dom childs }

∀ p : dom preorder • (p.1 = ∅ ⇒ preorder(p) = ⟨ ⟩) ∧

(p.1 ≠ ∅ ⇒ ∃ st : iseq tree↘1↖ X ; s : seq iseq X

| st = childs(p.1,root(p.1),p.2) ∧

#st = #s ∧

∀ i : 1 .. #st • s(i) = preorder(st(i),p.2)

• preorder(p) = ⟨root(p.1)⟩ ⌢ ⌢/s)

```
  ∀ p : dom postorder • ( p.1 = ∅ ⇒ postorder(p) = ⟨ ⟩ ) ∧
                        ( p.1 ≠ ∅ ⇒ ∃ st : iseq tree↘1↖ X ; s : seq iseq X
                                    | st = childs(p.1,root(p.1),p.2) ∧
                                      #st = #s ∧
                                      ∀ i : 1 .. #st • s(i) = postorder(st(i),p.2)
                                    • postorder(p) = ⌢/s ⌢ ⟨root(p.1)⟩ )
```

∟

Un albero binario di ricerca (*bst*) è un albero in cui ogni nodo ha al più due figli e a cui è associata una relazione d'ordine totale sul dominio dei suoi elementi, tale per cui tutti i nodi minori o uguali della radice si trovano nel sotto-albero sinistro e tutti i nodi maggiori della radice sono nel sotto-albero destro.

— generic (bst _ ◀ _)

∟

—

```
  bst X ◀ L == { bt : tree X × L
                 | #(⋃ (ran bt.1)) ≤ 2 ∧
                   ( ∀ a , b : X • (a,b) ∈ ℂ(bt.2) ∨ (b,a) ∈ ℂ(bt.2) ) ∧
                   ( #(⋃ (ran bt.1)) = 1 ⇒ (∀ n :
nodes( head(childs(bt.1,root(bt.1),bt.2)) ) • (n,root(bt.1)) ∈ ℂ(bt.2)) ∨
                                           (∀ n :
nodes( head(childs(bt.1,root(bt.1),bt.2)) ) • (root(bt.1),n) ∈ ℂ(bt.2)) ) ∧
                   ( #(⋃ (ran bt.1)) = 2 ⇒ (∀ n :
nodes( childs(bt.1,root(bt.1),bt.2)(1) ) • (n,root(bt.1)) ∈ ℂ(bt.2)) ∧
                                           (∀ n :
nodes( childs(bt.1,root(bt.1),bt.2)(2) ) • (root(bt.1),n) ∈ ℂ(bt.2)) ) ∧
                   ∀ tc : ⋃ (ran bt.1) • (tc,bt.2) ∈ bst X ◀ L }
```

∟

L'abbreviazione seguente denota gli alberi binari di ricerca non vuoti.

```
—
  bst↘1↖ X ◀↘1↖ L == { bt : bst X ◀ L | bt.1 ≠ Ø }
└
```

La funzione *left*, dato un bst non vuoto restituisce il suo figlio sinistro (eventualmente vuoto).
La funzione *right*, dato un bst non vuoto, restituisce il suo figlio destro (eventualmente vuoto).
La funzione *inorder*, dato un bst, restituisce la sequenza di nodi dell'albero corrispondente ad una visita in ordine dell'albero (ossia prima il figlio sinistro, poi la radice, poi il figlio destro).
La funzione *consBst,* dato un elemento, un bst sinistro e un bst destro compatibili, costruisce un nuovo bst avente l'elemento fornito come radice.
La relazione containsBst assume valore vero se il suo argomento sinistro (un bst) contiene il suo argomento destro (un nodo). La presenza o meno del nodo nell'albero viene stabilito tramite il metodo della *ricerca binaria*.
La funzione *maximum*, dato un bst non vuoto, restituisce il suo nodo più grande in base all'ordinamento associato all'albero.
La funzione *minimum*, dato un bst non vuoto, restituisce il suo nodo più piccolo in base all'ordinamento associato all'albero.
La funzione *add*, dato un albero e un nodo non presente nell'albero, inserisce ordinatamente il nodo nel bst.
La funzione *remove*, dato un albero e un nodo, elimina dal bst il nodo fornito (se presente), preservando l'ordinamento della struttura.

```
— relation ( _ containsBst _ )
└
¦═[ L , X ]
  left : bst↘1↖ X ◀↘1↖ L ⟶ bst X ◀ L
  right : bst↘1↖ X ◀↘1↖ L ⟶ bst X ◀ L
  inorder : bst X ◀ L ⟶ iseq X
  consBst : X × bst X ◀ L × bst X ◀ L ⇸ bst↘1↖ X ◀↘1↖ L
```

```
  _ containsBst _ : bst↘1↖ X ◀↘1↖ L ↔ X
  maximum : bst↘1↖ X ◀↘1↖ L ⟶ X
  minimum : bst↘1↖ X ◀↘1↖ L ⟶ X
  add : bst X ◀ L × X ⇸ bst↘1↖ X ◀↘1↖ L
  remove : bst X ◀ L × X ⟶ bst X ◀ L
|
  ∀ t : bst↘1↖ X ◀↘1↖ L
  • let cs == childs(t.1,root(t.1),t.2)
    • [| cs = ⟨ ⟩ ∨ (cs ≠ ⟨ ⟩ ∧ (root(t.1),root(head cs)) ∈ 𝕆(t.2)) ⇒ left(t) = ∅ ↦
t.2
         cs ≠ ⟨ ⟩ ∧ (root(head cs),root(t.1)) ∈ 𝕆(t.2) ⇒ left(t) = head cs ↦ t.2

         cs = ⟨ ⟩ ∨ (#cs = 1 ∧ (root(head cs),root(t.1)) ∈ 𝕆(t.2)) ⇒ right(t) = ∅
↦ t.2
         #cs = 1 ∧ (root(t.1),root(head cs)) ∈ 𝕆(t.2) ⇒ right(t) = head cs ↦ t.2
         #cs = 2 ⇒ right(t) = cs(2) ↦ t.2 ]

  ∀ t : bst X ◀ L • ( t.1 = ∅ ⇒ inorder(t) = ⟨ ⟩ ) ∧
                    ( t.1 ≠ ∅ ⇒ inorder(t) = inorder(left(t)) ⌢ ⟨root(t.1)⟩ ⌢
inorder(right(t)) )

  dom consBst = { p : X × bst X ◀ L × bst X ◀ L
                  | (p.1,{p.2.1,p.3.1} ⩤ {∅}) ∈ dom consTree ∧
                    p.2.2 = p.3.2 ∧
                    (∀ n : nodes(p.2.1) • (n,p.1) ∈ 𝕆(p.2.2)) ∧
                    (∀ n : nodes(p.3.1) • (p.1,n) ∈ 𝕆(p.3.2)) }
  ∀ p : dom consBst • consBst(p) = consTree(p.1,{p.2.1,p.3.1} ⩤ {∅}) ↦ p.2.2
```

∀ t : bst↘1↖ X ◀↘1↖ L ; n : X • t containsBst n ⇔ n = root(t.1) ∨
((n,root(t.1)) ∈ ℂ(t.2) ∧
left(t) containsBst n) ∨
((root(t.1),n) ∈ ℂ(t.2) ∧
right(t) containsBst n)

∀ t : bst↘1↖ X ◀↘1↖ L • (right(t) = ∅ ↦ t.2 ⇒ maximum(t) = root(t.1)) ∧
(right(t) ≠ ∅ ↦ t.2 ⇒ maximum(t) =
maximum(right(t)))

∀ t : bst↘1↖ X ◀↘1↖ L • (left(t) = ∅ ↦ t.2 ⇒ minimum(t) = root(t.1)) ∧
(left(t) ≠ ∅ ↦ t.2 ⇒ minimum(t) = minimum(left(t)))

dom add = { p : bst X ◀ L × X | p.1.1 ≠ ∅ ⇒ ¬ p.1 containsBst p.2 }
∀ p : dom add • (p.1.1 = ∅ ⇒ add(p) = consBst(p.2 , ∅ ↦ p.1.2 , ∅ ↦
p.1.2)) ∧
(p.1.1 ≠ ∅ ∧ (p.2,root(p.1.1)) ∈ ℂ(p.1.2) ⇒ add(p) =
consBst(root(p.1.1) , add(left(p.1),p.2) , right(p.1))) ∧
(p.1.1 ≠ ∅ ∧ (root(p.1.1),p.2) ∈ ℂ(p.1.2) ⇒ add(p) =
consBst(root(p.1.1) , left(p.1) , add(right(p.1),p.2)))

∀ t : bst X ◀ L ; n : X • (t.1 = ∅ ∨ (t.1 ≠ ∅ ∧ n = root(t.1) ∧ #(⋃ (ran t.1)) =
0) ⇒ remove(t,n) = ∅ ↦ t.2) ∧
(t.1 ≠ ∅ ∧ n = root(t.1) ∧ #(⋃ (ran t.1)) = 1 ⇒
remove(t,n) = head(childs(t.1,n,t.2)) ↦ t.2) ∧
(t.1 ≠ ∅ ∧ n = root(t.1) ∧ #(⋃ (ran t.1)) = 2 ⇒
remove(t,n) =
consBst(minimum(right(t)),left(t),remove(right(t),minimum(right(t))))) ∧

(t.1 ≠ Ø ∧ n ≠ root(t.1) ∧ (n,root(t.1)) ∈ ℂ(t.2) ⇒

remove(t,n) = consBst(root(t.1) , remove(left(t),n) , right(t))) ∧

(t.1 ≠ Ø ∧ n ≠ root(t.1) ∧ (root(t.1),n) ∈ ℂ(t.2) ⇒

remove(t,n) = consBst(root(t.1) , left(t) , remove(right(t),n)))

∟

5.3 Introduzione del concetto di Interfaccia

Nella prassi della progettazione dei sistemi software e nell'uso dei pattern di progettazione, per esprimere la struttura concreta di un componente si fa ampio uso del concetto di *interfaccia*. In questa sede per interfaccia si intende un contenitore di eventi operanti su un certo stato astratto. Un'interfaccia rappresenta un “contratto” fra un componente software e il suo utilizzatore finale, definendo il comportamento atteso del componente a seguito del verificarsi degli eventi messi a disposizione, senza tuttavia fornire dettagli circa il funzionamento interno del componente. Questo consente fra le altre cose di modificare in un secondo momento la realizzazione di una interfaccia, senza che gli utilizzatori ne risentano. Per tali motivi una interfaccia rappresenta un importante riferimento logico ed è spesso utilizzata come identificatore di tipo all'interno di un progetto. In questo paragrafo si vedrà come introdurre tale concetto nella notazione Concepts Z.

Solitamente si dice che una interfaccia ne estende un'altra quando aggiunge componenti allo stato astratto di riferimento per l'interfaccia, aggiunge nuovi eventi o definisce specializzazioni degli eventi, nel rispetto dei vincoli dell'altra. La relazione di estensione fra interfacce è simile alla relazione di ereditarietà fra classi e può essere portata a termine correttamente seguendo le regole esposte nel paragrafo 4.5, *tenendo però a mente che una interfaccia è assimilabile in tale contesto a una classe concreta*. Si noti che la relazione di ereditarietà fra classi e quella di estensione fra interfacce sono concetti simili ma distinti. In particolare poiché una interfaccia non contiene informazioni circa la sua realizzazione concreta, è possibile per essa estendere più interfacce contemporaneamente, al contrario come detto una classe eredita da una sola superclasse. L'estensione fra interfacce risulta dunque essere una relazione, mentre

l'ereditarietà fra classi è una funzione.
Lo stato associato ad un'interfaccia, su cui gli eventi in essa definiti operano, rappresenta uno “stato astratto” assumibile dall'esterno, ma non necessariamente presente nella forma indicata nel componente che realizza l'interfaccia.
Una classe può implementare una interfaccia, rappresentando in tal modo un *raffinamento* di essa, nel senso classico del termine (vedi paragrafo 3.1). Affinché ciò avvenga, la classe deve soddisfare il contratto imposto dall'interfaccia che implementa. In quanto mera realizzazione di contratti, una classe può implementare più interfacce contemporaneamente, dunque anche l'implementazione fra una classe e una interfaccia risulta essere una relazione.

— section DesignConcepts parents AnalysisConcepts , TreeToolkit
└

Si inizia definendo l'universo di tutte le possibili interfacce.
— [INTERFACE] └

Una interfaccia può estendere una o più interfacce ma la relazione di ereditarietà è comunque aciclica.
L'estensione di un'interfaccia è rappresentata da tutte le istanze di classi che implementano quell'interfaccia. Dunque se due interfacce sono in relazione di ereditarietà, l'estensione della sottointerfaccia è contenuta nell'estensione della superinterfaccia.
Come si è detto una classe può implementare una o più interfacce.
Le istanze dirette di una interfaccia sono le istanze di classi che implementano direttamente quell'interfaccia e non una sua sottointerfaccia.

Si adotterà la convenzione di premettere ai nomi delle interfacce e dei relativi riferimenti una “i” minuscola. Le classi che implementeranno una interfaccia avranno tipicamente un nome pari al nome dell'interfaccia (privato della “i”) concatenato con il suffisso “Impl”.
— relation (_ extends _)
└
— relation (_ implements _)

└

|

_ extends _ : INTERFACE ↔ INTERFACE

_ implements _ : CLASS ↔ INTERFACE

iExtension : INTERFACE ↔ INSTANCE

iDirect : INTERFACE ↔ INSTANCE

|

(_ extends _)↗+↙ ∩ id INTERFACE = ∅

∀ iSub , iSuper : INTERFACE | iSub extends iSuper • iExtension(|{iSub}|) ⊆ iExtension(|{iSuper}|)

iDirect = iExtension ⧵ ((_ extends _)~ ⨾ iExtension)

∀ c : CLASS ; i : INTERFACE | c implements i • extension(|{c}|) ⊆ iExtension(|{i}|)

└

Con tali definizioni è possibile scrivere dichiarazioni come le seguenti:

|

iPerson : INTERFACE

iPERSON : ℙ INSTANCE

|

iExtension(|{iPerson}|) = iPERSON

└

|

iStudent : INTERFACE

iSTUDENT : ℙ iPERSON

|

iStudent extends iPerson

iExtension(|{iStudent}|) = iSTUDENT

⌙

```
|
  StudentImpl : CLASS
  STUDENT_IMPL : ℙ INSTANCE
|
  isA(StudentImpl) = Class
  extension⦇{StudentImpl}⦈ = STUDENT_IMPL
  ¬ abstract StudentImpl
  StudentImpl implements iStudent
⌙
```

Si noti che una interfaccia introdotta dal progettista può corrispondere ad un blocco individuato dall'analista e rappresenta la sua “facciata” concreta. *In tale sede è possibile per il progettista modificare le signature degli eventi predisposti dall'analista* per ragioni progettuali, ma sempre nel rispetto dei vincoli imposti dall'analista sulla semantica del blocco. A rigor di logica in casi come questi occorrerebbe dimostrare che l'interfaccia raffina correttamente il blocco definito dall'analista. Tuttavia questi temi sono più inerenti la verifica di compatibilità fra modelli, che la trasformazione di un modello in una forma più articolata per giungere alla sua realizzazione pratica. Poiché quest'ultimo è l'obiettivo di questa tesi, ci si limita a fornire alcune osservazioni pratiche in tal senso.
Quando invece definita una interfaccia, si passa a specificare una classe che la realizza, le signature degli eventi presenti nell'interfaccia non possono più essere cambiate. La classe può tuttavia adottare una rappresentazione di stato differente da quella mostrata nell'interfaccia che implementa, per convenienze realizzative. Questo purché siano rispettate le relazioni ingresso uscita affermate nella definizione degli eventi dell'interfaccia e lo stato concreto della classe evolva consistentemente con quanto avverrebbe allo stato astratto dell'interfaccia, a seguito delle sue realizzazioni degli eventi.

5.3.1 Un esempio di interfaccia

Riprendendo l'esempio dei paragrafi 3.2.1 e 4.4.2, si supponga di voler definire un'interfaccia corrispondente alla classe Sample definita in precedenza dall'analista.

— section DomainModelDesign parents DesignConcepts

⌙

|

iSample : INTERFACE

iSAMPLE : ℙ INSTANCE

|

iExtension(|{iSample}|) = iSAMPLE

⌙

. . .

— section iSampleIntensional parents DomainModelDesign , ClassIntensional

⌙

Qui viene definita una label ℕiSample per indicare un ordinamento su istanze di iSample.

|

ℕiSample : ORDER

⌙

Si noti come una interfaccia riutilizzi gli schema definiti per Class. In tal senso una interfaccia è “implementata” da una classe nella notazione.

```
┌─ łiSample
 łClass
 value : ℝ
 next : iSAMPLE
|
 this ∈ iSAMPLE
 value ≥ 0
└
```

```
┌─ PrełiSampleGetValue
 łiSample
└
┌─ PostłiSampleGetValue
 ΞłiSample
 PrełiSampleGetValue
 value! : ℝ
└
```

```
┌─ PrełiSampleGetNext
 łiSample
└
┌─ PostłiSampleGetNext
 ΞłiSample
 PrełiSampleGetNext
 next! : iSAMPLE
└
```

```
┌─ PrełiSampleSetNext
 łiSample
 n? : iSAMPLE
└
┌─ PostłiSampleSetNext
 ΔłiSample
 PrełiSampleSetNext
└
```

Come è possibile notare, nella specifica di una interfaccia non sono presenti costruttori e finalizzatori. Questo poiché una interfaccia rappresenta esclusivamente un contratto tra un utilizzatore e un implementatore. Quest'ultimo dovrà fornire dettagli circa la particolare rappresentazione di stato con cui realizza l'interfaccia. Vedremo in seguito come sia possibile imporre vincoli sulla creazione di istanze di una certa interfaccia.

```
— section iSampleRelational parents iSampleIntensional , ClassRelational
└
```

. . .

```
┌─ ℝiSample
 ℝClass
 łiSample
 ΩiSample
|
 next ∈ dom ŁiSample
└
```

In questo schema viene definito un ordinamento sulle istanze di iSample e viene collegato

tale ordinamento alla label definita nella vista intensionale dell'interfaccia.

Si assume la convenzione di indicare gli schema che definiscono ordinamenti su una entità con ℚ seguito dal nome della entità a cui si applicano gli ordinamenti.

L'ordinamento ℕiSample afferma che un'istanza di iSample precede un'altra istanza dello stesso tipo se il valore del suo campo value è minore o uguale di quello dell'altra. L'utilizzo della link function dell'interfaccia per accedere ai valori di stato delle istanze implica la presenza di eventi di query (ℸ), che restituiscano tali componenti di stato, di nome Get + nome della componente di stato. Nel nostro caso, una tale specifica di ordinamento, implica l'invocazione dell'evento di query GetValue contenuto nell'interfaccia, al fine di recuperare i valori desiderati.

Si noti infine come tale schema aggiunga dei vincoli sulla funzione ℚ (parzialmente specificata) definita nel TreeToolkit.

```
┌─ ℚiSample
  ΩiSample
  iSampleCompare : iSAMPLE ↔ iSAMPLE
|
  ∀ s1 , s2 : dom ŁiSample • (s1,s2) ∈ iSampleCompare ⇔ (ŁiSample s1).value
≤ (ŁiSample s2).value
  (ℕiSample,iSampleCompare) ∈ ℚ
└─
```

```
— section iSampleExtensional parents iSampleRelational , ClassExtensional
└─
```

Questo schema è generabile automaticamente, viene qui mostrato solo per far notare l'inclusione dello schema ℚiSample che definisce gli ordinamenti fra le variabili dell'estensione.

```
┌─ ÆiSample
  ΩiSample
```

```
 ℗iSample
 iSamples : ℙ iSample
|
 ∀ s1 , s2 : iSamples • s1.this = s2.this ⇔ s1 = s2
 ŁiSample = { s : iSamples • s.this ↦ s }
└
```

. . .

```
┌ ÐiSample
 ÐClass
 ÆiSample
|
 iSampleIsClass
└
```

Come si è detto, la relazione B extends A fra due *interfacce* è simile al concetto di ereditarietà. Dunque per la correttezza formale del modello, è possibile utilizzare l'usuale schema BIsA, già mostrato per le classi e generabile automaticamente, all'interno del dependency schema dell'interfaccia.

. . .

```
┌ PreiSampleGetValue
 PreŁiSampleGetValue
 ÐiSample
|
 ℝiSample
└
┌ PostiSampleGetValue
 PostŁiSampleGetValue
```

Ξ ĐiSample

|

Δ ℝiSample

value! = value

└

—

iSampleGetValue == ∃ Ξ łiSample • ƒiSample ∧ PreiSampleGetValue ∧ PostiSampleGetValue

└

┌ PreiSampleGetNext

PrełiSampleGetNext

ĐiSample

|

ℝiSample

└

┌ PostiSampleGetNext

PostłiSampleGetNext

Ξ ĐiSample

|

Δ ℝiSample

next! = next

└

—

iSampleGetNext == ∃ Ξ łiSample • ƒiSample ∧ PreiSampleGetNext ∧ PostiSampleGetNext

└

```
┌─ PreiSampleSetNext
  PrełiSampleSetNext
  ÐiSample
|
  ℝiSample
  n? ∈ dom ŁiSample
└─
┌─ PostiSampleSetNext
  PostłiSampleSetNext
  ΔÐiSample
|
  ΔℝiSample
  this′ = this
  value′ = value
  next′ = n?
└─
──
  iSampleSetNext == ∃ ΔłiSample • ƒiSample ∧ PreiSampleSetNext ∧
PostiSampleSetNext
└─
```

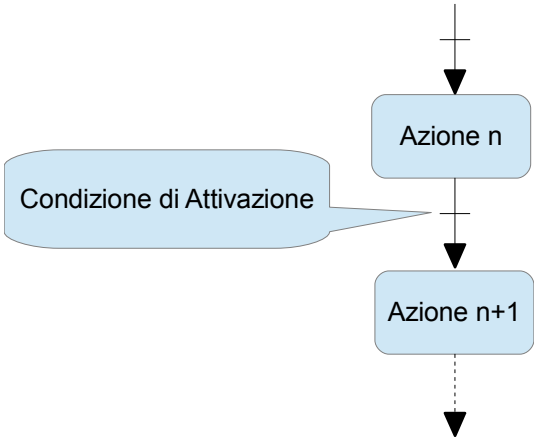
5.4 La realizzazione degli eventi

Definita una interfaccia e stabilita la rappresentazione di stato di una classe che la implementa, a partire dalla specifica degli eventi dell'interfaccia e dalla rappresentazione di stato adottata dalla classe, seguono logicamente le specifiche “concrete” degli eventi che avvengono alla classe, corrispondenti agli eventi contenuti nell'interfaccia. Tali eventi tuttavia, sono semplicemente “nuove versioni” degli eventi espressi nell'interfaccia implementata, in base alla nuova rappresentazione per lo stato astratto di essa. Dunque

non è ancora presente informazione circa la modalità con cui sia possibile passare dalle precondizioni espresse in un evento di una classe, alle sue postcondizioni. Il progettista è quindi chiamato ad individuare la sequenza di passi più conveniente che consenta di controllare se le precondizioni sono verificate ed in caso positivo, di giungere allo stato finale desiderato. Questo rende completo un modello di progetto e consente la successiva generazione automatica di codice.
Essendo Z un linguaggio di specifica, non offre supporti diretti per esprimere una sequenza di istruzioni. In esso, così come nella nostra notazione, si individua una relazione fra uno stato iniziale prima del verificarsi di un evento, ed uno stato finale dopo il verificarsi di esso. E' dunque accettabile una qualunque sequenza di passi che rispetti quanto specificato. Tuttavia, il progressivo processo di raffinamento di un modello tramite l'introduzione di vincoli ulteriori negli elementi realizzativi (classi), può portare a rendere unica la soluzione possibile. A tal fine introduciamo il concetto di *azione* nella nostra notazione.

Si definisce *azione* una *relazione* fra un insieme di variabili di ingresso ed uno stato iniziale ed un'insieme di variabili d'uscita ed uno stato finale.
Una azione rappresenta una computazione nella notazione Concepts Z.
Così come una relazione può essere definita tramite composizione o ricorsione, anche *una azione può essere definita come composizione di altre azioni o in modo ricorsivo*.
Una sequenza di azioni costituisce una implementazione di un evento.
Una azione è dotata di una *condizione di attivazione*, un predicato sulle variabili di ingresso e sullo stato iniziale, che afferma, in base al suo valore di verità, se l'azione può avvenire o meno. Una azione può produrre un *effetto* sul sistema, ossia un cambiamento dei valori di alcune delle sue componenti di stato o delle variabili di uscita.

Per rappresentare una azione nella notazione verrà utilizzato il costrutto schema. In tal modo è possibile sfruttare lo schema calculus di Z per esprimere azioni composte.
L'idea di base è quella di utilizzare le precondizioni espresse in uno schema per rappresentare la condizione di attivazione di una azione e le postcondizioni dello schema per rappresentare l'effetto di una azione.

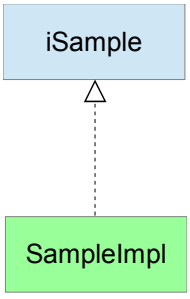


L'utilizzo degli schema consente inoltre di attribuire un nome unico ad una azione generica, che dunque potrà essere riutilizzata dalle implementazioni di diversi eventi della classe.

La parte predicativa di uno schema, nel linguaggio Z, non implica una sequenzialità per i predicati in congiunzione in essa definiti. Tuttavia nulla vieta ad uno strumento che elabori la specifica di progetto (ad esempio un generatore di codice), di interpretare in modo sequenziale i congiunti, in base all'ordine con cui siano stati scritti dal progettista.
In questa fase dunque, *l'ordine con cui vengono asserite le proposizioni* all'interno della notazione, *relativamente agli schema che esprimono le implementazioni, diviene rilevante*, poiché sarà una discriminante per la successiva generazione di codice.

5.4.1 Un esempio di classe implementativa

Riprendendo l'esempio del paragrafo 5.3.1, si supponga di voler dare una implementazione dell'interfaccia iSample.



```
— section DomainModelDesign parents DesignConcepts
└
        . . .
|
  SampleImpl : CLASS
  SAMPLE_IMPL : ℙ INSTANCE
|
  isA( SampleImpl ) = Class
  extension⁅{SampleImpl}⁆ = SAMPLE_IMPL
  ¬ abstract SampleImpl
  SampleImpl implements iSample
└


— section SampleImplIntensional parents iSampleIntensional
└

┌─ łSampleImpl
  łClass
  value : ℝ
  next : iSAMPLE
|
  this ∈ SAMPLE_IMPL
  value ≥ 0
└

┌─ łSampleImplInit
  łClassInit
  łSampleImpl ’
```

value? : ℝ

└

┌─ łSampleImplFin

łClassFin

łSampleImpl

└

┌─ PrełSampleImplGetValue

łSampleImpl

└

┌─ PostłSampleImplGetValue

Ξ łSampleImpl

PrełSampleImplGetValue

value! : ℝ

└

┌─ PrełSampleImplGetNext

łSampleImpl

└

┌─ PostłSampleImplGetNext

Ξ łSampleImpl

PrełSampleImplGetNext

next! : iSAMPLE

└

┌─ PrełSampleImplSetNext

łSampleImpl

```
  n? : iSAMPLE
└
┌─ PostłSampleImplSetNext
  ΔłSampleImpl
  PrełSampleImplSetNext
└
```

— section SampleImplRelational parents SampleImplIntensional , iSampleRelational

```
└
```

. . .

```
┌─ ℝSampleImpl
  ℝClass
  łSampleImpl
  ΩiSample
  ΩSampleImpl
|
  next ∈ dom ŁiSample
└
```

— section SampleImplExtensional parents SampleImplRelational , iSampleExtensional

```
└
```

. . .

Lo schema seguente è generabile automaticamente ed è necessario per la correttezza formale del modello. È stato qui mostrato solo per sottolineare che essendo l'estensione di

un'interfaccia l'insieme di tutte le istanze di classi che la implementano, occorre asserire nel dependency schema di una classe che implementa un'interfaccia, che il suo insieme di istanze attualmente presenti nel sistema, è contenuto nelle istanze attualmente presenti nel sistema dell'interfaccia implementata.

```
┌─ SampleImplImplementsiSample
 ÆSampleImpl
 ÆiSample
|
 {s : sampleImpls • s.this} ⊆ {c : iSamples • c.this}
└─
```

```
┌─ ÐSampleImpl
 ÐiSample
 ÆSampleImpl
|
 SampleImplIsClass
 SampleImplImplementsiSample
└─
```

. . .

```
┌─ SampleImplInit
 łSampleImplInit
 ΔÐSampleImpl
|
 ℝSampleImpl ’
 ClassInit
 this? ∈ SAMPLE_IMPL
 value? ≥ 0
 value’ = value?
```

next′ = this?

└

—

SampleImplNew == ∃ łSampleImpl ′ • ƒSampleImplN ∧ SampleImplInit[os!/this?]

└

┌─ SampleImplNewImpl

SampleImplInit

|

□§SampleImplNewImpl_1

└

┌─ □§SampleImplNewImpl_1_1

łSampleImpl ′

ΔÐSampleImpl

this? : INSTANCE

value? : ℝ

|

this? ∈ SAMPLE_IMPL

value? ≥ 0

ClassInitImpl

└

┌─ □§SampleImplNewImpl_1_2

łSampleImpl ′

this? : INSTANCE

value? : ℝ

|

value′ = value?

next′ = this?

└

—

□§SampleImplNewImpl_1 == □§SampleImplNewImpl_1_1 ∧

□§SampleImplNewImpl_1_2

└

Per convenzione, l'implementazione di un evento viene espressa all'interno di uno schema avente il nome dell'evento seguito dal suffisso “Impl”.

Tale schema contiene la sequenza di azioni scelta dal progettista per realizzare l'evento.

Una azione è rappresentata da uno schema il cui nome inizia con il simbolo □□.

Qui si specifica la realizzazione del costruttore della classe SampleImpl attraverso due fasi. In particolare si afferma che l'implementazione consiste in un'unica azione, a sua volta composta di due azioni. La prima controlla le precondizioni dell'evento e richiama l'implementazione dell'evento della superclasse, la seconda inizializza i valori di stato specifici di un'istanza di SampleImpl.

┌─ SampleImplFin

ŁSampleImplFin

ΔÐSampleImpl

|

ℝSampleImpl

ClassFin

└

—

SampleImplDelete == ∃ ŁSampleImpl • ƒSampleImplD ∧ SampleImplFin

└

┌─ SampleImplDeleteImpl

SampleImplFin

|

□§SampleImplDeleteImpl_1

└

```
┌─ □§SampleImplDeleteImpl_1
 łSampleImpl
 ΔÐSampleImpl
|
 ClassFinImpl
└
```

```
┌─ PreSampleImplGetValue
 PrełSampleImplGetValue
 ÐSampleImpl
|
 ℝSampleImpl
└
┌─ PostSampleImplGetValue
 PostłSampleImplGetValue
 ΞÐSampleImpl
|
 ΔℝSampleImpl
 value! = value
└
```

```
—
 SampleImplGetValueSpec == PreSampleImplGetValue ∧
PostSampleImplGetValue
└
```

```
—
 SampleImplGetValue == ∃ ΞłSampleImpl • ƒSampleImpl ∧
SampleImplGetValueSpec
└
```

┌─ SampleImplGetValueImpl

SampleImplGetValueSpec

|

SampleImplGetValueImpl_1

└─

┌─ SampleImplGetValueImpl_1

SampleImpl

value! : ℝ

|

value! = value

└─

┌─ PreSampleImplGetNext

PreSampleImplGetNext

SampleImpl

|

ℝSampleImpl

└─

┌─ PostSampleImplGetNext

PostSampleImplGetNext

SampleImpl

|

ΔℝSampleImpl

next! = next

└─

─

SampleImplGetNextSpec == PreSampleImplGetNext ∧ PostSampleImplGetNext

└

—

SampleImplGetNext == ∃ Ξ łSampleImpl • ƒSampleImpl ∧
SampleImplGetNextSpec

└

┌─ SampleImplGetNextImpl

SampleImplGetNextSpec

|

SampleImplGetNextImpl_1

└

┌─ SampleImplGetNextImpl_1

Ξ łSampleImpl

next! : iSAMPLE

|

next! = next

└

┌─ PreSampleImplSetNext

PrełSampleImplSetNext

ĐSampleImpl

|

ℝSampleImpl

n? ∈ dom ŁiSample

└

┌─ PostSampleImplSetNext

PostłSampleImplSetNext

ΔĐSampleImpl

|

```
  ΔℝSampleImpl
  this′ = this
  value′ = value
  next′ = n?
└
─
  SampleImplSetNextSpec == PreSampleImplSetNext ∧
PostSampleImplSetNext
└
─
  SampleImplSetNext == ∃ ΔłSampleImpl • ƒSampleImpl ∧
SampleImplSetNextSpec
└
┌─ SampleImplSetNextImpl
  SampleImplSetNextSpec
|
  □§SampleImplSetNextImpl_1
└
┌─ □§SampleImplSetNextImpl_1_1
  ΔÐSampleImpl
  n? : iSAMPLE
|
  n? ∈ dom ŁiSample
└
┌─ □§SampleImplSetNextImpl_1_2
  ΔłSampleImpl
  n? : iSAMPLE
|
```

this′ = this

value′ = value

next′ = n?

⌙

—

[]SampleImplSetNextImpl_1 == []SampleImplSetNextImpl_1_1 ∧ []SampleImplSetNextImpl_1_2

⌙

Si noti come la specifica di ogni evento sia a disposizione del progettista e rappresenti un utile ausilio per l'individuazione della sequenza di azioni che la implementa. L'idea di base per l'individuazione di una implementazione è quella di individuare le fasi necessarie a portare a compimento un evento. Si noti che un'azione può dichiarare variabili ausiliare nella sua parte predicativa e passarle alle successive attraverso l'usuale meccanismo di unificazione di variabili aventi lo stesso nome, presente in Z. Si noti infine che ogni azione specificata lavora sul dependency schema della classe, sulla sola intensione della classe, o su entrambe le cose. Tipicamente un'azione lavora sul solo dependency schema quando intende invocare altri eventi (ad esempio metodi di altre classi), lavora sulla sola intensione della classe quando manipola lo stato interno e lavora su entrambe quando richiama una implementazione dell'evento specializzato di una superclasse. Si noti che *lo stato associato alla sequenza di azioni, deve necessariamente evolvere in modo consistente*. Per raggiungere tale obiettivo si rivela utile individuare le varie fasi in funzione di queste tre tipologie di azioni. In particolare, si vedrà più avanti come le azioni che lavorano sulla sola intensione di stato, rivestano un ruolo speciale per la generazione di codice del sistema.

5.4.2 Un esempio di progettazione

Riprendendo l'esempio dei paragrafi 3.2.1, 4.4.2, 5.3.1 e 5.4.1, si vuole ora mostrare un esempio dell'uso della notazione Concepts Z in fase di progetto.

5.4.2.1 Il blocco ConsumptionEstimator

Si supponga che l'analista, oltre ad aver previsto il blocco Sample, abbia specificato un secondo blocco in grado di effettuare la stima del valore di una variabile casuale a valori reali. Tale stima viene effettuata utilizzando una finestra dinamica di campioni, attraverso il metodo statistico dei quantili [25].

```
— section DomainModelConcepts parents AnalysisConcepts
└
        . . .
|
  ConsumptionEstimator : CLASS
  CONSUMPTION_ESTIMATOR : ℙ INSTANCE
|
  isA(ConsumptionEstimator) = Class
  extension(⦇{ConsumptionEstimator}⦈) = CONSUMPTION_ESTIMATOR
  ¬ abstract ConsumptionEstimator
└


— section ConsumptionEstimatorIntensional parents DomainModelConcepts ,
ClassIntensional
└

┌ ł ConsumptionEstimator
  ł Class
  sampleWindow : iseq SAMPLE
  maxSize : ℕ↘1↖
  oldest : □□SAMPLE
```

newest : □□SAMPLE

q1 : ℝ

q2 : ℝ

q3 : ℝ

iqr : ℝ

minConsecutiveStrangeSamples : ℕ↘1↖

strangeSamples : ℕ

senseFault : □ □

|

this ∈ CONSUMPTION_ESTIMATOR

#sampleWindow ≤ maxSize

#oldest ∈ 0 .. 1

#newest ∈ 0 .. 1

oldest ⊆ ran sampleWindow

newest ⊆ ran sampleWindow

maxSize ≥ 7

strangeSamples ≤ maxSize

maxSize * 5 div 100 < minConsecutiveStrangeSamples ≤ maxSize * 15 div 100

strangeSamples ≥ minConsecutiveStrangeSamples ∧ #sampleWindow ≥ maxSize * 85 div 100 ⇔ senseFault = True

└

┌ ł ConsumptionEstimatorInit

ł ClassInit

ł ConsumptionEstimator ’

maxSize? : ℕ↘1↖

minConsecutiveStrangeSamples? : ℕ↘1↖

└

┌─ łConsumptionEstimatorFin
łClassFin
łConsumptionEstimator
└

┌─ PrełConsumptionEstimatorTrim
łConsumptionEstimator
└

┌─ PostłConsumptionEstimatorTrim
ΔłConsumptionEstimator
PrełConsumptionEstimatorTrim
└

┌─ PrełConsumptionEstimatorNewData
łConsumptionEstimator
data? : ℝ
└

┌─ PostłConsumptionEstimatorNewData
ΔłConsumptionEstimator
PrełConsumptionEstimatorNewData
q2! : ℝ
iqr! : ℝ
└

┌─ PrełConsumptionEstimatorIsSenseFault
łConsumptionEstimator

└

┌─ PostłConsumptionEstimatorIsSenseFault

≡łConsumptionEstimator

PrełConsumptionEstimatorIsSenseFault

senseFault! : □□

└

— section ConsumptionEstimatorRelational parents

ConsumptionEstimatorIntensional , SampleRelational

└

. . .

┌─ ℝConsumptionEstimator

ℝClass

łConsumptionEstimator

ΩSample

ΩConsumptionEstimator

|

∀ s : ran sampleWindow • s ∈ dom łSample

∀ i : 1 .. #sampleWindow − 1 • (łSample (sampleWindow i)).value ≤

(łSample (sampleWindow (i+1))).value

#sampleWindow > 0 ⇒ let np25 == 25 div 100 * #sampleWindow ;

np50 == 50 div 100 * #sampleWindow ;

np75 == 75 div 100 * #sampleWindow

•[| np25 ∉ ℕ ⇒ q1 = (łSample (sampleWindow

(ceiling np25))).value

np25 ∈ ℕ ⇒ q1 = ((łSample (sampleWindow

np25)).value + (ŁSample (sampleWindow (np25+1))).value) div 2

np50 ∉ ℕ ⇒ q2 = (ŁSample (sampleWindow (ceiling np50))).value

np50 ∈ ℕ ⇒ q2 = ((ŁSample (sampleWindow np50)).value + (ŁSample (sampleWindow (np50+1))).value) div 2

np75 ∉ ℕ ⇒ q3 = (ŁSample (sampleWindow (ceiling np75))).value

np75 ∈ ℕ ⇒ q3 = ((ŁSample (sampleWindow np75)).value + (ŁSample (sampleWindow (np75+1))).value) div 2

iqr = q3 − q1]

└

— section ConsumptionEstimatorExtensional parents ConsumptionEstimatorRelational , SampleExtensional

└

. . .

┌─ ÐConsumptionEstimator
ÐSample
ÆConsumptionEstimator
|
ConsumptionEstimatorIsClass
└

. . .

```
┌─ ConsumptionEstimatorInit
 ł ConsumptionEstimatorInit
 ΔĐConsumptionEstimator
|
 ℝConsumptionEstimator ’
 ClassInit
 this? ∈ CONSUMPTION_ESTIMATOR
 maxSize? ≥ 7
 maxSize? * 5 div 100 < minConsecutiveStrangeSamples? ≤ maxSize? * 15 div
100
 sampleWindow’ = ⟨ ⟩
 maxSize’ = maxSize?
 oldest’ = Ø
 newest’ = Ø
 minConsecutiveStrangeSamples’ = minConsecutiveStrangeSamples?
 strangeSamples’ = 0
 senseFault’ = False
└─
──
 ConsumptionEstimatorNew == ∃ ł ConsumptionEstimator ’ •
ƒ ConsumptionEstimatorN ∧ ConsumptionEstimatorInit[oc!/this?]
└─

┌─ ConsumptionEstimatorFin
 ł ConsumptionEstimatorFin
 ΔĐConsumptionEstimator
|
```

ℝConsumptionEstimator

ClassFin

#sampleWindow = 0

└

—

ConsumptionEstimatorDelete == ∃ łConsumptionEstimator •

ƒConsumptionEstimatorD ∧ ConsumptionEstimatorFin

└

┌ EmptySampleWindow

ΔĐSample

w? : iseq SAMPLE

|

w? = 〈 〉 ⇒ ΞĐSample

w? ≠ 〈 〉 ⇒ ∃ s : SAMPLE ; tw : iseq SAMPLE | s = head w? ∧ tw = tail w? •

SampleDelete[s/os?] ⨟ EmptySampleWindow[tw/w?]

└

┌ PreConsumptionEstimatorTrim

PrełConsumptionEstimatorTrim

ĐConsumptionEstimator

|

ℝConsumptionEstimator

└

┌ PostConsumptionEstimatorTrim

PostłConsumptionEstimatorTrim

ΔĐConsumptionEstimator

|

ΔℝConsumptionEstimator

this′ = this

sampleWindow′ = ⟨ ⟩

maxSize′ = maxSize

oldest′ = ∅

newest′ = ∅

minConsecutiveStrangeSamples′ = minConsecutiveStrangeSamples

strangeSamples′ = 0

senseFault′ = False

EmptySampleWindow[sampleWindow/w?]

└

—

ConsumptionEstimatorTrim == ∃ ΔłConsumptionEstimator •

ƒConsumptionEstimator ∧ PreConsumptionEstimatorTrim ∧

PostConsumptionEstimatorTrim

└

┌ PreConsumptionEstimatorNewDataNoSamples

PrełConsumptionEstimatorNewData

ĐConsumptionEstimator

|

ℝConsumptionEstimator

data? ≥ 0

#sampleWindow = 0

└

┌ PostConsumptionEstimatorNewDataNoSamples

PostłConsumptionEstimatorNewData

ΔĐConsumptionEstimator

|

ΔℝConsumptionEstimator

this′ = this

maxSize′ = maxSize

minConsecutiveStrangeSamples′ =minConsecutiveStrangeSamples

#sampleWindow′ ≥ maxSize * 85 div 100 ∧ senseFault = False

⇒ (data? ∈ (q2 − iqr div 2) .. (q2 + iqr div 2) ⇒ strangeSamples′ = 0) ∧

(data? ∉ (q2 − iqr div 2) .. (q2 + iqr div 2) ⇒ strangeSamples′ =

strangeSamples + 1)

#sampleWindow′ < maxSize * 85 div 100 ∨ senseFault = True ⇒

strangeSamples′ = strangeSamples

q2! = q2′

iqr! = iqr′

∃ os! : SAMPLE

• SampleNew[data?/value?] ∧

[| sampleWindow′ = ⟨ os! ⟩

oldest′ = {os!}

newest′ = {os!}]

└

┌ PreConsumptionEstimatorNewDataRoomForSamples

PrełConsumptionEstimatorNewData

ÐConsumptionEstimator

|

ℝConsumptionEstimator

data? ≥ 0

0 < #sampleWindow < maxSize

└

┌ PostConsumptionEstimatorNewDataRoomForSamples

PostłConsumptionEstimatorNewData

ΔÐConsumptionEstimator

|

ΔℝConsumptionEstimator

this′ = this

maxSize′ = maxSize

minConsecutiveStrangeSamples′ =minConsecutiveStrangeSamples

#sampleWindow′ ≥ maxSize * 85 div 100 ∧ senseFault = False

⇒ (data? ∈ (q2 − iqr div 2) .. (q2 + iqr div 2) ⇒ strangeSamples′ = 0) ∧

(data? ∉ (q2 − iqr div 2) .. (q2 + iqr div 2) ⇒ strangeSamples′ =

strangeSamples + 1)

#sampleWindow′ < maxSize * 85 div 100 ∨ senseFault = True ⇒

strangeSamples′ = strangeSamples

q2! = q2′

iqr! = iqr′

∃ os! , oldNewest : SAMPLE

| newest = {oldNewest}

• SampleNew[data?/value?] ⨟

SampleSetNext[oldNewest/os?,os!/n?] ∧

[| sampleWindow′ = sampleWindow ↾ {s : ran sampleWindow ; v : ℝ |

SampleGetValue[s/os?,v/value!] ∧ v ≤ data? • s} ⌢ ⟨ os! ⟩ ⌢

sampleWindow ↾ {s : ran sampleWindow ; v : ℝ |

SampleGetValue[s/os?,v/value!] ∧ v > data? • s}

oldest′ = oldest

newest′ = {os!}]

└

┌ PreConsumptionEstimatorNewDataNoRoomForSamples

PrełConsumptionEstimatorNewData

ÐConsumptionEstimator

|

ℝConsumptionEstimator

data? ≥ 0

#sampleWindow = maxSize

└

┌─ PostConsumptionEstimatorNewDataNoRoomForSamples

PostłConsumptionEstimatorNewData

ΔÐConsumptionEstimator

|

ΔℝConsumptionEstimator

this′ = this

maxSize′ = maxSize

minConsecutiveStrangeSamples′ =minConsecutiveStrangeSamples

#sampleWindow′ ≥ maxSize * 85 div 100 ∧ senseFault = False

⇒ (data? ∈ (q2 − iqr div 2) .. (q2 + iqr div 2) ⇒ strangeSamples′ = 0) ∧

(data? ∉ (q2 − iqr div 2) .. (q2 + iqr div 2) ⇒ strangeSamples′ =

strangeSamples + 1)

#sampleWindow′ < maxSize * 85 div 100 ∨ senseFault = True ⇒

strangeSamples′ = strangeSamples

q2! = q2′

iqr! = iqr′

∃ os! , oldOldest , oldNewest , newOldest : SAMPLE

| oldest = {oldOldest} ∧ newest = {oldNewest}

• SampleNew[data?/value?] ⨾

SampleSetNext[oldNewest/os?,os!/n?] ⨾

SampleGetNext[oldOldest/os?,newOldest/next!] ⨾

SampleDelete[oldOldest/os?] ∧

[| sampleWindow′ = sampleWindow ⨾ {s : ran sampleWindow ⧵ {oldOldest} ; v : ℝ | SampleGetValue[s/os?,v/value!] ∧ v ≤ data? • s} ⌢ ⟨ os! ⟩ ⌢ sampleWindow ⨾ {s : ran sampleWindow ⧵ {oldOldest} ; v : ℝ | SampleGetValue[s/os?,v/value!] ∧ v > data? • s}
oldest′ = {newOldest}
newest′ = {os!}]

└

—

ConsumptionEstimatorNewData ==
∃ ΔłConsumptionEstimator • ƒConsumptionEstimator ∧
((PreConsumptionEstimatorNewDataNoSamples ∧ PostConsumptionEstimatorNewDataNoSamples) ∨

(PreConsumptionEstimatorNewDataRoomForSamples ∧ PostConsumptionEstimatorNewDataRoomForSamples) ∨

(PreConsumptionEstimatorNewDataNoRoomForSamples ∧ PostConsumptionEstimatorNewDataNoRoomForSamples))
└

┌ PreConsumptionEstimatorIsSenseFault
PrełConsumptionEstimatorIsSenseFault
ĐConsumptionEstimator
|
ℝConsumptionEstimator
└
┌ PostConsumptionEstimatorIsSenseFault

PostłConsumptionEstimatorIsSenseFault

ΞÐConsumptionEstimator

|

ΔℝConsumptionEstimator

senseFault! = senseFault

⌙

—

ConsumptionEstimatorIsSenseFault == ∃ ΞłConsumptionEstimator •

ƒConsumptionEstimator ∧ PreConsumptionEstimatorIsSenseFault ∧

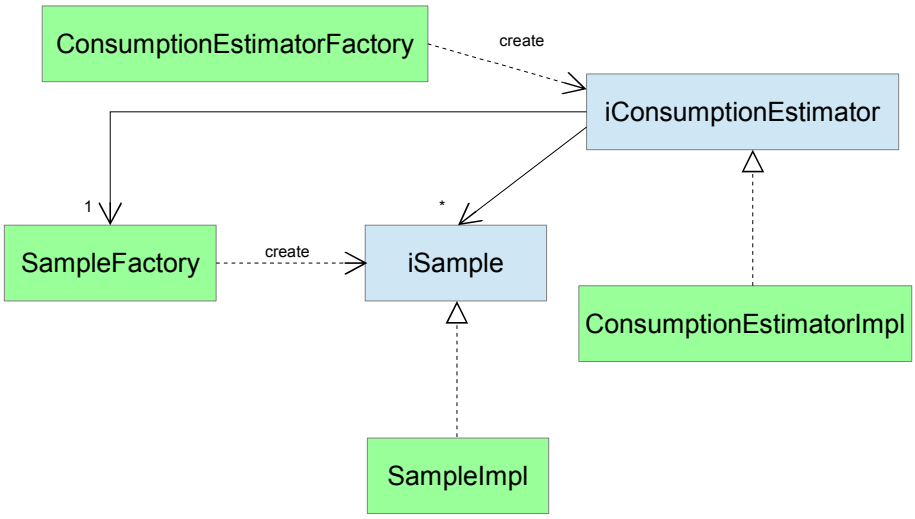
PostConsumptionEstimatorIsSenseFault

⌙

Ancora una volta si noti come si tratti di una specifica che esprime *cosa* fa il blocco senza dire *come* lo fa. Ad esempio per calcolare i valori del secondo quantile q2 e del range inter-quantile iqr, negli invarianti si afferma che la lista di campioni attualmente disponibili allo stimatore sampleWindow, è sempre ordinata. Come ciò si ottenga è lasciato non specificato e dunque rappresenta un grado di libertà per il progettista.

5.4.2.2 Uno schema di riferimento

Si supponga ora di voler specificare la seguente soluzione progettuale in cui si fa uso del *pattern factory* [11] :



Si prevedono dunque due interfacce, una per il concetto di Sample (iSample) ed una per il concetto di ConsumptionEstimator (iConsumptionEstimator). Si prevede inoltre di realizzare queste due interfacce tramite rispettivamente la classe SampleImpl e la classe ConsumptionEstimatorImpl. Infine di prevede la presenza di due factory, le quali consentono di creare istanze delle due interfacce previste, nascondendo i dettagli relativi alla loro realizzazione interna.

```
— section DomainModelDesign parents DesignConcepts
└
        . . .
|
  SampleFactory : CLASS
  SAMPLE_FACTORY : ℙ INSTANCE
|
  isA( SampleFactory ) = Class
  extension(⦇{SampleFactory}⦈) = SAMPLE_FACTORY
  ¬ abstract SampleFactory
└


|
  iConsumptionEstimator : INTERFACE
```

```
  iCONSUMPTION_ESTIMATOR : ℙ INSTANCE
|
  iExtension(⦇{iConsumptionEstimator}⦈) = iCONSUMPTION_ESTIMATOR
└─

┌
  ConsumptionEstimatorImpl : CLASS
  CONSUMPTION_ESTIMATOR_IMPL : ℙ INSTANCE
|
  isA(ConsumptionEstimatorImpl) = Class
  extension(⦇{ConsumptionEstimatorImpl}⦈) = CONSUMPTION_ESTIMATOR_IMPL
  ¬ abstract ConsumptionEstimatorImpl
  ConsumptionEstimatorImpl implements iConsumptionEstimator
└─

┌
  ConsumptionEstimatorFactory : CLASS
  CONSUMPTION_ESTIMATOR_FACTORY : ℙ INSTANCE
|
  isA( ConsumptionEstimatorFactory ) = Class
  extension(⦇{ConsumptionEstimatorFactory}⦈) =
CONSUMPTION_ESTIMATOR_FACTORY
  ¬ abstract ConsumptionEstimatorFactory
└─
```

Lo stimatore iConsumptionEstimator utilizzerà delle istanze di iSample per rappresentare i campioni in suo possesso. Poiché si è già mostrata la specifica dell'interfaccia iSample e la sua classe realizzativa SampleImpl, si partirà dalla specifica della factory dei campioni.

5.4.2.3 La classe SampleFactory

```
— section SampleFactoryIntensional parents DomainModelDesign ,
ClassIntensional
└─

┌─ łSampleFactory
 łClass
|
 this ∈ SAMPLE_FACTORY
└─

┌─ łSampleFactoryInit
 łClassInit
 łSampleFactory ’
└─

┌─ łSampleFactoryFin
 łClassFin
 łSampleFactory
└─

┌─ PrełSampleFactoryGetSample
 łSampleFactory
 value? : ℝ
└─
┌─ PostłSampleFactoryGetSample
 ΔłSampleFactory
```

```
  PrełSampleFactoryGetSample
  os! : iSAMPLE
└─
```

```
— section SampleFactoryRelational parents SampleFactoryIntensional ,
ClassRelational
└─
        . . .

┌─ ℝSampleFactory
  ℝClass
  łSampleFactory
  ΩSampleFactory
└─
```

```
— section SampleFactoryExtensional parents SampleFactoryRelational ,
SampleImplExtensional
└─
        . . .

┌─ ÐSampleFactory
  ÐSampleImpl
  ÆSampleFactory
|
  SampleFactoryIsClass
└─
        . . .
```

┌─ SampleFactoryInit
ł SampleFactoryInit
Δ Đ SampleFactory
|
ℝ SampleFactory ’
ClassInit
this? ∈ SAMPLE_FACTORY
└
—
SampleFactoryNew == ∃ ł SampleFactory ’ • ƒ SampleFactoryN ∧ SampleFactoryInit[os!/this?]
└
┌─ SampleFactoryNewImpl
SampleFactoryInit
|
□§ SampleFactoryNewImpl_1
└
┌─ □§ SampleFactoryNewImpl_1
ł SampleFactory ’
Δ Đ SampleFactory
this? :INSTANCE
|
this? ∈ SAMPLE_FACTORY
ClassInitImpl
└

┌─ SampleFactoryFin

łSampleFactoryFin

ΔÐSampleFactory

|

ℝSampleFactory

ClassFin

└

—

SampleFactoryDelete == ∃ łSampleFactory • ƒSampleFactoryD ∧ SampleFactoryFin

└

┌ SampleFactoryDeleteImpl

SampleFactoryFin

|

SampleFactoryDeleteImpl_1

└

┌ SampleFactoryDeleteImpl_1

łSampleFactory

ΔÐSampleFactory

|

ClassFinImpl

└

┌ PreSampleFactoryGetSample

PrełSampleFactoryGetSample

ÐSampleFactory

|

ℝSampleFactory

value? > 0

└

┌─ PostSampleFactoryGetSample

PostłSampleFactoryGetSample

ΔÐSampleFactory

|

ΔℝSampleFactory

this′ = this

SampleImplNew

└

—

SampleFactoryGetSampleSpec == PreSampleFactoryGetSample ∧ PostSampleFactoryGetSample

└

—

SampleFactoryGetSample == ∃ ΔłSampleFactory • ƒSampleFactory ∧ SampleFactoryGetSampleSpec

└

┌─ SampleFactoryGetSampleImpl

SampleFactoryGetSampleSpec

|

□§SampleFactoryGetSampleImpl_1

└

┌─ □§SampleFactoryGetSampleImpl_1_1

ΔÐSampleFactory

value? : ℝ

os! : iSAMPLE

|

value? > 0

```
 SampleImplNew
└
┌ □§SampleFactoryGetSampleImpl_1_2
 ΔłSampleFactory
|
 this′ = this
└
—
 □§SampleFactoryGetSampleImpl_1 == □§SampleFactoryGetSampleImpl_1_1 ∧
□§SampleFactoryGetSampleImpl_1_2
└
```

Si noti come la factory imponga la precondizione che il valore fornito in ingresso per il campione debba essere positivo. Tra le altre cose il *pattern factory* è dunque *un possibile modo per imporre vincoli sulla creazione delle istanze di una interfaccia*.

5.4.2.4 L'interfaccia iConsumptionEstimator

Si passa ora a definire l'interfaccia dello stimatore. Tale interfaccia impone l'utilizzo della factory appena definita, introducendo ed usando un riferimento ad essa come variabile di istanza.

```
— section iConsumptionEstimatorIntensional parents DomainModelDesign ,
ClassIntensional
└

┌ łiConsumptionEstimator
 łClass
 sampleWindow : iseq iSAMPLE
```

maxSize : ℕ↘1↖

oldest : □□SAMPLE

newest : □□SAMPLE

q1 : ℝ

q2 : ℝ

q3 : ℝ

iqr : ℝ

minConsecutiveStrangeSamples : ℕ↘1↖

strangeSamples : ℕ

senseFault : □ □

factory : SAMPLE_FACTORY

|

this ∈ iCONSUMPTION_ESTIMATOR

#sampleWindow ≤ maxSize

#oldest ∈ 0 .. 1

#newest ∈ 0 .. 1

oldest ⊆ ran sampleWindow

newest ⊆ ran sampleWindow

maxSize ≥ 7

strangeSamples ≤ maxSize

maxSize * 5 div 100 < minConsecutiveStrangeSamples ≤ maxSize * 15 div 100

strangeSamples ≥ minConsecutiveStrangeSamples ∧ #sampleWindow ≥ maxSize * 85 div 100 ⇔ senseFault = True

∟

┌─ PrełiConsumptionEstimatorTrim

łiConsumptionEstimator

└

┌─ PostłiConsumptionEstimatorTrim

ΔłiConsumptionEstimator

PrełiConsumptionEstimatorTrim

└

┌─ PrełiConsumptionEstimatorNewData

łiConsumptionEstimator

data? : ℝ

└

┌─ PostłiConsumptionEstimatorNewData

ΔłiConsumptionEstimator

PrełiConsumptionEstimatorNewData

q2! : ℝ

iqr! : ℝ

└

┌─ PrełiConsumptionEstimatorIsSenseFault

łiConsumptionEstimator

└

┌─ PostłiConsumptionEstimatorIsSenseFault

Ξ łiConsumptionEstimator

PrełiConsumptionEstimatorIsSenseFault

senseFault! : □□

└

— section iConsumptionEstimatorRelational parents

iConsumptionEstimatorIntensional , iSampleRelational ,

SampleFactoryRelational

└

. . .

┌─ ℝiConsumptionEstimator

ℝClass

łiConsumptionEstimator

ΩiSample

ΩSampleFactory

ΩiConsumptionEstimator

|

∀ s : ran sampleWindow • s ∈ dom ŁiSample

factory ∈ dom ŁSampleFactory

∀ i : 1 .. #sampleWindow − 1 • (ŁiSample (sampleWindow i)).value ≤ (ŁiSample (sampleWindow (i+1))).value

#sampleWindow > 0 ⇒ let np25 == 25 div 100 * #sampleWindow ;

np50 == 50 div 100 * #sampleWindow ;

np75 == 75 div 100 * #sampleWindow

•[| np25 ∉ ℕ ⇒ q1 = (ŁiSample (sampleWindow (ceiling np25))).value

np25 ∈ ℕ ⇒ q1 = ((ŁiSample (sampleWindow np25)).value + (ŁiSample (sampleWindow (np25+1))).value) div 2

np50 ∉ ℕ ⇒ q2 = (ŁiSample (sampleWindow (ceiling np50))).value

np50 ∈ ℕ ⇒ q2 = ((ŁiSample (sampleWindow np50)).value + (ŁiSample (sampleWindow (np50+1))).value) div 2

np75 ∉ ℕ ⇒ q3 = (ŁiSample (sampleWindow (ceiling np75))).value

np75 ∈ ℕ ⇒ q3 = ((ŁiSample (sampleWindow np75)).value + (ŁiSample (sampleWindow (np75+1))).value) div 2

iqr = q3 − q1]

└─

— section iConsumptionEstimatorExtensional parents iConsumptionEstimatorRelational , SampleFactoryExtensional

└─

. . .

┌─ ĐiConsumptionEstimator

ĐSampleFactory

ÆiConsumptionEstimator

|

iConsumptionEstimatorIsClass

└─

. . .

┌─ PreiConsumptionEstimatorTrim

PrełiConsumptionEstimatorTrim

ĐiConsumptionEstimator

|

ℝiConsumptionEstimator

└─

┌─ PostiConsumptionEstimatorTrim

PostłiConsumptionEstimatorTrim

ΔĐiConsumptionEstimator

|

ΔℝiConsumptionEstimator

this′ = this

sampleWindow′ = ⟨ ⟩

maxSize′ = maxSize

oldest′ = ∅

newest′ = ∅

minConsecutiveStrangeSamples′ = minConsecutiveStrangeSamples

strangeSamples′ = 0

senseFault′ = False

factory′ = factory

└

—

iConsumptionEstimatorTrim == ∃ ΔłiConsumptionEstimator •

ƒiConsumptionEstimator ∧ PreiConsumptionEstimatorTrim ∧

PostiConsumptionEstimatorTrim

└

┌─ PreiConsumptionEstimatorNewDataNoSamples

PrełiConsumptionEstimatorNewData

ĐiConsumptionEstimator

|

ℝiConsumptionEstimator

data? ≥ 0

#sampleWindow = 0

└

┌─ PostiConsumptionEstimatorNewDataNoSamples

PostłiConsumptionEstimatorNewData

ΔĐiConsumptionEstimator

|

ΔℝiConsumptionEstimator

this′ = this

maxSize′ = maxSize

minConsecutiveStrangeSamples′ =minConsecutiveStrangeSamples

#sampleWindow′ ≥ maxSize * 85 div 100 ∧ senseFault = False

⇒ (data? ∈ (q2 − iqr div 2) .. (q2 + iqr div 2) ⇒ strangeSamples′ = 0) ∧

(data? ∉ (q2 − iqr div 2) .. (q2 + iqr div 2) ⇒ strangeSamples′ =

strangeSamples + 1)

#sampleWindow′ < maxSize * 85 div 100 ∨ senseFault = True ⇒

strangeSamples′ = strangeSamples

q2! = q2′

iqr! = iqr′

factory′ = factory

∃ os! : iSAMPLE

• SampleFactoryGetSample[factory/os?,data?/value?] ∧

[| sampleWindow′ = ⟨ os! ⟩

oldest′ = {os!}

newest′ = {os!}]

└

┌─ PreiConsumptionEstimatorNewDataRoomForSamples

PrełiConsumptionEstimatorNewData

ĐiConsumptionEstimator

|

ℝiConsumptionEstimator

data? ≥ 0

0 < #sampleWindow < maxSize

└

┌─ PostiConsumptionEstimatorNewDataRoomForSamples

PostłiConsumptionEstimatorNewData

ΔĐiConsumptionEstimator

|

ΔℝiConsumptionEstimator

this′ = this

maxSize′ = maxSize

minConsecutiveStrangeSamples′ =minConsecutiveStrangeSamples

#sampleWindow′ ≥ maxSize * 85 div 100 ∧ senseFault = False

⇒ (data? ∈ (q2 − iqr div 2) .. (q2 + iqr div 2) ⇒ strangeSamples′ = 0) ∧

(data? ∉ (q2 − iqr div 2) .. (q2 + iqr div 2) ⇒ strangeSamples′ =

strangeSamples + 1)

#sampleWindow′ < maxSize * 85 div 100 ∨ senseFault = True ⇒

strangeSamples′ = strangeSamples

q2! = q2′

iqr! = iqr′

factory′ = factory

∃ os! , oldNewest : iSAMPLE

| newest = {oldNewest}

• SampleFactoryGetSample[factory/os?,data?/value?] ⨟

iSampleSetNext[oldNewest/os?,os!/n?] ∧

[| sampleWindow′ = sampleWindow ↾ {s : ran sampleWindow ; v : ℝ |

iSampleGetValue[s/os?,v/value!] ∧ v ≤ data? • s} ⌢ ⟨ os! ⟩ ⌢

sampleWindow ↾ {s : ran sampleWindow ; v : ℝ |

```
iSampleGetValue[s/os?,v/value!] ∧ v > data? • s}
      oldest' = oldest
      newest' = {os!}]
└
┌─ PreiConsumptionEstimatorNewDataNoRoomForSamples
 PrełiConsumptionEstimatorNewData
 ÐiConsumptionEstimator
|
 ℝiConsumptionEstimator
 data? ≥ 0
 #sampleWindow = maxSize
└
┌─ PostiConsumptionEstimatorNewDataNoRoomForSamples
 PostłiConsumptionEstimatorNewData
 ΔÐiConsumptionEstimator
|
 ΔℝiConsumptionEstimator
 this' = this
 maxSize' = maxSize
 minConsecutiveStrangeSamples' =minConsecutiveStrangeSamples
 #sampleWindow' ≥ maxSize * 85 div 100 ∧ senseFault = False
 ⇒ (data? ∈ (q2 − iqr div 2) .. (q2 + iqr div 2) ⇒ strangeSamples' = 0) ∧
      (data? ∉ (q2 − iqr div 2) .. (q2 + iqr div 2) ⇒ strangeSamples' =
strangeSamples + 1)
 #sampleWindow' < maxSize * 85 div 100 ∨ senseFault = True ⇒
strangeSamples' = strangeSamples
 q2! = q2'
 iqr! = iqr'
```

factory′ = factory

∃ os! , oldOldest , oldNewest , newOldest : iSAMPLE

| oldest = {oldOldest} ∧ newest = {oldNewest}

• SampleFactoryGetSample[factory/os?,data?/value?] ⨾

iSampleSetNext[oldNewest/os?,os!/n?] ⨾

iSampleGetNext[oldOldest/os?,newOldest/next!] ∧

[| sampleWindow′ = sampleWindow ↾ {s : ran sampleWindow ⧵ {oldOldest} ; v : ℝ | iSampleGetValue[s/os?,v/value!] ∧ v ≤ data? • s} ⌢ ⟨ os! ⟩ ⌢ sampleWindow ↾ {s : ran sampleWindow ⧵ {oldOldest} ; v : ℝ | iSampleGetValue[s/os?,v/value!] ∧ v > data? • s}

oldest′ = {newOldest}

newest′ = {os!}]

└

—

iConsumptionEstimatorNewData ==

∃ ΔłiConsumptionEstimator

• ƒiConsumptionEstimator ∧

((PreiConsumptionEstimatorNewDataNoSamples ∧ PostiConsumptionEstimatorNewDataNoSamples) ∨

(PreiConsumptionEstimatorNewDataRoomForSamples ∧ PostiConsumptionEstimatorNewDataRoomForSamples) ∨

(PreiConsumptionEstimatorNewDataNoRoomForSamples ∧ PostiConsumptionEstimatorNewDataNoRoomForSamples))

└

┌ PreiConsumptionEstimatorIsSenseFault

PrełiConsumptionEstimatorIsSenseFault

ÐiConsumptionEstimator

|

ℝiConsumptionEstimator

└

┌ PostiConsumptionEstimatorIsSenseFault

PostłiConsumptionEstimatorIsSenseFault

Ξ ÐiConsumptionEstimator

|

ΔℝiConsumptionEstimator

senseFault! = senseFault

└

—

iConsumptionEstimatorIsSenseFault == ∃ Ξ łiConsumptionEstimator •

ƒiConsumptionEstimator ∧ PreiConsumptionEstimatorIsSenseFault ∧

PostiConsumptionEstimatorIsSenseFault

└

5.4.2.5 La classe ConsumptionEstimatorImpl

Si passa ora a definire la classe implementativa dello stimatore. In tale progetto di sceglie di rappresentare la lista di campioni come un albero binario di ricerca. In questo modo è possibile garantire automaticamente il vincolo di ordinamento ed allo stesso tempo avere accessi efficienti alla struttura.

— section ConsumptionEstimatorImplIntensional parents iSampleIntensional

└

┌ łConsumptionEstimatorImpl

łClass

sampleWindow : bst iSAMPLE ◀ ORDER

maxSize : ℕ↘1↖

oldest : iseq iSAMPLE

newest : iseq iSAMPLE

q1 : ℝ

q2 : ℝ

q3 : ℝ

iqr : ℝ

minConsecutiveStrangeSamples : ℕ↘1↖

strangeSamples : ℕ

senseFault : □□

factory : SAMPLE_FACTORY

|

this ∈ CONSUMPTION_ESTIMATOR_IMPL

size(sampleWindow.1) ≤ maxSize

#oldest ≤ 1

#newest ≤ 1

∀ s : ran oldest • sampleWindow containsBst s

∀ s : ran newest • sampleWindow containsBst s

maxSize ≥ 7

strangeSamples ≤ maxSize

maxSize * 5 div 100 < minConsecutiveStrangeSamples ≤ maxSize * 15 div 100

└

┌ ł ConsumptionEstimatorImplInit

ł ClassInit

ł ConsumptionEstimatorImpl ’

maxSize? : ℕ↘1↖

minConsecutiveStrangeSamples? : ℕ↘1↖

factory? : SAMPLE_FACTORY

└

┌─ łConsumptionEstimatorImplFin

łClassFin

łConsumptionEstimatorImpl

└

┌─ PrełConsumptionEstimatorImplTrim

łConsumptionEstimatorImpl

└

┌─ PostłConsumptionEstimatorImplTrim

ΔłConsumptionEstimatorImpl

PrełConsumptionEstimatorImplTrim

└

┌─ PrełConsumptionEstimatorImplNewData

łConsumptionEstimatorImpl

data? : ℝ

└

┌─ PostłConsumptionEstimatorImplNewData

ΔłConsumptionEstimatorImpl

PrełConsumptionEstimatorImplNewData

q2! : ℝ

iqr! : ℝ

└

┌─ PrełConsumptionEstimatorImplIsSenseFault
 łConsumptionEstimatorImpl
└

┌─ PostłConsumptionEstimatorImplIsSenseFault
 ΞłConsumptionEstimatorImpl
 PrełConsumptionEstimatorImplIsSenseFault
 senseFault! : □□
└

— section ConsumptionEstimatorImplRelational parents
ConsumptionEstimatorImplIntensional , iSampleRelational ,
SampleFactoryRelational
└

. . .

┌─ ℝConsumptionEstimatorImpl
 ℝClass
 łConsumptionEstimatorImpl
 ΩiSample
 ΩSampleFactory
 ΩConsumptionEstimatorImpl
|
 ∀ s : nodes(sampleWindow.1) • s ∈ dom ŁiSample
 factory ∈ dom ŁSampleFactory
 size(sampleWindow.1) > 0 ⇒
let np25 == 25 div 100 * size(sampleWindow.1) ;

np50 == 50 div 100 * size(sampleWindow.1) ;

np75 == 75 div 100 * size(sampleWindow.1)

• [| np25 ∉ ℕ ⇒ q1 = (ŁiSample ((inorder sampleWindow) (ceiling np25))).value

np25 ∈ ℕ ⇒ q1 = ((ŁiSample ((inorder sampleWindow) np25)).value + (ŁiSample ((inorder sampleWindow) (np25+1))).value) div 2

np50 ∉ ℕ ⇒ q2 = (ŁiSample ((inorder sampleWindow) (ceiling np50))).value

np50 ∈ ℕ ⇒ q2 = ((ŁiSample ((inorder sampleWindow) np50)).value + (ŁiSample ((inorder sampleWindow) (np50+1))).value) div 2

np75 ∉ ℕ ⇒ q3 = (ŁiSample ((inorder sampleWindow) (ceiling np75))).value

np75 ∈ ℕ ⇒ q3 = ((ŁiSample ((inorder sampleWindow) np75)).value + (ŁiSample ((inorder sampleWindow) (np75+1))).value) div 2

iqr = q3 − q1]

strangeSamples ≥ minConsecutiveStrangeSamples ∧ size(sampleWindow.1) ≥ maxSize * 85 div 100 ⇔ senseFault = True

∟

Si noti come il progettista abbia scelto di spostare alcuni invarianti definiti nell'intensione allo schema relazionale, in modo tale da *rilassare i vincoli imposti ad una evoluzione consistente* di un'istanza di ConsumptionEstimatorImpl. Questo consente di ottenere con maggiore facilità una sequenza di azioni valida per gli eventi della classe.

— section ConsumptionEstimatorImplExtensional parents ConsumptionEstimatorImplRelational , iConsumptionEstimatorExtensional

└

. . .

┌─ ÐConsumptionEstimatorImpl

ÐiConsumptionEstimator

ÆConsumptionEstimatorImpl

|

ConsumptionEstimatorImplIsClass

ConsumptionEstimatorImplImplementsiConsumptionEstimator

└

. . .

┌─ ConsumptionEstimatorImplInit

łConsumptionEstimatorImplInit

ΔÐConsumptionEstimatorImpl

|

ℝConsumptionEstimatorImpl ′

ClassInit

this? ∈ CONSUMPTION_ESTIMATOR_IMPL

maxSize? ≥ 7

maxSize? * 5 div 100 < minConsecutiveStrangeSamples? ≤ maxSize? * 15 div 100

factory? ∈ dom łSampleFactory

sampleWindow′ = ∅ ↦ ℕiSample

maxSize′ = maxSize?

oldest′ = ⟨ ⟩

newest′ = ⟨ ⟩

minConsecutiveStrangeSamples′ = minConsecutiveStrangeSamples?

strangeSamples′ = 0

senseFault′ = False

factory′ = factory?

└

—

ConsumptionEstimatorImplNew == ∃ łConsumptionEstimatorImpl ′

• ƒConsumptionEstimatorImplN ∧

ConsumptionEstimatorImplInit[oc!/this?]

└

┌ ConsumptionEstimatorImplNewImpl

ConsumptionEstimatorImplInit

|

ConsumptionEstimatorImplNewImpl_1

└

┌ ConsumptionEstimatorImplNewImpl_1_1

łConsumptionEstimatorImpl ′

ΔÐConsumptionEstimatorImpl

this? : INSTANCE

maxSize? : ℕ↘1↖

minConsecutiveStrangeSamples? : ℕ↘1↖

factory? : SAMPLE_FACTORY

|

this? ∈ CONSUMPTION_ESTIMATOR_IMPL

maxSize? ≥ 7

maxSize? * 5 div 100 < minConsecutiveStrangeSamples? ≤ maxSize? * 15 div 100

factory? ∈ dom łSampleFactory

ClassInitImpl

```
└
┌─ □[C]ConsumptionEstimatorImplNewImpl_1_2
 ł ConsumptionEstimatorImpl ’
 maxSize? : ℕ↘1↖
 minConsecutiveStrangeSamples? : ℕ↘1↖
 factory? : SAMPLE_FACTORY
|
 sampleWindow’ = ∅ ↦ ℕ iSample
 maxSize’ = maxSize?
 oldest’ = ⟨ ⟩
 newest’ = ⟨ ⟩
 minConsecutiveStrangeSamples’ = minConsecutiveStrangeSamples?
 strangeSamples’ = 0
 senseFault’ = False
 q1’ = q2’ = q3’ = iqr’ = -1
 factory’ = factory?
└
—
 □[C]ConsumptionEstimatorImplNewImpl_1 ==
□[C]ConsumptionEstimatorImplNewImpl_1_1 ∧
□[C]ConsumptionEstimatorImplNewImpl_1_2
└

┌─ ConsumptionEstimatorImplFin
 ł ConsumptionEstimatorImplFin
 ΔÐ ConsumptionEstimatorImpl
|
 ℝ ConsumptionEstimatorImpl
```

ClassFin

⌙

—

ConsumptionEstimatorImplDelete == ∃ łConsumptionEstimatorImpl •

ƒConsumptionEstimatorImplD ∧ ConsumptionEstimatorImplFin

⌙

┌─ ConsumptionEstimatorImplDeleteImpl

ConsumptionEstimatorImplFin

|

□□ConsumptionEstimatorImplDeleteImpl_1

⌙

┌─ □□ConsumptionEstimatorImplDeleteImpl_1

łConsumptionEstimatorImpl

ΔÐConsumptionEstimatorImpl

|

ClassFinImpl

⌙

┌─ PreConsumptionEstimatorImplTrim

PrełConsumptionEstimatorImplTrim

ÐConsumptionEstimatorImpl

|

ℝConsumptionEstimatorImpl

⌙

┌─ PostConsumptionEstimatorImplTrim

PostłConsumptionEstimatorImplTrim

ΔÐConsumptionEstimatorImpl

|

ΔℝConsumptionEstimatorImpl

this′ = this

sampleWindow′ = ∅ ↦ ℕiSample

maxSize′ = maxSize

oldest′ = ⟨ ⟩

newest′ = ⟨ ⟩

minConsecutiveStrangeSamples′ = minConsecutiveStrangeSamples

strangeSamples′ = 0

senseFault′ = False

factory′ = factory

└

—

ConsumptionEstimatorImplTrimSpec == PreConsumptionEstimatorImplTrim ∧ PostConsumptionEstimatorImplTrim

└

—

ConsumptionEstimatorImplTrim == ∃ ΔłConsumptionEstimatorImpl • ƒConsumptionEstimatorImpl ∧ ConsumptionEstimatorImplTrimSpec

└

┌─ ConsumptionEstimatorImplTrimImpl

ConsumptionEstimatorImplTrimSpec

|

□ConsumptionEstimatorImplTrimImpl_1

└

┌─ □ConsumptionEstimatorImplTrimImpl_1

ΔłConsumptionEstimatorImpl

|

this′ = this

sampleWindow′ = ∅ ↦ ℕiSample

maxSize′ = maxSize

oldest′ = ⟨ ⟩

newest′ = ⟨ ⟩

minConsecutiveStrangeSamples′ = minConsecutiveStrangeSamples

strangeSamples′ = 0

senseFault′ = False

q1′ = q2′ = q3′ = iqr′ = -1

factory′ = factory

└

┌ PreConsumptionEstimatorImplNewDataNoSamples

PrełConsumptionEstimatorImplNewData

ÐConsumptionEstimatorImpl

|

ℝConsumptionEstimatorImpl

data? ≥ 0

size(sampleWindow.1) = 0

└

┌ PostConsumptionEstimatorImplNewDataNoSamples

PostłConsumptionEstimatorImplNewData

ΔÐConsumptionEstimatorImpl

|

ΔℝConsumptionEstimatorImpl

this′ = this

maxSize′ = maxSize

minConsecutiveStrangeSamples′ = minConsecutiveStrangeSamples

size(sampleWindow′.1) ≥ maxSize * 85 div 100 ∧ senseFault = False

⇒ (data? ∈ (q2 − iqr div 2) .. (q2 + iqr div 2) ⇒ strangeSamples′ = 0) ∧
(data? ∉ (q2 − iqr div 2) .. (q2 + iqr div 2) ⇒ strangeSamples′ =
strangeSamples + 1)
size(sampleWindow′.1) < maxSize * 85 div 100 ∨ senseFault = True ⇒
strangeSamples′ = strangeSamples
q2! = q2′
iqr! = iqr′
factory′ = factory
∃ os! : iSAMPLE
• SampleFactoryGetSample[factory/os?,data?/value?] ∧
[| sampleWindow′ = add(sampleWindow,os!)
oldest′ = ⟨os!⟩
newest′ = ⟨os!⟩]
└
┌─ PreConsumptionEstimatorImplNewDataRoomForSamples
PrełConsumptionEstimatorImplNewData
ÐConsumptionEstimatorImpl
|
ℝConsumptionEstimatorImpl
data? ≥ 0
0 < size(sampleWindow.1) < maxSize
└
┌─ PostConsumptionEstimatorImplNewDataRoomForSamples
PostłConsumptionEstimatorImplNewData
ΔÐConsumptionEstimatorImpl
|
ΔℝConsumptionEstimatorImpl
this′ = this

maxSize′ = maxSize

minConsecutiveStrangeSamples′ = minConsecutiveStrangeSamples

size(sampleWindow′.1) ≥ maxSize * 85 div 100 ∧ senseFault = False

⇒ (data? ∈ (q2 − iqr div 2) .. (q2 + iqr div 2) ⇒ strangeSamples′ = 0) ∧

(data? ∉ (q2 − iqr div 2) .. (q2 + iqr div 2) ⇒ strangeSamples′ =

strangeSamples + 1)

size(sampleWindow′.1) < maxSize * 85 div 100 ∨ senseFault = True ⇒

strangeSamples′ = strangeSamples

q2! = q2′

iqr! = iqr′

factory′ = factory

∃ os! , oldNewest : iSAMPLE

| oldNewest = head newest

• SampleFactoryGetSample[factory/os?,data?/value?] ⨾

iSampleSetNext[oldNewest/os?,os!/n?] ∧

[| sampleWindow′ = add(sampleWindow,os!)

oldest′ = oldest

newest′ = ⟨os!⟩]

└

┌─ PreConsumptionEstimatorImplNewDataNoRoomForSamples

PreConsumptionEstimatorImplNewData

ΘConsumptionEstimatorImpl

|

ℝConsumptionEstimatorImpl

data? ≥ 0

size(sampleWindow.1) = maxSize

└

┌─ PostConsumptionEstimatorImplNewDataNoRoomForSamples

```
  PostłConsumptionEstimatorImplNewData
  ΔÐConsumptionEstimatorImpl
|
  ΔℝConsumptionEstimatorImpl
  this′ = this
  maxSize′ = maxSize
  minConsecutiveStrangeSamples′ = minConsecutiveStrangeSamples
  size(sampleWindow′.1) ≥ maxSize * 85 div 100 ∧ senseFault = False
  ⇒  (data? ∈ (q2 − iqr div 2) .. (q2 + iqr div 2) ⇒ strangeSamples′ = 0) ∧
       (data? ∉ (q2 − iqr div 2) .. (q2 + iqr div 2) ⇒ strangeSamples′ =
strangeSamples + 1)
  size(sampleWindow′.1) < maxSize * 85 div 100 ∨ senseFault = True ⇒
strangeSamples′ = strangeSamples
  q2! = q2′
  iqr! = iqr′
  factory′ = factory
  ∃ os! , oldOldest , oldNewest , newOldest : iSAMPLE
  | oldOldest = head oldest ∧ oldNewest = head newest
  • SampleFactoryGetSample[factory/os?,data?/value?] ⨟
    iSampleSetNext[oldNewest/os?,os!/n?] ⨟
    iSampleGetNext[oldOldest/os?,newOldest/next!] ∧
    [| sampleWindow′ = add(remove(sampleWindow,oldOldest),os!)
      oldest′ = ⟨newOldest⟩
      newest′ = ⟨os!⟩ ]
└
—
  ConsumptionEstimatorImplNewDataSpec ==
  (PreConsumptionEstimatorImplNewDataNoSamples ∧
```

PostConsumptionEstimatorImplNewDataNoSamples) ∨

(PreConsumptionEstimatorImplNewDataRoomForSamples ∧

PostConsumptionEstimatorImplNewDataRoomForSamples) ∨

(PreConsumptionEstimatorImplNewDataNoRoomForSamples ∧

PostConsumptionEstimatorImplNewDataNoRoomForSamples)

└

─

ConsumptionEstimatorImplNewData == ∃ ΔłConsumptionEstimatorImpl

• ƒConsumptionEstimatorImpl ∧

ConsumptionEstimatorImplNewDataSpec

└

┌ ConsumptionEstimatorImplNewDataImpl_1_1

ΞłConsumptionEstimatorImpl

data? : ℝ

sizeBefore : ℕ

f : SAMPLE_FACTORY

|

data? ≥ 0

sizeBefore = size(sampleWindow.1)

f = factory

└

┌ ConsumptionEstimatorImplNewDataImpl_1_2

ΔÐSampleFactory

data? : ℝ

f : SAMPLE_FACTORY

os : iSAMPLE

|

SampleFactoryGetSample[f/os?,data?/value?,os/os!]

⌙

—

□[C]ConsumptionEstimatorImplNewDataImpl_1 == ∃ f : SAMPLE_FACTORY •
□[C]ConsumptionEstimatorImplNewDataImpl_1_1 ∧
□[C]ConsumptionEstimatorImplNewDataImpl_1_2

⌙

┌─ □[C]ConsumptionEstimatorImplNewDataImpl_2_NoSamples_1
ΔłConsumptionEstimatorImpl
sizeBefore : ℕ
os : iSAMPLE
|
sizeBefore = 0
sampleWindow’ = add(sampleWindow,os)
oldest’ = ⟨os⟩
newest’ = ⟨os⟩
Ξ łConsumptionEstimatorImpl ⧵ (sampleWindow,oldest,newest)

⌙

—

□[C]ConsumptionEstimatorImplNewDataImpl_2_NoSamples ==
□[C]ConsumptionEstimatorImplNewDataImpl_2_NoSamples_1 ∧
Ξ ÐSampleFactory

⌙

┌─ □[C]ConsumptionEstimatorImplNewDataImpl_2_RoomForSamples_1
ΔłConsumptionEstimatorImpl
sizeBefore : ℕ
os : iSAMPLE
oldNewest : iSAMPLE
|

0 < sizeBefore < maxSize

sampleWindow′ = add(sampleWindow,os)

newest′ = ⟨os⟩

oldNewest = head newest

Ξ ConsumptionEstimatorImpl \ (sampleWindow,newest)

└

┌─ ConsumptionEstimatorImplNewDataImpl_2_RoomForSamples_2

ΔSampleFactory

oldNewest : iSAMPLE

os : iSAMPLE

|

iSampleSetNext[oldNewest/os?,os/n?]

└

—

ConsumptionEstimatorImplNewDataImpl_2_RoomForSamples

== ∃ oldNewest : iSAMPLE •

ConsumptionEstimatorImplNewDataImpl_2_RoomForSamples_1 ∧

ConsumptionEstimatorImplNewDataImpl_2_RoomForSamples_2

└

┌─ ConsumptionEstimatorImplNewDataImpl_2_NoRoomForSamples_1

Ξ ConsumptionEstimatorImpl

sizeBefore : ℕ

oldOldest : iSAMPLE

oldNewest : iSAMPLE

|

sizeBefore = maxSize

oldOldest = head oldest

oldNewest = head newest

└

┌ □ConsumptionEstimatorImplNewDataImpl_2_NoRoomForSamples_2

ΔÐSampleFactory

oldOldest : iSAMPLE

oldNewest : iSAMPLE

os : iSAMPLE

newOldest : iSAMPLE

|

iSampleSetNext[oldNewest/os?,os/n?] ;

iSampleGetNext[oldOldest/os?,newOldest/next!]

└

┌ □ConsumptionEstimatorImplNewDataImpl_2_NoRoomForSamples_3

ΔłConsumptionEstimatorImpl

oldOldest : iSAMPLE

newOldest : iSAMPLE

os : iSAMPLE

|

sampleWindow′ = add(remove(sampleWindow,oldOldest),os)

oldest′ = ⟨newOldest⟩

newest′ = ⟨os⟩

Ξ łConsumptionEstimatorImpl ↖ (sampleWindow,oldest,newest)

└

—

□ConsumptionEstimatorImplNewDataImpl_2_NoRoomForSamples

== ∃ oldOldest , oldNewest , newOldest : iSAMPLE •

□ConsumptionEstimatorImplNewDataImpl_2_NoRoomForSamples_1 ;

(□ConsumptionEstimatorImplNewDataImpl_2_NoRoomForSamples_2 ∧

□ConsumptionEstimatorImplNewDataImpl_2_NoRoomForSamples_3)

└

—

□ConsumptionEstimatorImplNewDataImpl_2 ==

□ConsumptionEstimatorImplNewDataImpl_2_NoSamples ∨

□ConsumptionEstimatorImplNewDataImpl_2_RoomForSamples ∨

□ConsumptionEstimatorImplNewDataImpl_2_NoRoomForSamples

└

┌ □ConsumptionEstimatorImplNewDataImpl_3_1

ΔConsumptionEstimatorImpl

data? : ℝ

sizeAfter : ℕ

|

sizeAfter = size(sampleWindow.1)

sizeAfter ≥ maxSize * 85 div 100 ∧ senseFault = False

⇒ (data? ∈ (q2 − iqr div 2) .. (q2 + iqr div 2) ⇒ strangeSamples′ = 0) ∧

(data? ∉ (q2 − iqr div 2) .. (q2 + iqr div 2) ⇒ strangeSamples′ =

strangeSamples + 1)

sizeAfter < maxSize * 85 div 100 ∨ senseFault = True ⇒ strangeSamples′ =

strangeSamples

ΞConsumptionEstimatorImpl \ (strangeSamples)

└

—

□ConsumptionEstimatorImplNewDataImpl_3 ==

□ConsumptionEstimatorImplNewDataImpl_3_1 ∧ ΞDSampleFactory

└

┌ □ConsumptionEstimatorImplNewDataImpl_4_1

ΞConsumptionEstimatorImpl

sizeAfter : ℕ

w : iseq iSAMPLE

np : seq ℝ

|

w = inorder sampleWindow

np = ⟨ 25 div 100 * sizeAfter , 50 div 100 * sizeAfter , 75 div 100 * sizeAfter ⟩

└

┌ □ ComputeQ

≡ ÐSampleFactory

w? : iseq iSAMPLE

np? : ℝ

q! : ℝ

|

np? ∉ ℕ ⇒ ∃ s : iSAMPLE

| s = w?(ceiling(np?))

• iSampleGetValue[s/os?,q!/value!]

np? ∈ ℕ ⇒ ∃ s1 , s2 : iSAMPLE ; v1 , v2 : ℝ

| s1 = w?(np?) ∧

s2 = w?(np?+1)

• iSampleGetValue[s1/os?,v1/value!] ⨟

iSampleGetValue[s2/os?,v2/value!] ∧

[| q! = (v1 + v2) div 2]

└

┌ □ ConsumptionEstimatorImplNewDataImpl_4_2

≡ ÐSampleFactory

w : iseq iSAMPLE

np : seq ℝ

computedQ : seq ℝ

|

np = ⟨ ⟩ ⇒ computedQ = ⟨ ⟩

np ≠ ⟨ ⟩ ⇒ ∃ nph , qh : ℝ ; tNp , tComputedQ : seq ℝ

| nph = head np ∧

tNp = tail np

• ComputeQ[w/w?,nph/np?,qh/q!] ⨾

ConsumptionEstimatorImplNewDataImpl_4_2[tNp/np,tComputedQ/computedQ

] ∧

[| computedQ = ⟨qh⟩ ⌢ tComputedQ]

└

Si noti come questa azione implichi se stessa in modo ricorsivo, realizzando a tutti gli effetti il calcolo dei tre quantili tramite una *iterazione*.

┌ ConsumptionEstimatorImplNewDataImpl_4_3

ΔConsumptionEstimatorImpl

computedQ : seq ℝ

sizeAfter : ℕ

q2! : ℝ

iqr! : ℝ

|

q1′ = computedQ(1)

q2′ = computedQ(2)

q3′ = computedQ(3)

iqr′ = computedQ(3) − computedQ(1)

strangeSamples ≥ minConsecutiveStrangeSamples ∧ sizeAfter ≥ maxSize *

85 div 100 ⇒ senseFault′ = True

strangeSamples < minConsecutiveStrangeSamples ∨ sizeAfter < maxSize *

85 div 100 ⇒ senseFault′ = False

q2! = q2′

iqr! = iqr′

≡ ConsumptionEstimatorImpl \ (q1,q2,q3,iqr,senseFault)

[ConsumptionEstimatorImplNewDataImpl_4

== ∃ w : iseq iSAMPLE ; np , computedQ : seq ℝ •

[ConsumptionEstimatorImplNewDataImpl_4_1 ⨾

([ConsumptionEstimatorImplNewDataImpl_4_2 ∧

[ConsumptionEstimatorImplNewDataImpl_4_3)

ConsumptionEstimatorImplNewDataImpl

ConsumptionEstimatorImplNewDataSpec

∃ os : iSAMPLE ; sizeBefore , sizeAfter : ℕ

• [ConsumptionEstimatorImplNewDataImpl_1 ⨾

[ConsumptionEstimatorImplNewDataImpl_2 ⨾

[ConsumptionEstimatorImplNewDataImpl_3 ⨾

[ConsumptionEstimatorImplNewDataImpl_4

PreConsumptionEstimatorImplIsSenseFault

Pre ConsumptionEstimatorImplIsSenseFault

Đ ConsumptionEstimatorImpl

ℝ ConsumptionEstimatorImpl

PostConsumptionEstimatorImplIsSenseFault

Post ConsumptionEstimatorImplIsSenseFault

≡ Đ ConsumptionEstimatorImpl

|

ΔℝConsumptionEstimatorImpl

senseFault! = senseFault

└

—

ConsumptionEstimatorImplIsSenseFaultSpec ==

PreConsumptionEstimatorImplIsSenseFault ∧

PostConsumptionEstimatorImplIsSenseFault

└

—

ConsumptionEstimatorImplIsSenseFault == ∃ Ξ𝕀ConsumptionEstimatorImpl

• ƒConsumptionEstimatorImpl ∧

ConsumptionEstimatorImplIsSenseFaultSpec

└

┌ ConsumptionEstimatorImplIsSenseFaultImpl

ConsumptionEstimatorImplIsSenseFaultSpec

|

□□ConsumptionEstimatorImplIsSenseFaultImpl_1

└

┌ □□ConsumptionEstimatorImplIsSenseFaultImpl_1

Ξ𝕀ConsumptionEstimatorImpl

senseFault! : □□

|

senseFault! = senseFault

└

5.4.2.6 La classe ConsumptionEstimatorFactory

L'ultima classe rimasta è la factory dello stimatore, la quale utilizza una factory di iSample.

```
— section ConsumptionEstimatorFactoryIntensional parents
DomainModelDesign , ClassIntensional
└

┌─ łConsumptionEstimatorFactory
 łClass
 sampleFactory : SAMPLE_FACTORY
|
 this ∈ CONSUMPTION_ESTIMATOR_FACTORY
└

┌─ łConsumptionEstimatorFactoryInit
 łClassInit
 łConsumptionEstimatorFactory ’
 sampleFactory? : SAMPLE_FACTORY
└

┌─ łConsumptionEstimatorFactoryFin
 łClassFin
 łConsumptionEstimatorFactory
└

┌─ PrełConsumptionEstimatorFactoryGetConsumptionEstimator
 łConsumptionEstimatorFactory
 maxSize? : ℕ↘1↖
```

minConsecutiveStrangeSamples? : ℕ↘1↖

└

┌─ PostłConsumptionEstimatorFactoryGetConsumptionEstimator

ΔłConsumptionEstimatorFactory

PrełConsumptionEstimatorFactoryGetConsumptionEstimator

oc! : iCONSUMPTION_ESTIMATOR

└

— section ConsumptionEstimatorFactoryRelational parents

ConsumptionEstimatorFactoryIntensional , SampleFactoryRelational

└

. . .

┌─ ℝConsumptionEstimatorFactory

ℝClass

łConsumptionEstimatorFactory

ΩSampleFactory

ΩConsumptionEstimatorFactory

|

sampleFactory ∈ dom ŁSampleFactory

└

— section ConsumptionEstimatorFactoryExtensional parents

ConsumptionEstimatorFactoryRelational ,

ConsumptionEstimatorImplExtensional

└

. . .

┌─ ÐConsumptionEstimatorFactory
ÐConsumptionEstimatorImpl
ÆConsumptionEstimatorFactory
|
ConsumptionEstimatorFactoryIsClass
└

. . .

┌─ ConsumptionEstimatorFactoryInit
łConsumptionEstimatorFactoryInit
ΔÐConsumptionEstimatorFactory
|
ℝConsumptionEstimatorFactory ’
ClassInit
this? ∈ CONSUMPTION_ESTIMATOR_FACTORY
sampleFactory? ∈ dom ŁSampleFactory
sampleFactory’ = sampleFactory?
└

—
ConsumptionEstimatorFactoryNew == ∃ łConsumptionEstimatorFactory ’
• ƒConsumptionEstimatorFactoryN ∧
ConsumptionEstimatorFactoryInit[oc!/this?]
└

┌─ ConsumptionEstimatorFactoryNewImpl
ConsumptionEstimatorFactoryInit
|

```
  ConsumptionEstimatorFactoryNewImpl_1
└
┌─ ConsumptionEstimatorFactoryNewImpl_1_1
 ł ConsumptionEstimatorFactory ’
 Δ Đ ConsumptionEstimatorFactory
 this? : INSTANCE
 sampleFactory? : SAMPLE_FACTORY
|
 this? ∈ CONSUMPTION_ESTIMATOR_FACTORY
 sampleFactory? ∈ dom Ł SampleFactory
 ClassInitImpl
└
┌─ ConsumptionEstimatorFactoryNewImpl_1_2
 ł ConsumptionEstimatorFactory ’
 sampleFactory? : SAMPLE_FACTORY
|
 sampleFactory’ = sampleFactory?
└
—
  ConsumptionEstimatorFactoryNewImpl_1 ==
 ConsumptionEstimatorFactoryNewImpl_1_1 ∧
 ConsumptionEstimatorFactoryNewImpl_1_2
└


┌─ ConsumptionEstimatorFactoryFin
 ł ConsumptionEstimatorFactoryFin
 Δ Đ ConsumptionEstimatorFactory
|
```

ℝConsumptionEstimatorFactory

ClassFin

└

—

ConsumptionEstimatorFactoryDelete == ∃ łConsumptionEstimatorFactory

• ƒConsumptionEstimatorFactoryD ∧

ConsumptionEstimatorFactoryFin

└

┌─ ConsumptionEstimatorFactoryDeleteImpl

ConsumptionEstimatorFactoryFin

|

□□ConsumptionEstimatorFactoryDeleteImpl_1

└

┌─ □□ConsumptionEstimatorFactoryDeleteImpl_1

łConsumptionEstimatorFactory

ΔÐConsumptionEstimatorFactory

|

ClassFinImpl

└

┌─ PreConsumptionEstimatorFactoryGetConsumptionEstimator

PrełConsumptionEstimatorFactoryGetConsumptionEstimator

ÐConsumptionEstimatorFactory

|

ℝConsumptionEstimatorFactory

maxSize? ≥ 7

maxSize? * 5 div 100 < minConsecutiveStrangeSamples? ≤ maxSize? * 15 div

100

└

┌─ PostConsumptionEstimatorFactoryGetConsumptionEstimator

PostłConsumptionEstimatorFactoryGetConsumptionEstimator

ΔÐConsumptionEstimatorFactory

|

ΔℝConsumptionEstimatorFactory

this′ = this

sampleFactory′ = sampleFactory

ConsumptionEstimatorImplNew[sampleFactory/factory?]

└

—

ConsumptionEstimatorFactoryGetConsumptionEstimatorSpec

== PreConsumptionEstimatorFactoryGetConsumptionEstimator ∧ PostConsumptionEstimatorFactoryGetConsumptionEstimator

└

—

ConsumptionEstimatorFactoryGetConsumptionEstimator ==

∃ ΔłConsumptionEstimatorFactory

• ƒConsumptionEstimatorFactory ∧ ConsumptionEstimatorFactoryGetConsumptionEstimatorSpec

└

┌─ ConsumptionEstimatorFactoryGetConsumptionEstimatorImpl

ConsumptionEstimatorFactoryGetConsumptionEstimatorSpec

|

□□ConsumptionEstimatorFactoryGetConsumptionEstimatorImpl_1

└

┌─ □□ConsumptionEstimatorFactoryGetConsumptionEstimatorImpl_1_1

Ξ łConsumptionEstimatorFactory

maxSize? : ℕ↘1↖

minConsecutiveStrangeSamples? : ℕ↘1↖

sf : SAMPLE_FACTORY

|

maxSize? ≥ 7

maxSize? * 5 div 100 < minConsecutiveStrangeSamples? ≤ maxSize? * 15 div 100

sf = sampleFactory

└

┌─ □Ⓒ ConsumptionEstimatorFactoryGetConsumptionEstimatorImpl_1_2

ΔÐ ConsumptionEstimatorFactory

maxSize? : ℕ↘1↖

minConsecutiveStrangeSamples? : ℕ↘1↖

sf : SAMPLE_FACTORY

oc! : iCONSUMPTION_ESTIMATOR

|

ConsumptionEstimatorImplNew[sf/factory?]

└

—

□Ⓒ ConsumptionEstimatorFactoryGetConsumptionEstimatorImpl_1 ==

∃ sf : SAMPLE_FACTORY •

□Ⓒ ConsumptionEstimatorFactoryGetConsumptionEstimatorImpl_1_1 ∧

□Ⓒ ConsumptionEstimatorFactoryGetConsumptionEstimatorImpl_1_2

└

5.5 Sommario dei concetti introdotti

In questo capitolo si è estesa la notazione Concepts Z introducendo concetti che la

rendono in grado di supportare pienamente anche la fase di progetto. In particolare si è introdotto supporto, al livello primitivo, al concetto di albero, il quale consente piena libertà nella specifica delle rappresentazioni di stato. Si è inoltre introdotto il concetto di interfaccia, che consente di raffinare la struttura interna di un blocco secondo pattern di progettazione scelti. Infine si è introdotto il concetto di azione, il quale consente di raffinare la specifica del comportamento di un blocco fino ad un livello passibile di esecuzione.

Si noti che analogamente a quanto si è fatto negli esempi del paragrafo 5.4.2, è possibile dare una specifica di progetto per tutti i concetti introdotti nel capitolo 4, producendo dei componenti di base riusabili nelle successive analisi e progettazioni (sarebbe ad esempio possibile a questo punto introdurre il concetto di *semaforo* [22] come elemento progettuale).

A questo punto il modello del sistema contiene tutte le informazioni necessarie ad inferire automaticamente il codice che rappresenta il prodotto finale. In alternativa, un tale modello di progetto può in linea di principio, essere eseguito direttamente, tramite un opportuno interprete per la notazione Concepts Z. Tuttavia prima di poter “eseguire” la specifica o “compilarla” producendone una riscrittura nelle frasi di un linguaggio di programmazione scelto, occorre fare alcune precisazioni sulla semantica della notazione, che sono oggetto del prossimo capitolo.

6
Semantica della notazione Concepts Z

6.1 Obiettivi

In questo capitolo si intende caratterizzare meglio la semantica della notazione Concepts Z e delineare la struttura generale adottata per giungere alla sua definizione.
Come si vedrà infatti, i linguaggi definiti a partire da teorie di base molto generali e di alto livello, come la teoria degli insiemi, presentano alcune particolarità.

6.2 Precisazioni sulla semantica di Z

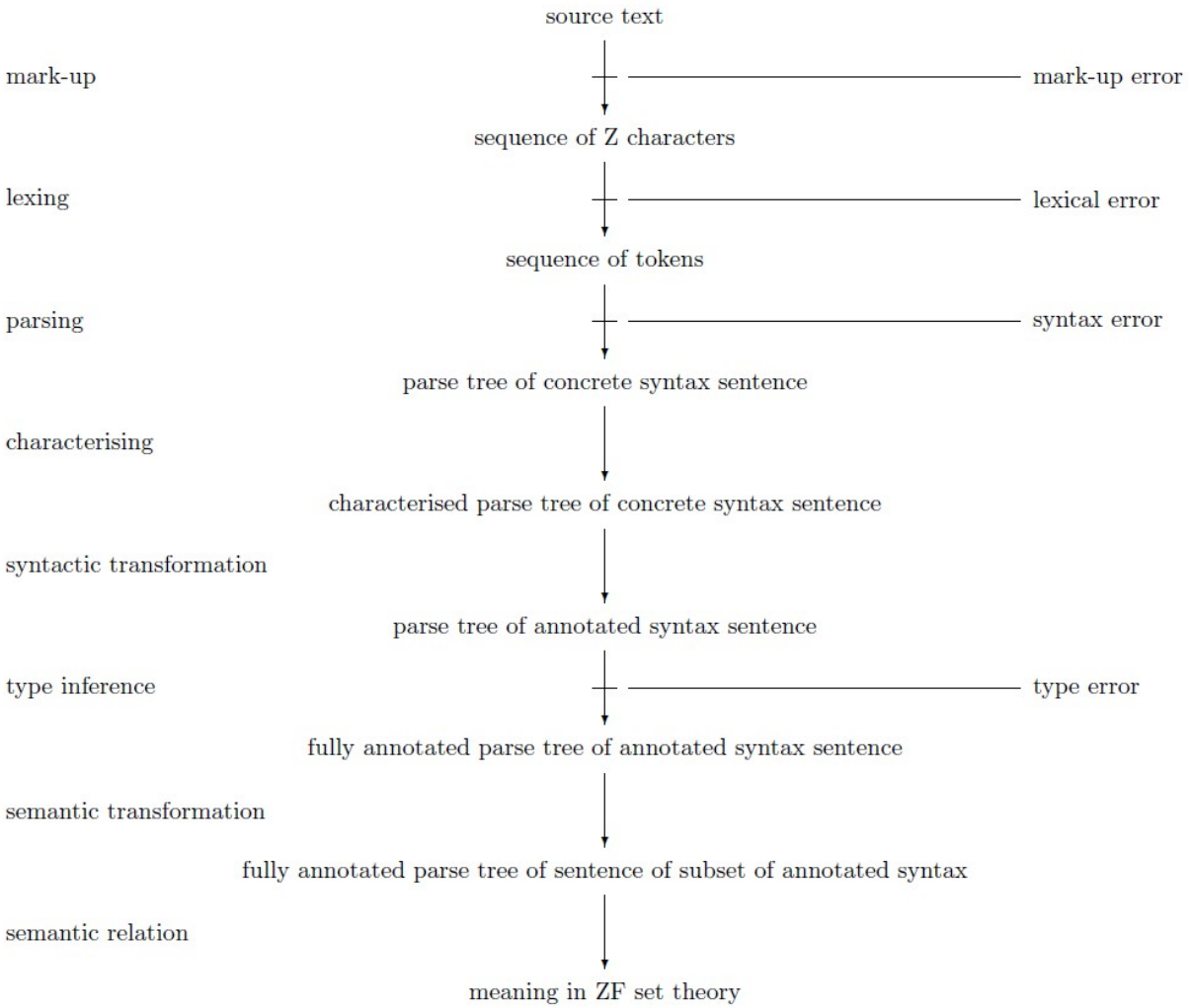
Come è noto sono possibili diversi approcci per la definizione della semantica di un linguaggio.

- *semantica operazionale*: attribuzione di un significato alle frasi del linguaggio tramite le operazioni elementari di una macchina di funzionamento noto. Come si è sottolineato nell'introduzione, questo approccio rende la semantica del linguaggio dipendente dalla macchina adottata.
- *semantica denotazionale*: attribuzione di un significato alle frasi di un linguaggio tramite un insieme di funzioni di valutazione. Ogni funzione associa ad un sottoinsieme dei costrutti del linguaggio (il suo universo di partenza), i corrispondenti elementi in un dominio di interpretazione (il suo universo di arrivo), tipicamente rappresentato sotto forma di entità matematiche. Una funzione di valutazione può essere espressa in termini di altre funzioni di valutazione in modo gerarchico. Tale semantica è più astratta della precedente poiché non impone la dipendenza da una particolare macchina di riferimento.
- *semantica assiomatica*: attribuzione di un significato alle frasi di un linguaggio attraverso assiomi e regole di inferenza della logica simbolica. Tale semantica è più

astratta delle precedenti, ma le proprietà che è in grado di dimostrare per un dato testo scritto in accordo alle regole di un linguaggio, possono non essere sufficienti a determinare completamente il suo significato, a causa della semi-decidibilità della logica del primo ordine.

Si intende inoltre con *decidibile* un problema per il quale esiste una sequenza *finita* di passi per giungere alla sua soluzione.

La standardizzazione del linguaggio Z [3] definisce per quest'ultimo la semantica assiomatica in termini della teoria degli insiemi ZF.



Data la generalità del dominio di interpretazione, alcune frasi scritte in Z, pur avendo un preciso significato, non risultano eseguibili. Questo poiché è possibile denotare elementi il cui calcolo rappresenta un problema indecidibile.

Questo porta alla necessità di associare al linguaggio una semantica operazionale

corretta, in relazione alla sua semantica assiomatica, per poter “animare” le specifiche Z su un certo automa esecutore.

Tale semantica operazionale non è unica ed è possibile proporre diversi interpreti per il linguaggio Z con un differente livello di copertura, sofisticatezza ed efficienza.

Per *copertura* di un interprete si intende la porzione della grammatica del linguaggio che può essere trattata con successo dall'interprete.

Per *efficienza* di un interprete si intende la velocità con cui esso riesce ad ottenere un risultato, quando lo ottiene.

Per *sofisticatezza* di un interprete di intendono le sue proprietà di terminazione. Data una specifica, un interprete che cade in una computazione infinita durante la sua valutazione è meno sofisticato di uno che invece termina valutando la stessa specifica.

Un interprete si dice però *corretto* se ogni volta che esso termina, produce un risultato compatibile con quanto ci si aspetterebbe da considerazioni teoriche sulla teoria degli insiemi.

Si noti che l'interpretazione di una specifica potrebbe non terminare ed essere comunque corretta. Data la natura non eseguibile della teoria degli insiemi, vi può essere un trade-off fra le proprietà di terminazione dell'interprete e una adeguata copertura del linguaggio. Ad esempio un interprete che copre tutto il linguaggio ma cade sempre in una computazione infinita non produce alcun risultato ed è dunque corretto, ma è senza dubbio non sofisticato. Un altro interprete più sofisticato e comunque corretto, potrebbe “ogni tanto” produrre alcuni risultati parziali, il cui calcolo dei dettagli fallisce producendo una computazione infinita. Tuttavia ciò che viene reso visibile in un tempo finito, sarebbe comunque accettabile se consistente con la teoria degli insiemi.

Ci sono dunque “gradazioni” fra il non avere alcun risultato ed il risultato esatto (in generale non ottenibile) della teoria degli insiemi. In generale dunque un risultato calcolato da un interprete sarà meno *raffinato* del corrispondente risultato esatto. Per cui un interprete risulta corretto se le sue operazioni concrete con cui ottiene i risultati sono *raffinate* dalle corrispondenti operazioni della teoria degli insiemi di ZF [26].

Al problema dell'animazione delle specifiche Z vi sono stati molti approcci, i quali possono essere raccolti in due categorie. La prima, detta *generate and test*, parte dall'idea di generare tutti i possibili valori per gli insiemi corrispondenti agli elementi specificati e valutare in base a tali assegnamenti le proposizioni asserite. Come è possibile immaginare, questo approccio, nonostante sia in linea di principio in grado di gestire qualsiasi documento scritto in Z, diviene rapidamente impraticabile al crescere della

complessità della specifica. La seconda categoria, detta procedurale, consiste nel trasformare un documento di specifica, scrivendolo in una forma che sia direttamente traslabile nelle procedure di un linguaggio di programmazione. L'obiettivo è quello di ottenere un programma che implementa la specifica. Ad esempio in [27] viene mostrato come un sottoinsieme di Z si possa ritenere equivalente ad un sottoinsieme del linguaggio Prolog.

Va notato che molti di questi approcci cercano di animare una specifica, con l'intento di ottenere un elemento prototipale non rappresentante il sistema software di per sé, bensì orientato alla verifica interattiva assieme a chi commissiona i requisiti.

Un ulteriore limite di tali approcci è quello di cercare di attribuire un “significato” unico ai costrutti del linguaggio Z “così come sono”. E' infatti possibile mantenere la correttezza pur interpretando in maniera leggermente diversa lo stesso costrutto in base contesto in cui si trova. Ad esempio si noti come in Concepts Z il costrutto schema venga utilizzato per rappresentare concetti via via differenti in base al luogo in cui si trova e man mano che si procede attraverso le varie fasi del processo di sviluppo.

Per tali motivi in questa sede, pur ispirandosi alle idee della categoria procedurale, si adotterà un approccio differente.

6.3 Semantica assiomatica di Concepts Z

La semantica assiomatica di Concepts Z è basata per costruzione su quanto definito per il linguaggio Z.

Dagli esempi mostrati, dovrebbe a questo punto essere chiaro che *la semantica assiomatica della notazione Concepts Z è parzialmente specificata*.

In particolare la notazione non dice nulla riguardo ai *meccanismi* con cui avvengono gli eventi nel sistema.

Ad esempio quando una classe ne estende un'altra, si afferma che un evento che ne specializza un altro deve rispettare quanto affermato nella superclasse. Tuttavia non vi è alcuna informazione riguardo a ciò che accade quando si invoca un evento della superclasse su un'istanza di sottoclasse. In altre parole non vi è alcuna indicazione circa il funzionamento del polimorfismo, ma solo vincoli sul consistente stato finale delle estensioni coinvolte.

Allo stesso modo non vi è alcuna informazione circa la modalità con cui vengono

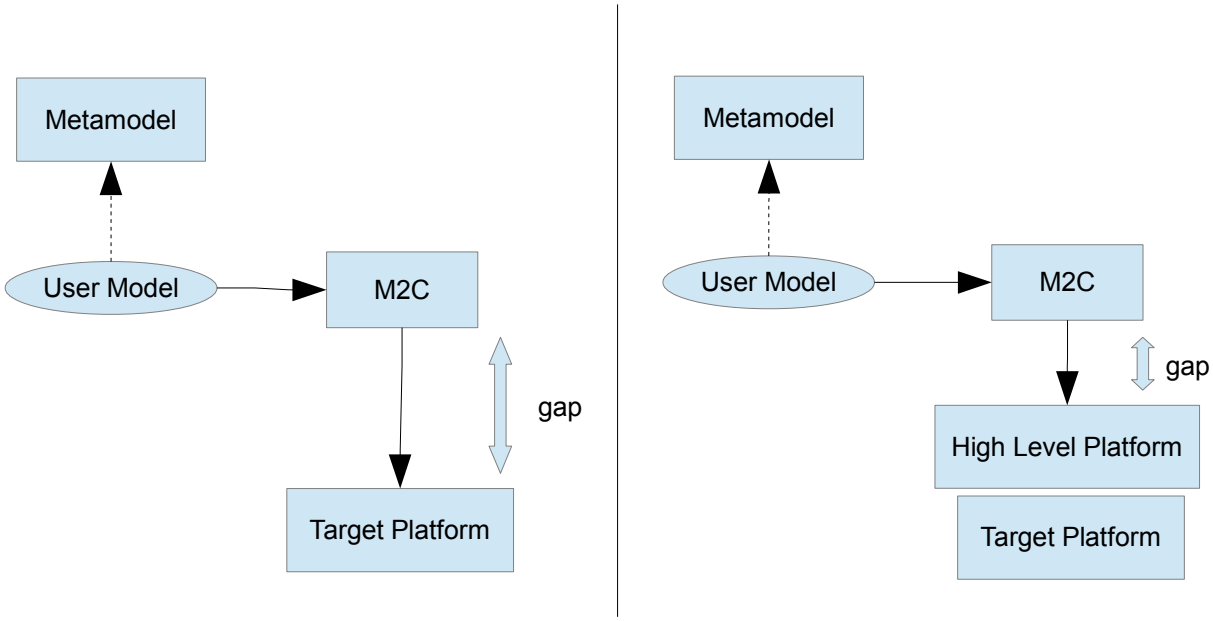
schedulate le attività definite.
Vi è invece un'alta precisione formale riguardo al significato delle informazioni che debbono essere inserite dagli addetti alla costruzione del sistema.
In questo senso dunque la semantica assiomatica viene utilizzata per fornire uno strumento di specifica non ambiguo delle informazioni necessarie riguardanti il sistema.
Note queste informazioni, la parte non specificata dalla semantica assiomatica, deve essere colmata associando una semantica operazionale alla notazione, consistente con la prima.
La semantica operazionale ha dunque l'obiettivo di chiarire il funzionamento di meccanismi assestati, che in quanto tali possono essere assunti noti da personale qualificato addetto alla costruzione del sistema e non necessitano di essere esplicitati formalmente.
Questo consente di delegare alle mosse elementari di una opportuna macchina virtuale la chiarificazione di dettagli implementativi non direttamente rilevanti per la costruzione del sistema.
Nel seguito della tesi si mostrerà dunque come sia possibile associare una semantica operazionale al linguaggio Concepts Z, la quale consente di definire completamente il suo significato.

6.4 Semantica Operazionale di Concepts Z

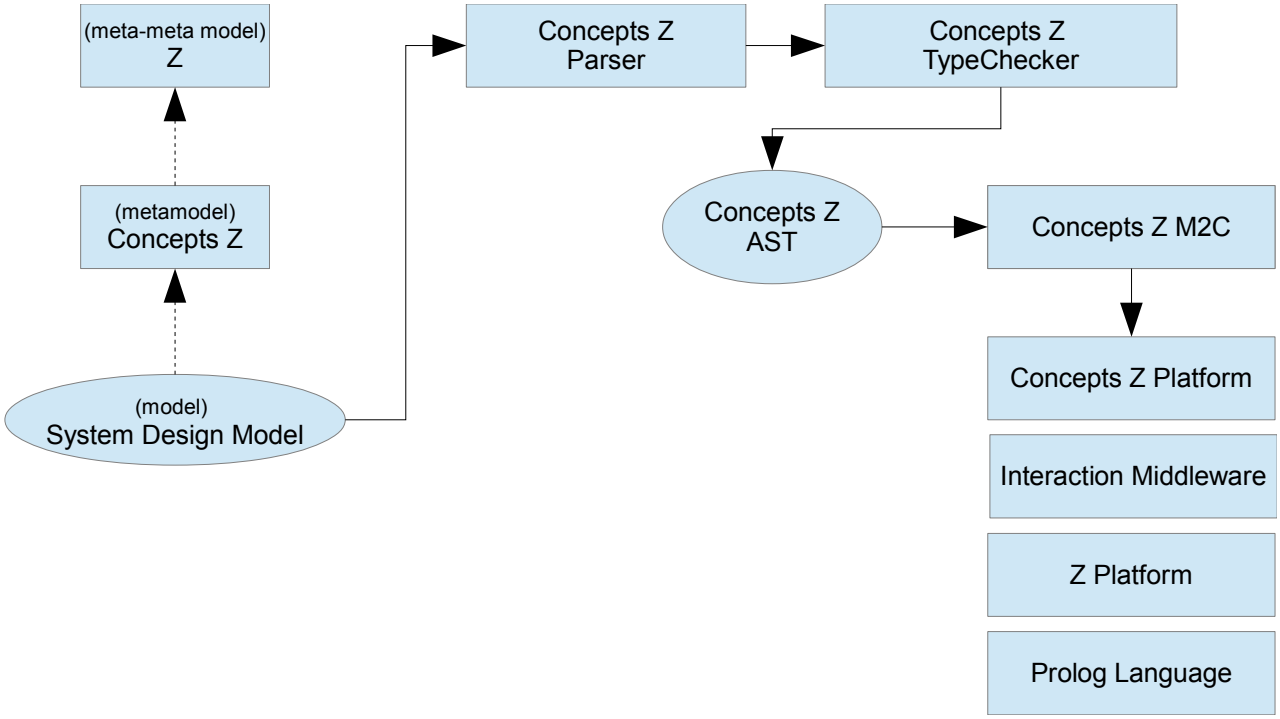
In questo paragrafo si introdurrà all'approccio scelto per dotare la notazione Concepts Z di una semantica operazionale consistente con la sua semantica assiomatica.
Si è detto nel capitolo 1 che l'obiettivo era quello di individuare un linguaggio di modellazione che consentisse l'applicazione pratica e generale dell'approccio MDSD.
In particolare si auspicava, al termine della fase progettuale, la generazione automatica del codice del sistema corrispondente al modello di progetto. Si noti come l'utilizzo di un generatore di codice, corrisponda a tutti gli effetti alla “compilazione” del modello di progetto verso una data piattaforma eseguibile.
Si è visto nei capitoli precedenti come sia possibile, tramite la notazione, produrre gradualmente un modello contenente le informazioni richieste a tal fine. Occorre ora individuare un approccio che consenta di tradurre il modello nelle frasi di un linguaggio di programmazione.

Poiché il linguaggio di specifica è una notazione basata sulla logica dei predicati del primo ordine, si sceglie come piattaforma esecutiva un linguaggio logico, il Prolog [28]. Questo consente di minimizzare il gap semantico fra i costrutti della notazione e quelli messi a disposizione dal linguaggio di programmazione.

In accordo all'approccio MDSD la notazione Concepts Z costituisce un metamodello, basato sul meta-meta modello Z. Un modello di sistema scritto in Concepts Z, una volta analizzato sintatticamente e semanticamente, diviene tipicamente un albero sintattico astratto (AST), basato sui concetti del metamodello e contenente tutte le informazioni essenziali presenti nella specifica. A questo punto un trasformatore M2C può elaborare l'AST generando il codice desiderato. Un trasformatore M2C è quindi un interprete la cui valutazione dell'AST corrisponde al codice del sistema specificato. Tuttavia, al fine di aumentare la leggibilità del codice prodotto e soprattutto semplificare la costruzione del generatore di codice, risulta conveniente la costruzione di una piattaforma che realizzi le metafore messe a disposizione dal linguaggio di specifica, riducendo in tal modo ulteriormente il gap semantico che il M2C deve colmare.



In particolare nel caso in esame occorre fornire i costrutti del linguaggio Z, le nozioni definite in Concepts Z ed un ambiente esecutivo che realizzi la macchina virtuale Concepts Z. L'architettura complessiva della nostra fabbrica del software è dunque la seguente:



Gli ellissi rappresentano elementi variabili nell'architettura mentre i rettangoli rappresentano componenti software.

In particolare il componente *Concepts Z M2C* esprime la semantica operazionale della notazione Concepts Z in termini della piattaforma esecutiva *Concepts Z Platform*.

Nel capitolo successivo sarà mostrata una possibile realizzazione di questa architettura.

7

Costruzione di una semantica operazionale per Concepts Z

7.1 Obiettivi

In questo capitolo si mostrerà come sia possibile implementare una semantica operazionale per il linguaggio di modellazione Concepts Z.
A questo fine, come spiegato, occorre colmare il gap semantico fra un modello di progetto espresso in Concepts Z e il linguaggio Prolog. E' dunque necessario costruire una serie di componenti che realizzino le varie metafore utilizzate dal linguaggio di specifica.
Nello specifico, occorre fornire componenti che realizzino le seguenti funzionalità

- realizzazione dei costrutti eseguibili di base del linguaggio Z e del suo standard toolkit
- realizzazione del TreeToolkit
- realizzazione del middleware per le interazioni presenti in Concepts Z
- realizzazione della macchina astratta Concepts Z
- realizzazione del generatore di codice di Concepts Z

Occorrono inoltre componenti che effettuino il parsing ed il typechecking della notazione Concepts Z.
Si sottolinea che i componenti che realizzano il linguaggio Concepts Z *non* necessitano del (e dunque non supportano pienamente il) sistema di tipi introdotto dal linguaggio Z, perché il loro obiettivo è unicamente quello di fornire un back-end che costituisca una base eseguibile per il linguaggio. Il sistema dei tipi di Z rappresenta un importante riferimento logico per la costruzione di una specifica, ma a valle delle fasi di parsing e typechecking esaurisce la sua utilità.
Nel resto del capitolo si assumerà che il lettore abbia familiarità con il linguaggio Prolog, la

piattaforma Java e il framework Xpand. Per una trattazione approfondita di queste tecnologie è possibile consultare [28,29,30,33].

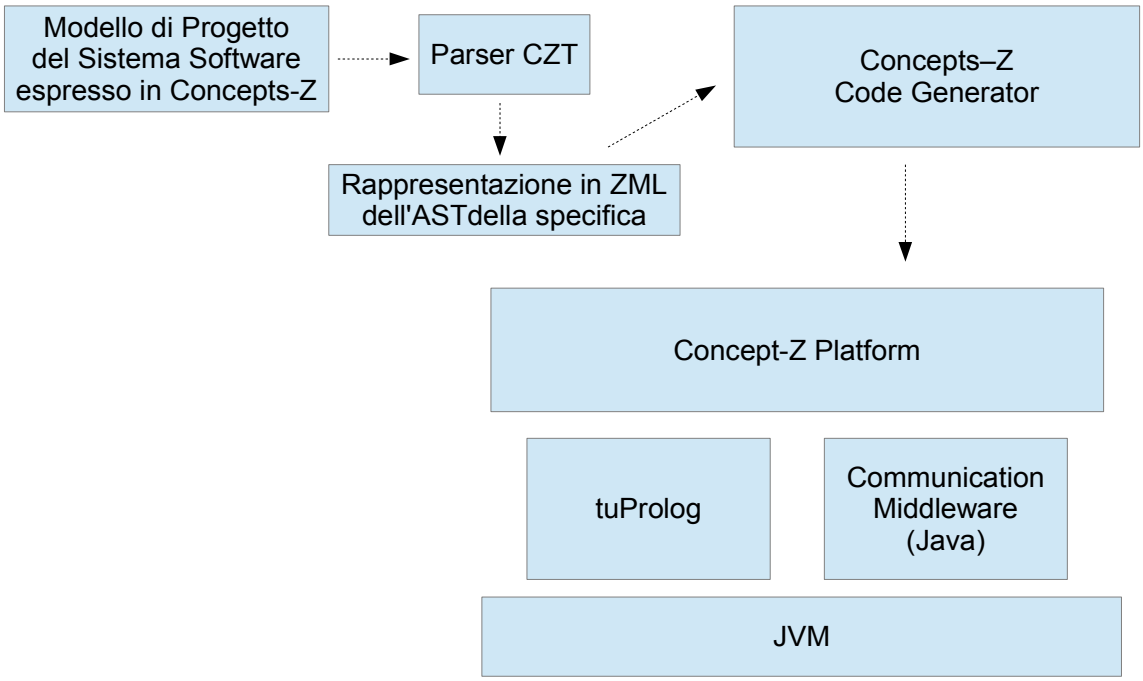
7.2 Selezione delle Tecnologie

Come implementazione del linguaggio Prolog si sceglie l'interprete tuProlog [31]. Si tratta di un motore Prolog scritto in Java e dunque in grado di sfruttare tutte le funzionalità di quest'ultimo. L'interprete tuProlog è inoltre in grado di gestire più processi di risoluzione concorrenti all'interno dello stesso motore. Data la naturale compatibilità di tuProlog con il linguaggio Java, alcuni componenti, per ragioni di efficienza e convenienza realizzativa, verrano implementati direttamente nel linguaggio Java e utilizzati attraverso l'interprete tuProlog.

Poiché Concepts Z è una notazione basata sul linguaggio Z, è possibile appoggiarsi ad un riconoscitore per quest'ultimo linguaggio per l'analisi sintattica e semantica delle specifiche. Dunque si sceglie di utilizzare il framework CZT [5,32], una serie di strumenti per il parsing, typechecking e la trasformazione di specifiche scritte in Z. In particolare, a valle dell'analisi sintattico-semantica, il tool CZT consente fra le altre cose, di esportare le specifiche Z sotto forma di un mark-up XML denominato ZML[32].

Per la costruzione del generatore di codice della notazione Concepts Z, si sceglie il framework Xpand parte di EMF[33].

La figura sottostante mostra l'architettura della fabbrica del software per Concepts Z, in relazione alle tecnologie selezionate.



Come è possibile notare il modello di progetto verrà elaborato dal parser CZT e successivamente tradotto in formato ZML per essere elaborato dal framework Xpand, tramite cui sarà espresso il generatore di codice. L'output del generatore sarà codice Prolog che utilizzerà le funzionalità della piattaforma di esecuzione per Concepts Z.

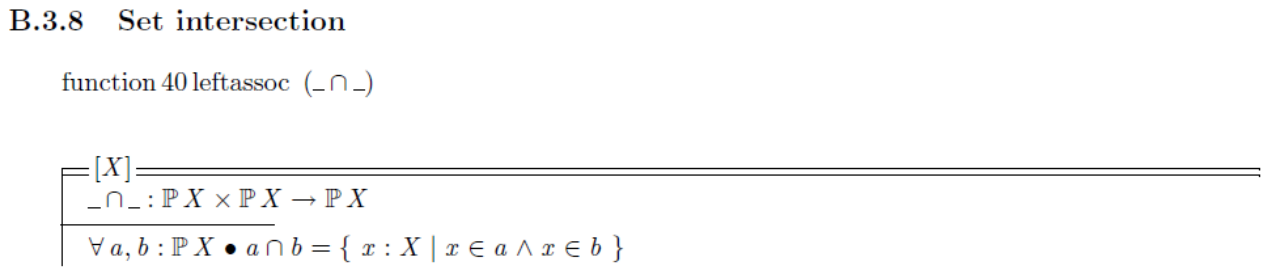
7.3 Realizzazione dei costrutti del linguaggio Z

Poiché la notazione Concepts Z si appoggia sul meccanismo espressivo del linguaggio Z, occorre fornire una implementazione dei costrutti principali di tale linguaggio.
In particolare una prima problematica riguarda la rappresentazione degli insiemi nel linguaggio Prolog.
Come detto più volte il linguaggio Z utilizza la semantica per valore e l'applicazione di un operatore ad un insieme produce un nuovo insieme distinto da quello di partenza. Per tale ragione la rappresentazione degli insiemi più efficiente e compatibile con la semantica di Z è quella di una *lista ordinata*. In particolare si sfrutta l'ordinamento totale dei termini definito in Prolog @< . *Un insieme è dunque una lista ordinata di valori senza ripetizioni*. Si noti come tale scelta consenta di sfruttare l'usuale meccanismo di unificazione del Prolog per realizzare l'uguaglianza fra insiemi di valori di qualunque tipo.
L'altro tipo di dato principale in Z è la n-upla. In accordo alla definizione data dal linguaggio Z[3], si sceglie di rappresentare una n-upla come una lista di valori di almeno due

elementi.
Si noti che nonostante il back-end non necessiti del sistema di tipi, in Z è possibile scrivere predicati che asseriscono l'appartenenza di un valore ad un certo tipo (ad esempio appartenenza alle sequenze iniettive) ed esprimere vincoli nella parte dichiarativa. Per supportare questo tipo di frasi è necessario prevedere degli ulteriori predicati Prolog in grado di strabilire se un certo valore sia o meno conforme ad un dato tipo.

I predicati che realizzano i costrutti standard di Z possiedono il corrispondente nome testuale attribuito ad essi dallo standard [3] e ne rispettano l'interfaccia specificata. In particolare le variabili di un predicato Prolog da sinistra a destra corrispondono nell'ordine agli argomenti specificati nello standard per gli operatori.

Ad esempio per l'intersezione fra insiemi nello standard [3] si ha la seguente definizione:



B.3.8 Set intersection

function 40 leftassoc $(\_ \cap \_)$

$$[X]$$
$$\_ \cap \_ : \mathbb{P}\, X \times \mathbb{P}\, X \to \mathbb{P}\, X$$
$$\forall\, a, b : \mathbb{P}\, X \bullet a \cap b = \{\, x : X \mid x \in a \land x \in b \,\}$$

The intersection of two sets of the same type is the set of values that are members of both sets.

a questa funzione corrisponderà perciò il predicato Prolog

```
intersection( S1 , S2 , SR )
```

dove le prime due variabili S1 ed S2 corrispondono nell'ordine ai parametri di ingresso della funzione e la variabile SR verrà legata all'insieme che costituisce il risultato dell'intersezione di S1 ed S2, ossia al valore associato alla coppia di ingresso dalla funzione.
Si noti che in questo contesto non ha alcun senso parlare di reversibilità del predicato intersection, in quanto la sua specifica dice espressamente che si tratta di una funzione che associa a due parametri di ingresso una uscita: per questo nella implementazione dei predicati che realizzano i costrutti desiderati si privilegia l'efficienza computazionale.

Allo stesso modo non vi è motivo di introdurre nella implementazione dei costrutti eccessivi controlli sulle loro relative precondizioni, in quanto il codice che farà uso di tali predicati sarà il risultato di una generazione automatica a valle di controlli sintattici e di tipo. Si può dunque supporre un uso “semanticamente corretto” dei predicati.

A titolo di esempio si mostrerà ora il codice Prolog che realizza gli insiemi,le tuple ed i primi predicati che implementano i corrispondenti costrutti di base del linguaggio Z.

```
% definisce l'ordinamento totale fra termini utilizzato

lessThan( X , Y ) :- X @< Y .


% elimina le ripetizioni da una lista ordinata

makeSet( [] , [] ) .

makeSet( [H,H|T] , X ) :- ! ,
                          makeSet( [H|T] , X ) .

makeSet( [H|T] , [H|X] ) :- makeSet( T , X ) .


% un insieme è una lista ordinata di valori senza ripetizioni

set( [] ) .

set( [H|T] ) :- ground( [H|T] ) ,
                quicksort( [H|T] , lessThan , [H|T] ) ,
                makeSet( [H|T] , [H|T] ) .


setExtension( [] , [] ) .

setExtension( [H|T] , [Hs|Ts] ) :- ground( [H|T] ) ,
                                   quicksort( [H|T] , lessThan , L ) ,
                                   makeSet( L , [Hs|Ts] ) .


% una tupla è una lista di valori di almeno due elementi

tuple( [C1,C2|T] ) :- ground( [C1,C2|T] ) .


tupleExtension( [C1,C2|T] , [C1,C2|T] ) :- ground( [C1,C2|T] ) .


tupleSelection( [H|T] , I , Component ) :- element( I , [H|T] , Component ) .


equality( X , X ) .
```

```
membership( E , [E|_] ) :- ! .

membership( E , [X|T] ) :- lessThan( X , E ) ,
                           membership( E , T ) .


cardinality( S , N ) :- length( S , N ) .


inequality( X , Y ) :- X \= Y .


nonMembership( E , S ) :- ground( E ) ,
                          ground( S ) ,
                          ! ,
                          not( membership( E , S ) ) .

nonMembership( _ , _ ) :- throw( error(instantiation_error,'nonMembership arguments
must be ground') ) .


emptySet( [] ) .


subset( [] , _ ) :- ! .

subset( S , S ) :- ! .

subset( [H|Ts1] , [H|Ts2] ) :- ! ,
                               subset( Ts1 , Ts2 ) .

subset( [Hs1|Ts1] , [Hs2|Ts2] ) :- lessThan( Hs2 , Hs1 ) ,
                                   subset( [Hs1|Ts1] , Ts2 ) .


properSubset( S1 , S2 ) :- inequality( S1 , S2 ) ,
                           subset( S1 , S2 ) .


nonEmptySet( [H|T] ) :- set( [H|T] ) .


union( S1 , [] , S1 ) :- ! .

union( [] , S2 , S2 ) :- ! .

union( S , S , S ) :- ! .

union( [H|Ts1] , [H|Ts2] , [H|X] ) :- ! ,
                                      union( Ts1 , Ts2 , X ) .

union( [Hs1|Ts1] , [Hs2|Ts2] , [Hs1|X] ) :- lessThan( Hs1 , Hs2 ) ,
                                            ! ,
                                            union( Ts1 , [Hs2|Ts2] , X ) .

union( [Hs1|Ts1] , [Hs2|Ts2] , [Hs2|X] ) :- union( [Hs1|Ts1] , Ts2 , X ) .

      . . .
```

E' attualmente disponibile un'implementazione in Prolog dei costrutti di base del Linguaggio Z, compresi gli operatori proposizionali, i quantificatori e le set comprehension. Si dispone inoltre di un'implementazione per l'intero standard-toolkit. In particolare per quest'ultimo gli unici operatori non implementati sono l'operatore _* (chiusura transitiva riflessiva) e l'operatore iter n, che risultano in generale non computabili.

7.4 Realizzazione del TreeToolkit

Occorre fornire un'implementazione anche per l'estensione allo standard toolkit di Z specificata nel paragrafo 5.2 che definisce gli alberi. Coerentemente con la rappresentazione degli altri costrutti, un albero è una lista ordinata avente un unico elemento, una coppia in cui il primo elemento è un nodo ed il secondo è l'insieme dei sottoalberi figli. Tale scelta di rappresentazione consente di poter applicare sulle parti dell'albero gli operatori definiti nelle altre parti dello standard toolkit. La scelta di una rappresentazione ricorsiva per la struttura dati consente inoltre di semplificare notevolmente l'implementazione delle funzioni che operano sugli alberi.

La direttiva `:- parents( fileName )` consente ad una teoria logica Prolog di includere le asserzioni presenti in un'altra teoria contenuta nel file fileName, qualora quest'ultima teoria non sia già stata caricata in precedenza.

```
:- parents( 'ZLibrary/SequenceToolkit.pl' ) .


graph( [[],[]] ) :- ! .

graph( [Nodes,Arcs] ) :- set( Nodes ) ,
                         relation( Arcs ) ,
                         domain( Arcs , DA ) ,
                         subset( DA , Nodes ) ,
                         range( Arcs , RA ) ,
                         subset( RA , Nodes ) .


inArcs( [Nodes,Arcs] , N , ArcSet ) :- membership( N , Nodes ) ,
                                       ! ,
                                       rangeRestriction( Arcs , [N] , ArcSet ) .

inArcs( _ , _ , _ ) :- throw( error(domain_error,'provided node must be a member of the
graph') ) .
```

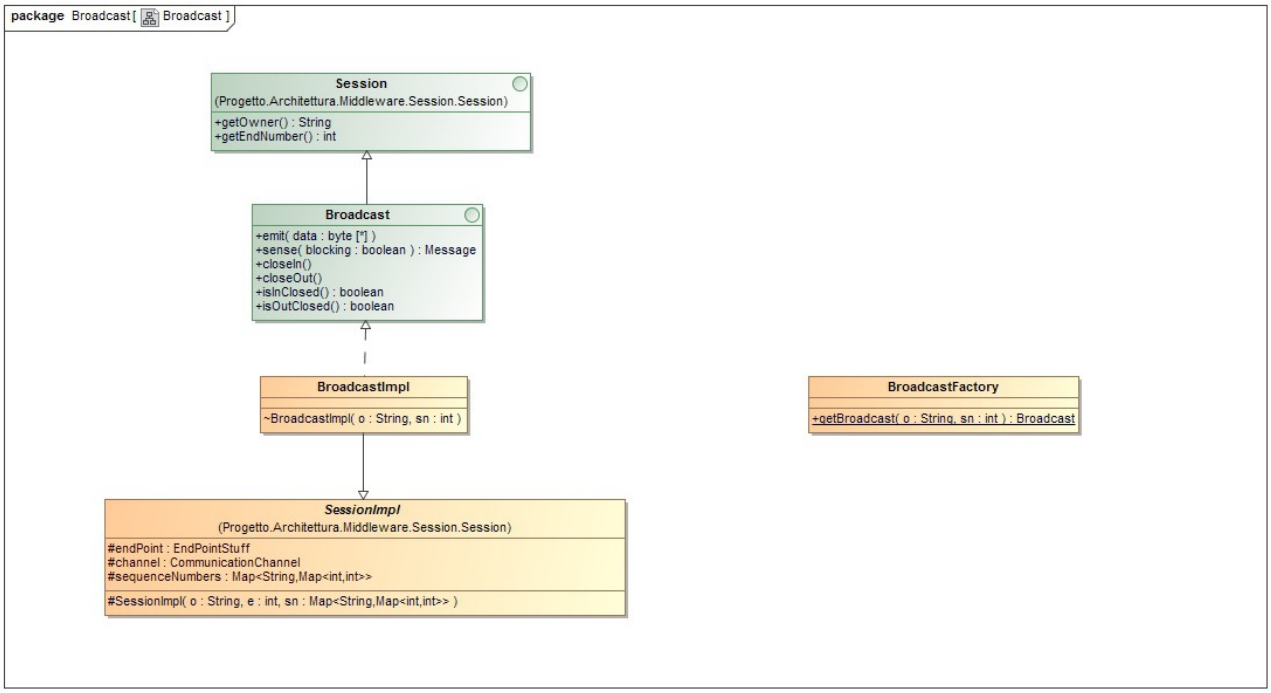
```
outArcs( [Nodes,Arcs] , N , ArcSet ) :- membership( N , Nodes ) ,
                                        ! ,
                                        domainRestriction( [N] , Arcs , ArcSet ) .

outArcs( _ , _ , _ ) :- throw( error(domain_error,'provided node must be a member of
the graph') ) .


treeRep( [] ) .

treeRep( [[N,S]] ) :- ground( N ) ,
                      set( S ) ,
                      treeReps( S ) .

treeReps( [] ) .

treeReps( [[X]] ) :- treeRep( [X] ) .

treeReps( [[[R1,CL1]],[[R2,CL2]]|T] ) :- R1 \= R2 ,
                                         treeRep( [[R1,CL1]] ) ,
                                         treeReps( [[[R2,CL2]]|T] ) .


nodes( [] , [] ) .

nodes( [[N,CL]] , S ) :- nodesRest( CL , [N] , S ) .

nodesRest( [] , Result , Result ) .

nodesRest( [[H]|T] , Acc , Result ) :- nodes( [H] , S1 ) ,
                                       union( S1 , Acc , NewAcc ) ,
                                       nodesRest( T , NewAcc , Result ) .


arcs( [] , [] ) .

arcs( [[N,CL]] , Arcs ) :- arcsRest( CL , N , [] , Arcs ) .

arcsRest( [] , _ , Result , Result ) .

arcsRest( [[[HN,HCL]]|T] , P , Acc , Result ) :- arcs( [[HN,HCL]] , AH ) ,
                                                 union( AH , [[P,HN]] , AHF ) ,
                                                 union( AHF , Acc , NewAcc ) ,
                                                 arcsRest( T , P , NewAcc , Result ) .


tree( [] ) .

tree( [[N,CL]] ) :- treeRep( [[N,CL]] ) ,
                    nodes( [[N,CL]] , Nodes ) ,
                    arcs( [[N,CL]] , Arcs ) ,
                    transitiveClosure( Arcs , TC ) ,
                    identityRelation( Nodes , IDN ) ,
                    intersection( TC , IDN , [] ) ,
                    inArcs( [Nodes,Arcs] , N , [] ) ,
                    difference( Nodes , [N] , NoRadix ) ,
                    setComprehension( [[Ns,Nodes]] , inArcs([Nodes,Arcs],Ns,[X]) ,
NoRadix ) .
```

```
nonEmptyTree( [[N,CL]] ) :- tree( [[N,CL]] ) .


root( [[N,_]] , N ) .

% poiché l'insieme dei sottoalberi figli è una lista ordinata, i sottoalberi figli sono
ordinati in base alla loro radice dalla relazione @<

contains( [[E,_]] , E ) :- ! .

contains( [[_,CL]] , E ) :- containsSubtree( CL , E ) .

containsSubtree( [[[HN,HCL]]|_] , E ) :- contains( [[HN,HCL]] , E ) ,
                                         ! .

containsSubtree( [[[_,_]]|T] , E ) :- containsSubtree( T , E ) .

      . . .
```

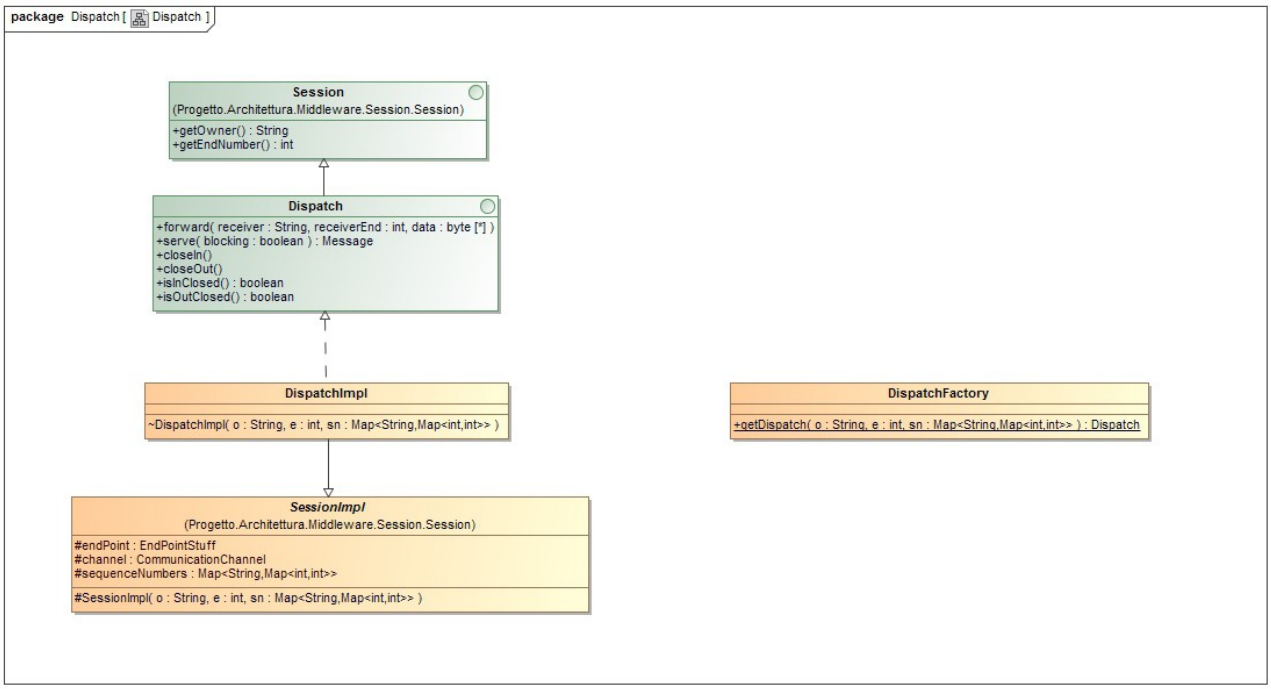
Ad esempio si noti come il predicato contains realizzi una ricerca left-first in un generico albero.
Al momento della scrittura della tesi si dispone di un'implementazione completa per il TreeToolkit specificato nel paragrafo 5.2 .

7.5 Realizzazione del middleware per le interazioni

Occorre fornire una implemetazione per le metafore relative alla comunicazione distribuita definite nel capitolo 4. Si ricorda che il middleware per le interazioni ha in questo contesto la funzione di innalzare il livello della comunicazione fra due entità, non di realizzare la comunicazione fisica, delegata a protocolli di comunicazione.
Si tratta di un componente molto generale, non necessariamente legato al linguaggio Concepts Z, che necessita di una gestione efficiente della memoria impiegata per i messaggi e le attività di marshalling/unmarshalling verso/da una rappresentazione esterna per essi [23]. Per tali motivi si sceglie di implementare direttamente in Java questo componente. A titolo di esempio si mostra una possibile organizzazione per le interfacce concrete corrispondenti ai blocchi specificati nel capitolo 4.



Si noti che un middleware è tipicamente utilizzato da molte attività differenti contemporaneamente e dunque necessita di una gestione molto sofisticata della concorrenza. Al momento della scrittura della tesi si dispone di un prototipo funzionante del middleware, in grado di effettuare due delle tre interazioni specificate (Broadcast e Dispatch) e la realizzazione della terza modalità di interazione è in fase di completamento.



Come è possibile notare le interfacce concrete degli oggetti Sessione non espongono direttamente le variabili condizione all'esterno bensì consentono all'attività invocante una chiamata potenzialmente bloccante (tipicamente la ricezione), di esplicitare la volontà di bloccarsi o meno su di essa.

7.6 Una macchina astratta per Concepts Z

Basandosi sulle implementazioni fornite per i costrutti di Z e per le interazioni, occorre ora costruire una macchina virtuale le cui mosse elementari consentano di associare una semantica operazionale alla notazione Concepts Z.
In particolare tale macchina virtuale deve:

- fornire un supporto agli oggetti ed al concetto di ereditarietà
- implementare il polimorfismo
- chiarire la gestione delle attività

Per quanto riguarda la gestione delle attività si rimanda alla gestione del threading all'interno dell'interprete tuProlog [31]. In questa sede si approfondiranno invece i primi due punti, più problematici dal punto di vista realizzativo vista l'assenza del concetto di oggetto in Prolog.

7.6.1 Il supporto alla object-orientation

La mancanza di un supporto object oriented all'interno del Prolog implica la necessita di rappresentare classi ed oggetti sotto forma di termini. Per farlo si può prendere spunto dalle definizioni formali espresse nella notazione e dalla sua organizzazione. Tuttavia poiché quello che si sta realizzando è un back-end di esecuzione per Concepts Z, esso deve essere il più possibile efficiente. Si sceglie dunque di tenere traccia delle sole estensioni delle classi concrete, a differenza di quanto avviene nella notazione.
Si noti che la realizzazione di un supporto a oggetti all'interno del Prolog implica la spinosa problematica di *implementare una semantica per riferimento all'interno di un linguaggio dichiarativo con semantica per valore*. Nel linguaggio Prolog infatti le variabili, una volta

legate ad un certo valore, mantengono tale legame sino al termine del processo di risoluzione o finché non si innesca un backtracking su un'alternativa cronologicamente precedente al legame stabilito. Una variabile ed il suo legame sono inoltre tipicamente visibili esclusivamente entro il loro processo di risoluzione. Il paradigma ad oggetti adotta invece una semantica per riferimento: lo stato degli oggetti è ospitato in un'area di memoria a cui è possibile accedere dereferenziando opportunamente puntatori ad essi. Una volta invocato un metodo che modifica lo stato di un oggetto, i successivi accessi allo stesso oggetto vedono tale modifica e dunque operano su variabili di istanza il cui valore è il risultato di tutte le manipolazioni precedentemente effettuate sull'oggetto.
La necessità di garantire un tale comportamento in presenza di accessi concorrenti all'oggetto ed un'evoluzione del loro stato consistente con le relazioni di ereditarietà fra oggetti complicano le cose. Occorre inoltre, all'atto dell'invocazione di un operazione su un oggetto, fornire un'implementazione per il polimorfismo tipico dei linguaggi a oggetti.

Si discuterà ora una possibile soluzione per fronteggiare queste problematiche.

7.6.1.1 Elementi preliminari della notazione

Si inizia con l'introduzione di alcuni elementi corrispondenti alle definizioni presenti nel paragrafo 4.2 .
Poiché come detto la scelta di un linguaggio logico riduce notevolmente il gap semantico per un linguaggio di specifica basato sulla logica dei predicati, molti elementi presenti nella notazione sono già disponibili in modo nativo all'interno del linguaggio Prolog o disponibili come predicati di libreria dell'interprete tuProlog. In tale sede occorre semplicemente fornire una rappresentazione per il tipo booleano.

```
% BaseConcepts.pl

:- parents( 'ZLibrary/SequenceToolkit.pl' ) .

boolean( t ) :- ! .

boolean( f ) .
```

7.6.1.2 Elementi di analisi della notazione

```
% AnalysisConcepts.pl

:- parents( 'ConceptsZLibrary/BaseConcepts.pl' ) .
```

Si sceglie di rappresentare ogni classe dichiarata nella specifica tramite un fatto Prolog del tipo `class( className )` , dove className è il nome attribuito alla classe con l'iniziale minuscola.
Allo stesso modo si asserisce che due classi sono in relazione di ereditarietà tramite un fatto Prolog del tipo `isA( subClass , superClass )` , dove subClass e superClass sono gli atomi corrispondenti ai nomi di tali classi.
Con queste premesse è possibile definire un predicato Prolog in grado di stabilire se due classi sono in relazione di ereditarietà. Il predicato subclass(C2 , C1) ha successo se C2 è sottoclasse di C1 (e dunque rappresenta una porzione della sua estensione).

```
subclass( C2 , C1 ) :- class( C2 ) ,
                       class( C1 ) ,
                       subclassRest( C2 , C1 ) .

subclassRest( C2 , C1 ) :- isA( C2 , C1 ) ,
                           ! .

subclassRest( C2 , C1 ) :- isA( C2 , X ) ,
                           subclass( X , C1 ) .
```

Oltre alle classi occorre rappresentare i riferimenti agli oggetti loro istanze.
Si sceglie di rappresentare un riferimento ad un oggetto tramite il termine `reference( id , className )` . In tale termine l'atomo id rappresenta un intero progressivamente aumentato alla creazione di ogni nuova istanza di className.
Ovviamente esisteranno riferimenti per le sole istanze di classi concrete. Dunque per ogni classe non astratta si tiene traccia del primo intero disponibile *n* attraverso il termine `referenceCount( className , n )` .
Se invece una classe è dichiarata astratta, quest'ultimo termine è sostituito dall'asserzione `abstract( className )` .
Quanto detto implica la possibilità di definire un predicato Prolog che verifichi se un riferimento rispetta la struttura stabilita.

```
wellFormedReference( reference(ID,C) ) :- class( C ) ,
                                          not( abstract( C ) ) ,
                                          integer( ID ) ,
                                          ! .
```

E' inoltre possibile a questo punto stabilire se un'istanza appartiene all'estensione di una data classe o se è istanza diretta di tale classe.
Il predicato `extension( C , R )` ha successo se l'istanza associata al riferimento R fa parte dell'estensione della classe C.

```
extension( C , R ) :- class( C ) ,
                      wellFormedReference( R ) ,
                      extensionRest( C , R ) .

extensionRest( C , reference(_,C) ) :- ! .

extensionRest( C1 , reference(_,C2) ) :- subclass( C2 , C1 ) .
```

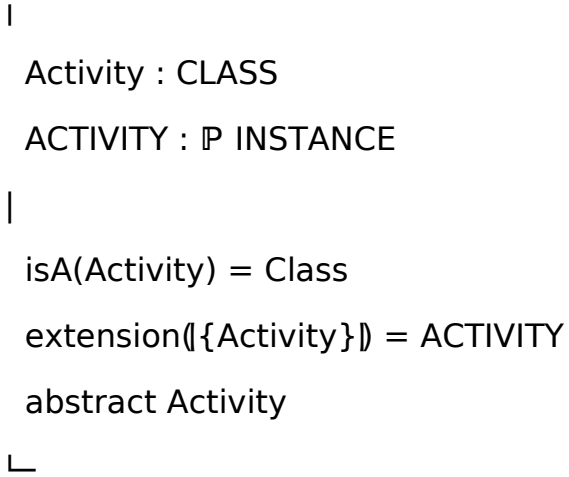
Il predicato `direct( C , R )` ha successo se R rappresenta un'istanza diretta della classe C.

```
direct( C , reference(ID,C) ) :- wellFormedReference( reference(ID,C) ) .
```

Quanto detto consente di asserire le prime informazioni riguardanti la classe radice Class:

```
class( class ) .
abstract( class ) .
```

A titolo di esempio il codice corrispondente alle due dichiarazioni seguenti nella specifica in Concepts Z



```
|
  Condition : CLASS
  CONDITION : ℙ INSTANCE
|
  isA(Condition) = Class
  extension(⦇{Condition}⦈) = CONDITION
  ¬ abstract Condition
└_
```

risulta essere

```
class( activity ) .
isA( activity , class ) .
abstract( activity ) .

class( condition ) .
isA( condition , class ) .
referenceCount( condition , 0 ) .
```

7.6.1.3 Elementi di progetto della notazione

```
% DesignConcepts.pl

:- parents( 'ConceptsZLibrary/AnalysisConcepts.pl' ) .

:- parents( 'ZLibrary/TreeToolkit.pl' ) .
```

Si sceglie di rappresentare un'interfaccia tramite asserzioni del tipo `interface( interfaceName )` , dove interfaceName è l'atomo che rappresenta l'interfaccia dichiarata.
La relazione di estensione fra due interfacce è rappresentata dal termine `extends( iSub , iSuper )` , dove iSub rappresenta il nome dell'interfaccia che estende iSuper.
La relazione implements è infine rappresentata da termini del tipo `implements( cName , iName )` , dove cName è il nome della classe che implementa l'interfaccia di nome iName.

E' possibile definire un predicato `subInterface( I2 , I1 )` che sfrutta tali asserzioni per stabilire se un'interfaccia I2 estende un'interfaccia I1:

```
subInterface( I2 , I1 ) :- interface( I2 ) ,
                           interface( I1 ) ,
                           subInterfaceRest( I2 , I1 ) .

subInterfaceRest( I2 , I1 ) :- extends( I2 , I1 ) ,
                               ! .

subInterfaceRest( I2 , I1 ) :- extends( I2 , X ) ,
                               subInterface( X , I1 ) .
```

Si può inoltre definire un predicato `implementationOf( C , I )` che ha successo se la classe C realizza l'interfaccia I:

```
implementationOf( C , I ) :- class( C ) ,
                             interface( I ) ,
                             implementationOfRest( C , I ) .

implementationOfRest( C , I ) :- implements( C , I ) ,
                                 ! .

implementationOfRest( C , I ) :- implements( C , X ) ,
                                 subInterface( X , I ) ,
                                 ! .

implementationOfRest( C , I ) :- isA( C , X ) ,
                                 implementationOf( X , I ) .
```

A questo punto diviene possibile dare delle implementazioni per gli operatori iExtension e iDirect definiti in Concepts Z.
Il predicato `iExtension ( I , R )` ha successo se R appartiene all'estensione di I.

```
iExtension( I , reference(ID,C) ) :- interface( I ) ,
                                     wellFormedReference( reference(ID,C) ) ,
                                     implementationOf( C , I ) .
```

Il predicato `iDirect( I , R )` ha successo se R è un'istanza diretta dell'interfaccia I e non di una sua sottointerfaccia.

```
iDirect( I , reference(ID,C) ) :- interface( I ) ,
                                  wellFormedReference( reference(ID,C) ) ,
                                  iDirectRest( I , reference(ID,C) ) .

iDirectRest( I , reference(_,C) ) :- implements( C , I ) ,
                                     ! .
```

```
iDirectRest( I , reference(_,C) ) :- isA( C , X ) ,
                                     iDirectRest( I , reference(_,X) ) .
```

7.6.1.4 Implementazione degli oggetti

Come si è sottolineato in 7.6.1 la maggiore problematica riguardante la realizzazione del paradigma ad oggetti nel linguaggio Prolog consiste nell'implementare in esso una semantica per riferimento riguardo agli oggetti stessi e alla loro manipolazione.
Una prima idea potrebbe essere quella di sfruttare la base di conoscenza ove vengono asserite le regole di un programma Prolog come un “heap” in cui ospitare gli oggetti.
La base di conoscenza di un motore Prolog costituisce in effetti uno spazio di memoria condiviso da tutti i processi di risoluzione concorrenti operanti all'interno di esso.
Il sistema software sarebbe dunque rappresentato da un programma dinamico, il quale aggiorna la propria base di conoscenza man mano che avvengono eventi sugli oggetti, apportando modifiche al loro stato.
Questa scelta non è tuttavia esente da problematiche. La modifica dello stato di un oggetto, rappresentato come termine, asserito nel database Prolog, richiede infatti dapprima la ritrattazione del vecchio termine dalla base di conoscenza ed in seguito l'asserzione in essa del termine contenete i nuovi valori di stato.
Tuttavia in tal modo tra le due azioni esiste una finestra temporale in cui non vi è alcuno stato noto per l'oggetto all'interno della base di conoscenza. Dunque un'altra attività che accede concorrentemente all'oggetto potrebbe incorrere in un fallimento.
Si noti che il problema si verifica anche con un'unico flusso di esecuzione. Si immagini che un metodo di un oggetto debba modificare il suo stato e dunque ritratti inizialmente lo stato dell'oggetto dalla base di conoscenza (effettuandone contestualmente la lettura dei valori). Se tale metodo invocasse ora un secondo metodo sullo stesso oggetto che modifica anch'esso lo stato, il secondo metodo incorrerebbe in un fallimento all'atto della ritrattazione dello stato dell'oggetto, poiché non presente in quel momento all'interno del database.
E' possibile superare questa problematica utilizzando il meccanismo dei semafori e collocando le modifiche alla base di conoscenza a livello delle singole azioni di ogni evento.
A tal fine si associa un semaforo ad ogni estensione di classe concreta ed un semaforo ad

ogni singolo oggetto istanziato.
Come si è detto nel paragrafo 5.4.1 una azione lavora sul solo dependency schema della classe, sulla sola intensione della classe, o su entrambe le cose.
In particolare *le azioni che lavorano sulla sola intensione di stato, prima della loro esecuzione, devono acquisire il semaforo associato all'oggetto su cui eseguono*. *Al termine della loro esecuzione devono rilasciare il semaforo precedentemente acquisito*.
Le altre due tipologie di azioni vengono invece tradotte in codice “così come sono”, in quanto invocano altri eventi che a loro volta possiederanno le loro implementazioni in termini di azioni.
Un'azione che opera su un oggetto deve dunque attendere che la precedente azione su di esso sia terminata. Poiché la sequenzializzazione avviene a livello di singola azione, è ancora possibile l'esecuzione concorrente dei metodi di un oggetto.
Le azioni indicate nella specifica che operano sulle sole intensioni di stato, rappresentano dunque le *mosse elementari* della macchina virtuale Concepts Z.
L'interprete tuProlog richiede di assegnare un identificatore univoco ad ogni semaforo [31]. Poiché le classi di un sistema devono avere un nome unico, è possibile attribuire al semaforo relativo all'estensione di una classe, l'identificatore “className + ExtensionMutex” . Tale semaforo è necessario per poter gestire in maniera consistente le creazioni e finalizzazioni degli oggetti di una certa classe in presenza di attività concorrenti. Per quanto riguarda il semaforo associato ad ogni singola istanza, un identificatore univoco per l'istanza si può ottenere concatenando il contenuto informativo del suo riferimento. In particolare dato un termine `reference( id , className )` , è possibile costruire un identificatore univoco per l'istanza, concatenando al campo className il campo id. Questo sarà dunque l'identificatore associato al semaforo dell'istanza.
Si noti infine che un'azione può in generale rappresentare la creazione di un'istanza, la sua finalizzazione, una lettura del suo stato corrente o una sua modifica, esattamente come esplicitato nella specifica delle azioni nella loro parte dichiarativa (łState’ , łState , Ξ łState , Δ łState).

All'interno di un motore Prolog, il riferimento ad una istanza viene associato al corrispondente stato di quest'ultima, attraverso asserzioni della forma `link( reference(id,className) , State )` . In tali asserzioni la variabile State risulterà, per costruzione, sempre legata ad un termine Prolog ground che rappresenta lo stato

dell'oggetto.

Si presenta ora un estratto del codice Prolog che realizza tale comportamento all'interno della macchina virtuale Concepts Z. Si tratta della teoria logica che implementa il costrutto Class, per cui essa viene riusata da tutte le altre classi definite all'interno del sistema.

```
% Class.pl

:- parents( 'ConceptsZLibrary/AnalysisConcepts.pl' ) .


appendAtoms( X , Y , Result ) :- atom( X ) ,
                                 atomic( Y ) ,
                                 atomConcatenation( X , Y , Result ) .



newStart( reference(ID,C) ) :- appendAtoms( C , 'ExtensionMutex' , MutexId ) ,
                               mutex_lock( MutexId ) ,
                               retract( referenceCount(C,ID) ) ,
                               ! ,
                               NewIdFree is ID + 1 ,
                               assert( referenceCount(C,NewIdFree) ) ,
                               mutex_unlock( MutexId ) .

newEnd( reference(ID,C) , InitialState ) :-
assert( link(reference(ID,C),InitialState) ) .


readStart( reference(ID,C) , State ) :- appendAtoms( C , ID , MutexId ) ,
                                        mutex_lock( MutexId ) ,
                                        link( reference(ID,C) , State ) .

readStart( reference(ID,C) , _ ) :- appendAtoms( C , ID , MutexId ) ,
                                    mutex_unlock( MutexId ) ,
                                    ! ,
                                    fail .
```

Le azioni di lettura, modifica e finalizzazione necessitano sempre di iniziare con una lettura dello stato. Dunque il predicato readStart(R , S) viene riusato da tutte e tre le tipologie di azioni, le quali differiscono solo per la loro parte finale. Si noti come il predicato readStart costituisca un punto di scelta all'atto della sua invocazione. Nel caso in cui l'azione corrente incorra in un fallimento (ad esempio per una precondizione violata) è infatti necessario sbloccare il semaforo associato all'istanza che era stato acquisito in fase di lettura dello stato.

```
deleteEnd( reference(ID,C) ) :- retract( link(reference(ID,C),_) ) ,
                                ! ,
```

```
                                appendAtoms( C , ID , MutexId ) ,
                                mutex_unlock( MutexId ) .

readEnd( reference(ID,C) ) :- appendAtoms( C , ID , MutexId ) ,
                              mutex_unlock( MutexId ) .

writeEnd( reference(ID,C) , State ) :- retract( link(reference(ID,C),_) ) ,
                                       ! ,
                                       assert( link(reference(ID,C),State) ) ,
                                       appendAtoms( C , ID , MutexId ) ,
                                       mutex_unlock( MutexId ) .
```

E' dunque possibile a questo punto definire anche un predicato che verrà utilizzato per tradurre le precondizioni di verifica di riferimenti non nulli presenti all'interno di una specifica di progetto.

```
workingReference( R ) :- wellFormedReference( R ) ,
                         readStart( R , _ ) ,
                         readEnd( R ) ,
                         ! .
```

E' il momento di stabilire una rappresentazione per lo stato di un oggetto.
Come si è detto la relazione di ereditarietà fra due classi impone che lo stato di una sottoclasse evolva coerentemente con quanto specificato nella superclasse. Inoltre deve essere possibile per le implementazioni di eventi definiti nella superclasse operare in modo generico anche sullo stato di istanze di sottoclassi.
Per tali motivi di sceglie di rappresentare lo stato di un oggetto attraverso una *lista differenza* [29]. All'interno della macchina virtuale una lista differenza è rappresentata dal termine `dl(X,Y)` (ad es. `dl([1,2,3|X],X)`). Una lista differenza rappresenta una struttura dati parzialmente specificata e dunque è un ottimo espediente per consentire agli eventi di una superclasse di agire sulla sola porzione di stato che le compete. Inoltre tale scelta consente di effettuare computazioni molto efficienti su liste di elementi.
Si noti che questa scelta consente al codice di una superclasse di restare invariato qualora venga aggiunta successivamente una nuova sottoclasse al sistema.
La parte restante di codice rappresenta le implementazioni definite nella classe Class (paragrafo 4.4.1).
Si noti che uno Z schema rappresentante un'azione può essere reso direttamente come una regola Prolog. In particolare la testa della regola ha funtore principale pari al nome dello schema (unico per costruzione) e variabili con nome identico al nome delle variabili indicate nella parte dichiarativa dello schema, con la lettera iniziale maiuscola ed il resto

minuscolo, come argomenti. Il corpo della regola è invece una congiunzione di predicati che corrispondono alle proposizioni espresse nella parte predicativa dello schema.
Per convenzione indicheremo nel codice Prolog le variabili corrispondenti a controparti presenti nella specifica con la sola iniziale maiuscola, mentre le variabili con tutti i caratteri maiuscoli rappresenteranno “variabili di appoggio” utilizzate all'interno della macchina virtuale. Le variabili di ingresso contrassegnate nella specifica con “?” avranno inoltre il suffisso “_In” mentre le variabili d'uscita contrassegnate con “!” avranno il suffisso “_Out” nel codice Prolog. Infine, per quanto riguarda le variabili dello stato finale, la decorazione “**’**” presente nella specifica corrisponderà al suffisso “_1” in una variabile Prolog.

```
% dl( [reference(ID,C)|REST] , REST )


classInitImpl( STATE_1 ,
               This_In ) :- classInitImpl_1( STATE_1 ,
                                             This_In ) .

classInitImpl_1( dl([reference(ID_1,C_1)|_],_) ,
                 This_In ) :- equality( reference(ID_1,C_1) , This_In ) .


classFinImpl( _ ) .
```

L'interprete tiene traccia solo delle estensioni delle classi concrete. Questo significa che degli eventi definiti da superclassi astratte saranno presenti esclusivamente i predicati di pre e post condizione dell'evento ma non il predicato che rappresenta globalmente il metodo stesso. Questo poiché non esistono istanze su cui applicare il metodo definito dalla superclasse astratta, bensì solo sottoclassi concrete che ne definiscono sue specializzazioni, che devono essere coerenti con quanto definito dalla superclasse.
A tal fine gli eventi definiti nelle sottoclassi concrete, invocano i predicati di pre e post condizione dell'evento definiti nella superclasse astratta.

7.6.2 Il polimorfismo

Si vuole in questo paragrafo mettere in evidenza quale sia l'*implementazione del polimorfismo* all'interno della macchina virtuale Concepts Z.
Come è possibile vedere dalle specifiche espresse nella notazione Concepts Z, le signature degli eventi che possono avvenire ad una classe, iniziano sempre con un

riferimento all'oggetto su cui deve avvenire l'evento. Tale variabile viene aggiunta in testa alla signature dell'evento dallo *schema di framing* (*f*), il quale rende un evento invocabile dall'esterno. Come detto la forma concreta di tali riferimenti è un termine del tipo `reference( id , className )`. Tale riferimento contiene al suo interno informazione circa la classe più specifica dell'istanza (className) ed *è sempre ground*.
Dopo essere stato creato, un riferimento viene però legato ad una variabile ed è sempre trattato come *opaco* nel codice cliente.
Riprendendo l'esempio del paragrafo 4.5.1, qualora avessimo una serie di istanze appartenenti a diverse classi della gerarchia dell'esempio, l'invocazione seguente

```
... ,
setNext( R , N ) ,
...
```

secondo il paradigma object-oriented, dovrebbe avere un comportamento dipendente dalla classe più specifica a cui appartiene l'oggetto riferito da R. Occorre dunque raggiungere in qualche modo la versione dell'evento definita da tale classe.
Seguendo l'approccio suggerito dalla notazione Concepts Z, si rendono gli eventi di una classe come regole Prolog, aventi il nome dell'evento come funtore principale e un riferimento del tipo desiderato come primo argomento.
Per l'esempio del paragrafo 4.5.1, relativamente all'evento polimorfo SetNext, si avrebbero all'interno della base di conoscenza dell'interprete Prolog, le seguenti regole:

```
setNext( reference(ID,sampleImpl) , N_In ) :-
sampleImplSetNextImpl( reference(ID,sampleImpl) , N_In ) .

      . . .

setNext( reference(ID,resetSampleImpl) , N_In ) :-
resetSampleImplSetNextImpl( reference(ID,resetSampleImpl) , N_In ) .

      . . .

setNext( reference(ID,historySampleImpl) , N_In ) :-
historySampleImplSetNextImpl( reference(ID,historySampleImpl) , N_In ) .
```

Poiché il riferimento utilizzato come primo argomento della chiamata è sempre ground e contiene al suo interno la classe più specifica dell'istanza riferita, l'unificazione su tale primo argomento consente di selezionare univocamente la versione corretta dell'evento da

eseguire.

Dunque è possibile *sfruttare il meccanismo di unificazione* dell'interprete Prolog *per implementare direttamente il polimorfismo*.

Si noti infine che a causa dalla esplicita dichiarazione del nome di classe nel primo argomento di un evento, l'ordine delle regole mostrate precedentemente non è rilevante.

L'informazione relativa alla classe più specifica di un'istanza, presente nel suo riferimento, fin dalla sua creazione, consente di utilizzare l'unificazione per invocare la regola corretta, corrispondente alla versione fornita dalla sua classe per l'evento. Questo consente all'atto della invocazione di un evento di raggiungere la regola corrispondente alla versione più specifica per l'istanza, la quale nella sua implementazione, può richiamare i predicati della implementazione dell'evento, forniti nella superclasse.

7.7 Generatore di codice di Concepts Z

Come detto in precedenza il componente Concepts Z M2C mappa gli elementi del linguaggio Concepts Z in codice Prolog che utilizza la macchina astratta sviluppata per Concepts Z. Tale componente incarna dunque la definizione della semantica operazionale del linguaggio.

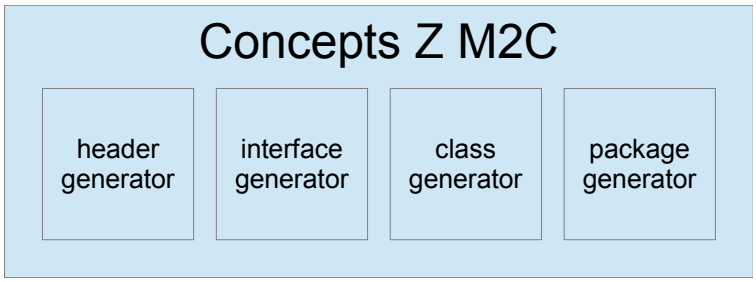
Si è sottolineata nel paragrafo 7.2 la scelta di delegare le fasi di parsing e di typechecking di una specifica in Concepts Z al tool CZT. Questa scelta consente di saltare le fasi di sviluppo di un riconoscitore per Concepts Z, tuttavia non è esente da problematiche.

Il tool CZT, a valle delle fasi di parsing e typechecking, costruisce un AST della specifica, le cui classi sono generate automaticamente a partire da un XML schema che definisce un mark-up XML per le specifiche Z [32]. Tale mark-up supporta la ricostruzione completa delle frasi della sintassi concreta che hanno portato alla costruzione di un certo albero. Questo consente di attribuire un significato leggermente diverso ai costrutti del linguaggio, in base al contesto in cui appaiono (esattamente come si è scelto di fare per molti costrutti della notazione Concepts Z), senza perdere tali informazioni a valle dell'analisi sintattico-semantica. Tuttavia questo complica la sintassi astratta, aumentando il numero di classi necessarie nell'AST, che dunque diviene più simile ad un albero di derivazione che ad un albero sintattico astratto. Ne segue una maggiore complessità nella costruzione di valutatori per tali alberi.

Un altro fattore che aumenta la complessità è dovuto al fatto che CZT è un parser per il

linguaggio Z e non per Concepts Z. Dunque si è costretti nell'AST ad avere a che fare con tutte le “sotto-strutture” di Z che realizzano sintatticamente i costrutti di Concepts Z.
Tipicamente un valutatore per un AST è esprimibile sotto forma di un componente che attraversa l'albero tramite il pattern Visitor [11]. CZT mette a disposizione alcune utilities per la loro realizzazione [5].
E' possibile, in certa misura, affrontare la accresciuta complessità nella costruzione del valutatore, attraverso strumenti specificamente volti alla produzione di generatori di codice come il framework Xpand [33]. Tale framework permette di specificare in forma pseudo-dichiarativa un componente che elabora un grafo di oggetti e produce testo in uscita. Ciò equivale a tutti gli effetti alla definizione di un Visitor per un AST. E' necessaria però la trasformazione in formato XML della specifica contenente il modello di progetto e la sua successiva elaborazione. Come è noto, l'elaborazione di un documento XML introduce un considerevole overhead rispetto alla elaborazione di un grafo di classi Java, rappresentando l'inevitabile trade-off per l'utilizzo del framework Xpand, la scelta esplorata in questa sede.

E' possibile dividere in 4 sotto-componenti fondamentali il problema della realizzazione del Concepts Z M2C. In particolare si identificano le seguenti parti:



header generator: si occupa delle parti della specifica che nominano classi e interfacce presenti nel sistema e le loro relazioni.

interface generator: si occupa della generazione del codice Prolog relativo alla specifica di un'interfaccia.

class generator: si occupa della generazione del codice corrispondente ad una classe implementativa

package generator: si occupa della generazione del codice relativo all'inizializzazione del sistema e ai suoi eventi complessivi. Può essere esteso, assieme all'introduzione di ulteriore informazione nella system view, per la generazione di artefatti ulteriori (ad es. file di configurazione di piattaforme a componenti).

Tali generatori verranno utilizzati in modo coordinato all'interno di un *workflow* [13], il quale preso in ingresso un modello di progetto espresso in Concepts Z, collega tutte le stazioni della fabbrica del software per quest'ultimo.

Come detto l'utilizzo del Prolog rende già disponibili in modo nativo alcuni costrutti a livello di linguaggio. Si rende tuttavia necessario, per questioni realizzative, definire un operatore ad-hoc per le disgiunzioni presenti nella specifica Z:

```
:- op( 1000 , xfy , or ) .

A or _ :- call( A ) ,
          ! .

_ or B :- call( B ) .
```

Le congiunzioni vengono invece rese tramite l'usuale operatore Prolog “,” .

Per illustrare il mapping Concepts Z-Prolog utilizzato dal generatore, verrà mostrato il codice relativo all'esempio di progettazione presente nel capitolo 5.
Si noti che le viste intensionale,relazionale,estensionale vengono collassate in un'unica teoria logica relativa all'entità descritta.
La forma concreta del sistema sarà dunque quella di una serie di teorie logiche che si includono tramite la direttiva `:- parents(...)` .
Si noti che, data una implementazione corretta degli eventi, a run time non è strettamente necessaria la verifica degli invarianti, ne delle post-condizioni, ma solo la verifica delle precondizioni specificate negli eventi per determinare se in un dato stato un evento può avvenire o meno e in quest'ultimo caso fallire o lanciare eccezione.
Si sottolinea che il codice seguente non riflette una programmazione manuale, ma l'output

di un generatore automatico, che produce codice in accordo a schemi ben riconoscibili. Il codice prodotto dal Concepts Z M2C può poi essere oggetto di ulteriori ottimizzazioni, in modo analogo a quanto avviene per un linguaggio intermedio all'interno dei compilatori [34].

7.7.1 La Teoria logica DomainModelDesign

```
% DomainModelDesign.pl

:- parents( 'ConceptsZLibrary/DesignConcepts.pl' ) .


interface( iSample ) .

class( sampleImpl ) .
isA( sampleImpl , class ) .
referenceCount( sampleImpl , 0 ) .
implements( sampleImpl , iSample ) .

class( sampleFactory ) .
isA( sampleFactory , class ) .
referenceCount( sampleFactory , 0 ) .

interface( iConsumptionEstimator ) .

class( consumptionEstimatorImpl ) .
isA( consumptionEstimatorImpl , class ) .
referenceCount( consumptionEstimatorImpl , 0 ) .
implements( consumptionEstimatorImpl , iConsumptionEstimator ) .

class( consumptionEstimatorFactory ) .
isA( consumptionEstimatorFactory , class ) .
referenceCount( consumptionEstimatorFactory , 0 ) .
```

7.7.2 La teoria logica iSample

```
% iSample.pl

:- parents( 'ModelloDelDominio/DomainModelDesign.pl' ) .

:- parents( 'ConceptsZLibrary/Class.pl' ) .


iSampleCompare( S1 , S2 ) :- getValue( S1 , Vs1 ) ,
                             getValue( S2 , Vs2 ) ,
                             Vs1 =< Vs2 .


order( niSample , iSampleCompare ) .
```

7.7.3 La teoria logica SampleImpl

Si noti come la variabile ID risulti non legata all'atto della chiamata a newStart() e come il mapping dei costruttori e finalizzatori faccia eccezione rispetto a a quello degli eventi ordinari. Si noti infine come, in caso di esito positivo dell'azione, si provveda sempre a tagliare il punto di scelta lasciato aperto da readStart() tramite l'operatore “!”.

```
% SampleImpl.pl

:- parents( 'ModelloDelDominio/iSample.pl' ) .


% dl( [This,Value,Next|REST] , REST )


sampleImplNew( reference(ID,sampleImpl) , Value_In ) :-
newStart( reference(ID,sampleImpl) ) ,
sampleImplNewImpl( dl([This,
                       Value,
                       Next|REST],REST) ,
                   reference(ID,sampleImpl) ,
                   Value_In ) ,

newEnd( reference(ID,sampleImpl) , dl([This,
                                       Value,
                                       Next|REST],REST) ) .

sampleImplNewImpl( STATE_1,
                   This_In,
                   Value_In ) :- sampleImplNewImpl_1( STATE_1,
                                                      This_In ,
                                                      Value_In ) .

sampleImplNewImpl_1_1( STATE_1,
                       This_In,
                       Value_In ) :- extension( sampleImpl , This_In ) ,
                                     Value_In >= 0 ,
                                     classInitImpl( STATE_1 ,
                                                    This_In ) .

sampleImplNewImpl_1_2( dl([This_1,
                           Value_1,
                           Next_1|TAIL],REST),
                       This_In,
                       Value_In ) :- equality( Value_1 , Value_In ) ,
                                     equality( Next_1 , This_In ) .

sampleImplNewImpl_1( STATE_1 ,
                     This_In ,
                     Value_In ) :- sampleImplNewImpl_1_1( STATE_1 ,
                                                          This_In ,
```

```
                                                Value_In ) ,
                                    sampleImplNewImpl_1_2( STATE_1 ,
                                                This_In ,
                                                Value_In ) .


sampleImplDelete( reference(ID,sampleImpl) ) :-
readStart( reference(ID,sampleImpl) , dl([This,
                                          Value,
                                          Next|REST],REST) ) ,
sampleImplDeleteImpl( dl([This,
                          Value,
                          Next|REST],REST) ) ,
deleteEnd( reference(ID,sampleImpl) ) ,
! .

sampleImplDeleteImpl( STATE ) :- sampleImplDeleteImpl_1( STATE ) .

sampleImplDeleteImpl_1( STATE ) :- classFinImpl( STATE ) .



getValue( reference(ID,sampleImpl) , Value_Out ) :-
sampleImplGetValueImpl( reference(ID,sampleImpl) , Value_Out ) .

sampleImplGetValueImpl( REF , Value_Out ) :- sampleImplGetValueImpl_1( REF ,
Value_Out ) .

sampleImplGetValueImpl_1( REF , Value_Out ) :- readStart( REF , dl([This,Value,Next|
TAIL],REST) ) ,
                                               equality( Value_Out , Value ) ,
                                               readEnd( REF ) ,
                                               ! .



getNext( reference(ID,sampleImpl) , Next_Out ) :-
sampleImplGetNextImpl( reference(ID,sampleImpl) , Next_Out ) .

sampleImplGetNextImpl( REF , Next_Out ) :- sampleImplGetNextImpl_1( REF , Next_Out ) .

sampleImplGetNextImpl_1( REF , Next_Out ) :- readStart( REF , dl([This,Value,Next|
TAIL],REST) ) ,
                                             equality( Next_Out , Next ) ,
                                             readEnd( REF ) ,
                                             !.


setNext( reference(ID,sampleImpl) , N_In ) :-
sampleImplSetNextImpl( reference(ID,sampleImpl) , N_In ) .

sampleImplSetNextImpl( REF , N_In ) :- sampleImplSetNextImpl_1( REF , N_In ) .

sampleImplSetNextImpl_1_1( N_In ) :- workingReference( N_In ) .

sampleImplSetNextImpl_1_2( REF , N_In ) :- readStart( REF , dl([This,Value,Next|
TAIL],REST) ) ,
                                           equality( This_1 , This ) ,
```

```
                                        equality( Value_1 , Value ) ,
                                        equality( Next_1 , N_In ) ,
                                        STATE_1 = dl([This_1,Value_1,Next_1|
TAIL],REST) ,
                                        writeEnd( REF , STATE_1 ) ,
                                        ! .

sampleImplSetNextImpl_1( REF , N_In ) :- sampleImplSetNextImpl_1_1( N_In ) ,
                                         sampleImplSetNextImpl_1_2( REF , N_In ) .
```

7.7.4 La teoria logica SampleFactory

```
% SampleFactory.pl

:- parents( 'ModelloDelDominio/SampleImpl.pl' ) .


% dl( [This|REST] , REST )


sampleFactoryNew( reference(ID,sampleFactory) ) :-
newStart( reference(ID,sampleFactory) ) ,
sampleFactoryNewImpl( dl([This|REST],REST) ,
                      reference(ID,sampleFactory) ) ,
newEnd( reference(ID,sampleFactory) , dl([This|REST],REST) ) .

sampleFactoryNewImpl( STATE_1,
                      This_In) :- sampleFactoryNewImpl_1( STATE_1,
                                                          This_In ) .

sampleFactoryNewImpl_1( STATE_1,
                        This_In ) :- extension( sampleFactory , This_In ) ,
                                     classInitImpl( STATE_1 ,
                                                    This_In ) .



sampleFactoryDelete( reference(ID,sampleFactory) ) :-
readStart( reference(ID,sampleFactory) , dl([This|REST],REST) ) ,

sampleFactoryDeleteImpl( dl([This|REST],REST) ) ,

deleteEnd( reference(ID,sampleFactory) ) ,
                                                    ! .

sampleFactoryDeleteImpl( STATE ) :- sampleFactoryDeleteImpl_1( STATE ) .

sampleFactoryDeleteImpl_1( STATE ) :- classFinImpl( STATE ) .



sampleFactoryGetSample( reference(ID,sampleFactory) , Value_In , Os_Out ) :-
sampleFactoryGetSampleImpl( reference(ID,sampleFactory) , Value_In , Os_Out ) .

sampleFactoryGetSampleImpl( REF , Value_In , Os_Out ) :-
sampleFactoryGetSampleImpl_1( REF , Value_In , Os_Out ) .
```

```
sampleFactoryGetSampleImpl_1_1( Value_In , Os_Out ) :- Value_In > 0 ,
                                                       sampleImplNew( Os_Out , Value_In
) .

sampleFactoryGetSampleImpl_1_2( REF ) :- readStart( REF , dl([This|TAIL],REST) ) ,
                                         equality( This_1 , This ) ,
                                         STATE_1 = dl([This_1|TAIL],REST) ,
                                         writeEnd( REF , STATE_1 ) ,
                                         ! .

sampleFactoryGetSampleImpl_1( REF , Value_In , Os_Out ) :-
sampleFactoryGetSampleImpl_1_1( Value_In , Os_Out ) ,
sampleFactoryGetSampleImpl_1_2( REF ) .
```

7.7.5 La teoria logica iConsumptionEstimator

Nella specifica di questa interfaccia non vi sono definizioni di ordinamenti. Dunque la corrispondente teoria logica di limita ad includere le altre teorie da cui l'interfaccia dipende. Con gli accorgimenti adottati in fase di specifica non occorre infatti la generazione di alcunché per le specifiche di interfaccia, eccetto per le dipendenze ed eventuali ordinamenti definiti sulle istanze.

```
% iConsumptionEstimator.pl

:- parents( 'ModelloDelDominio/SampleFactory.pl' ) .
```

7.7.6 la teoria logica ConsumptionEstimatorImpl

In questa teoria logica è possibile notare come vengono gestiti i vincoli impliciti nelle dichiarazioni (declaration constraints): il predicato corrispondente allo schema che dichiara variabili con vincoli impliciti ha il compito aggiuntivo di verificarli.
Si può inoltre vedere come le espressioni numeriche siano sempre valutate tramite il predicato “is” prima del loro uso.
Si noti che è possibile ottenere la sequenza di predicati Prolog che implementano una certa espressione Z, tramite una visita in postordine del sottoalbero dell'AST corrispondente all'espressione.

```
% ConsumptionEstimatorImpl.pl
```

```
:- parents( 'ModelloDelDominio/iConsumptionEstimator.pl' ) .


%
dl( [This,SampleWindow,MaxSize,Oldest,Newest,Q1,Q2,Q3,Iqr,MinConsecutiveStrangeSamples,
StrangeSamples,SenseFault|REST] , REST )


consumptionEstimatorImplNew( reference(ID,consumptionEstimatorImpl) ,
                             MaxSize_In ,
                             MinConsecutiveStrangeSamples_In ,
                             Factory_In ) :-
strictlyPositiveNatural( MaxSize_In ) ,
strictlyPositiveNatural( MinConsecutiveStrangeSamples_In ) ,
newStart( reference(ID,consumptionEstimatorImpl) ) ,
consumptionEstimatorImplNewImpl( dl([This,
                                     SampleWindow,
                                     MaxSize,
                                     Oldest,
                                     Newest,
                                     Q1,
                                     Q2,
                                     Q3,
                                     Iqr,
                                     MinConsecutiveStrangeSamples,
                                     StrangeSamples,
                                     SenseFault,
                                     Factory|REST],REST) ,
                                  reference(ID,consumptionEstimatorImpl) ,
                                  MaxSize_In ,
                                  MinConsecutiveStrangeSamples_In ,
                                  Factory_In ) ,
newEnd( reference(ID,consumptionEstimatorImpl) , dl([This,
                                                     SampleWindow,
                                                     MaxSize,
                                                     Oldest,
                                                     Newest,
                                                     Q1,
                                                     Q2,
                                                     Q3,
                                                     Iqr,
                                                     MinConsecutiveStrangeSamples,
                                                     StrangeSamples,
                                                     SenseFault,
                                                     Factory|REST],REST) ) .

consumptionEstimatorImplNewImpl( STATE_1 ,
                                 This_In ,
                                 MaxSize_In ,
                                 MinConsecutiveStrangeSamples_In ,
                                 Factory_In ) :-
consumptionEstimatorImplNewImpl_1( STATE_1 ,
                                   This_In ,
                                   MaxSize_In ,
                                   MinConsecutiveStrangeSamples_In ,
                                   Factory_In ) .


consumptionEstimatorImplNewImpl_1_1( STATE_1 ,
```

```
                                    This_In ,
                                    MaxSize_In ,
                                    MinConsecutiveStrangeSamples_In ,
                                    Factory_In ) :-
extension( consumptionEstimatorImpl , This_In ) ,
MaxSize_In >= 7 ,
VAR1 is MaxSize_In * 5 / 100 ,
VAR1 < MinConsecutiveStrangeSamples_In ,
VAR2 is MaxSize_In * 15 / 100 ,
MinConsecutiveStrangeSamples_In =< VAR2 ,
workingReference( Factory_In ) ,
classInitImpl( STATE_1 ,
               This_In ) .


consumptionEstimatorImplNewImpl_1_2( dl([This_1,
                                         SampleWindow_1,
                                         MaxSize_1,
                                         Oldest_1,
                                         Newest_1,
                                         Q1_1,
                                         Q2_1,
                                         Q3_1,
                                         Iqr_1,
                                         MinConsecutiveStrangeSamples_1,
                                         StrangeSamples_1,
                                         SenseFault_1,
                                         Factory_1|TAIL],REST) ,
                                    MaxSize_In ,
                                    MinConsecutiveStrangeSamples_In ,
                                    Factory_In ) :-
maplet( [] , niSample , VAR1 ) ,
equality( SampleWindow_1 , VAR1 ) ,
equality( MaxSize_1 , MaxSize_In ) ,
sequenceBrackets( [] , VAR2 ) ,
equality( Oldest_1 , VAR2 ) ,
sequenceBrackets( [] , VAR3 ) ,
equality( Newest_1 , VAR3 ) ,
equality( MinConsecutiveStrangeSamples_1 , MinConsecutiveStrangeSamples_In ) ,
equality( StrangeSamples_1 , 0 ) ,
equality( SenseFault_1 , f ) ,
equality( Iqr_1 , -1 ) ,
equality( Q3_1 , Iqr_1 ) ,
equality( Q2_1 , Q3_1 ) ,
equality( Q1_1 , Q2_1 ) ,
equality( Factory_1 , Factory_In ) .

consumptionEstimatorImplNewImpl_1( STATE_1 ,
                                   This_In ,
                                   MaxSize_In ,
                                   MinConsecutiveStrangeSamples_In ,
                                   Factory_In ) :-
consumptionEstimatorImplNewImpl_1_1( STATE_1 ,
                                     This_In ,
                                     MaxSize_In ,
                                     MinConsecutiveStrangeSamples_In ,
                                     Factory_In ) ,
consumptionEstimatorImplNewImpl_1_2( STATE_1 ,
                                      MaxSize_In ,
```

```
                                    MinConsecutiveStrangeSamples_In ,
                                    Factory_In ) .



consumptionEstimatorImplDelete( reference(ID,consumptionEstimatorImpl) ) :-
readStart( reference(ID,consumptionEstimatorImpl) , dl([This,
                                                        SampleWindow,
                                                        MaxSize,
                                                        Oldest,
                                                        Newest,
                                                        Q1,
                                                        Q2,
                                                        Q3,
                                                        Iqr,
                                                        MinConsecutiveStrangeSamples,
                                                        StrangeSamples,
                                                        SenseFault,
                                                        Factory|REST],REST)) ,

consumptionEstimatorImplDeleteImpl( dl([This,
                                        SampleWindow,
                                        MaxSize,
                                        Oldest,
                                        Newest,
                                        Q1,
                                        Q2,
                                        Q3,
                                        Iqr,
                                        MinConsecutiveStrangeSamples,
                                        StrangeSamples,
                                        SenseFault,
                                        Factory|REST],REST) ) ,
deleteEnd( reference(ID,consumptionEstimatorImpl) ) ,
! .

consumptionEstimatorImplDeleteImpl( STATE ) :-
consumptionEstimatorImplDeleteImpl_1( STATE ) .

consumptionEstimatorImplDeleteImpl_1( STATE ) :- classFinImpl( STATE ) .



trim( reference(ID,consumptionEstimatorImpl) ) :-
consumptionEstimatorImplTrimImpl( reference(ID,consumptionEstimatorImpl) ) .

consumptionEstimatorImplTrimImpl( REF ) :- consumptionEstimatorImplTrimImpl_1( REF ) .

consumptionEstimatorImplTrimImpl_1( REF ) :- readStart( REF , dl([This,
                                                                  SampleWindow,
                                                                  MaxSize,
                                                                  Oldest,
                                                                  Newest,
                                                                  Q1,
                                                                  Q2,
                                                                  Q3,
```

```
                                                                Iqr,

MinConsecutiveStrangeSamples,
                                                                StrangeSamples,
                                                                SenseFault,
                                                                Factory|TAIL],REST) )
,
                                                  equality( This_1 , This ) ,
                                                  maplet( [] , niSample , VAR1 ) ,
                                                  equality( SampleWindow_1 , VAR1 ) ,
                                                  equality( MaxSize_1 , MaxSize ) ,
                                                  sequenceBrackets( [] , VAR2 ) ,
                                                  equality( Oldest_1 , VAR2 ) ,
                                                  sequenceBrackets( [] , VAR3 ) ,
                                                  equality( Newest_1 , VAR3 ) ,
                                                  equality( MinConsecutiveStrangeSamples_1 ,
MinConsecutiveStrangeSamples ) ,
                                                     equality( StrangeSamples_1 , 0 ) ,
                                                     equality( SenseFault_1 , f ) ,
                                                     equality( Iqr_1 , -1 ) ,
                                                     equality( Q3_1 , Iqr_1 ) ,
                                                     equality( Q2_1 , Q3_1 ) ,
                                                     equality( Q1_1 , Q2_1 ) ,
                                                     equality( Factory_1 , Factory ) ,
                                                     STATE_1 = dl([This_1,
                                                                   SampleWindow_1,
                                                                   MaxSize_1,
                                                                   Oldest_1,
                                                                   Newest_1,
                                                                   Q1_1,
                                                                   Q2_1,
                                                                   Q3_1,
                                                                   Iqr_1,

MinConsecutiveStrangeSamples_1,
                                                                   StrangeSamples_1,
                                                                   SenseFault_1,
                                                                   Factory_1|TAIL],REST) ,
                                                  writeEnd( REF , STATE_1 ) ,
                                                  ! .


newData( reference(ID,consumptionEstimatorImpl) ,
         Data_In ,
         Q2_Out ,
         Iqr_Out ) :-
consumptionEstimatorImplNewDataImpl( reference(ID,consumptionEstimatorImpl) , Data_In ,
Q2_Out , Iqr_Out ) .

consumptionEstimatorImplNewDataImpl( REF ,
                                     Data_In ,
                                     Q2_Out ,
                                     Iqr_Out ) :-
consumptionEstimatorImplNewDataImpl_1( REF , Data_In , Os , SizeBefore ) ,
consumptionEstimatorImplNewDataImpl_2( REF , SizeBefore , Os ) ,
consumptionEstimatorImplNewDataImpl_3( REF , Data_In , SizeAfter ) ,
consumptionEstimatorImplNewDataImpl_4( REF , SizeAfter , Q2_Out , Iqr_Out ) .
```

```
consumptionEstimatorImplNewDataImpl_1_1( REF , Data_In , SizeBefore , F ) :-
readStart( REF , dl([This,
                     SampleWindow,
                     MaxSize,
                     Oldest,
                     Newest,
                     Q1,
                     Q2,
                     Q3,
                     Iqr,
                     MinConsecutiveStrangeSamples,
                     StrangeSamples,
                     SenseFault,
                     Factory|TAIL],REST) ) ,
Data_In >= 0 ,
tupleSelection( SampleWindow , 1 , VAR1 ) ,
size( VAR1 , VAR2 ) ,
equality( SizeBefore , VAR2 ) ,
equality( F , Factory ) ,
readEnd( REF ) ,
! .

consumptionEstimatorImplNewDataImpl_1_2( Data_In , F , Os ) :-
sampleFactoryGetSample( F , Data_In , Os ) .

consumptionEstimatorImplNewDataImpl_1( REF , Data_In , Os , SizeBefore ) :-
consumptionEstimatorImplNewDataImpl_1_1( REF , Data_In , SizeBefore , F ) ,
consumptionEstimatorImplNewDataImpl_1_2( Data_In , F , Os ) .

consumptionEstimatorImplNewDataImpl_2_NoSamples_1( REF , SizeBefore , Os ) :-
readStart( REF , dl([This,
                     SampleWindow,
                     MaxSize,
                     Oldest,
                     Newest,
                     Q1,
                     Q2,
                     Q3,
                     Iqr,
                     MinConsecutiveStrangeSamples,
                     StrangeSamples,
                     SenseFault,
                     Factory|TAIL],REST) ) ,
equality( SizeBefore , 0 ) ,
add( SampleWindow , Os , VAR1 ) ,
equality( SampleWindow_1 , VAR1 ) ,
sequenceBrackets( [Os] , VAR2 ) ,
equality( Oldest_1 , VAR2 ) ,
sequenceBrackets( [Os] , VAR3 ) ,
equality( Newest_1 , VAR3 ) ,
STATE_1 = dl([This,
              SampleWindow_1,
              MaxSize,
              Oldest_1,
              Newest_1,
              Q1,
              Q2,
              Q3,
              Iqr,
```

```
                MinConsecutiveStrangeSamples,
                StrangeSamples,
                SenseFault,
                Factory|TAIL],REST) ,
writeEnd( REF , STATE_1 ) ,
! .

consumptionEstimatorImplNewDataImpl_2_NoSamples( REF , SizeBefore , Os ) :-
consumptionEstimatorImplNewDataImpl_2_NoSamples_1( REF , SizeBefore , Os ) .

consumptionEstimatorImplNewDataImpl_2_RoomForSamples_1( REF , SizeBefore , Os ,
OldNewest ) :- readStart( REF , dl([This,
                                    SampleWindow,
                                    MaxSize,
                                    Oldest,
                                    Newest,
                                    Q1,
                                    Q2,
                                    Q3,
                                    Iqr,
                                    MinConsecutiveStrangeSamples,
                                    StrangeSamples,
                                    SenseFault,
                                    Factory|TAIL],REST) ) ,
                  SizeBefore > 0 ,
                  SizeBefore < MaxSize ,
                  add( SampleWindow , Os , VAR1 ) ,
                  equality( SampleWindow_1 , VAR1 ) ,
                  sequenceBrackets( [Os] , VAR2 ) ,
                  equality( Newest_1 , VAR2 ) ,
                  head( Newest , VAR3 ) ,
                  equality( OldNewest , VAR3 ) ,
                  STATE_1 = dl([This,
                                SampleWindow_1,
                                MaxSize,
                                Oldest,
                                Newest_1,
                                Q1,
                                Q2,
                                Q3,
                                Iqr,
                                MinConsecutiveStrangeSamples,
                                StrangeSamples,
                                SenseFault,
                                Factory|TAIL],REST) ,
                  writeEnd( REF , STATE_1 ) ,
                  ! .

consumptionEstimatorImplNewDataImpl_2_RoomForSamples_2( OldNewest , Os ) :-
setNext( OldNewest , Os ) .

consumptionEstimatorImplNewDataImpl_2_RoomForSamples( REF ,
                                                      SizeBefore ,
                                                      Os ) :-
consumptionEstimatorImplNewDataImpl_2_RoomForSamples_1( REF , SizeBefore , Os ,
OldNewest ) ,
consumptionEstimatorImplNewDataImpl_2_RoomForSamples_2( OldNewest , Os ) .

consumptionEstimatorImplNewDataImpl_2_NoRoomForSamples_1( REF ,
```

```
                                                        SizeBefore ,
                                                        OldOldest ,
                                                        OldNewest ) :-
readStart( REF , dl([This,
                     SampleWindow,
                     MaxSize,
                     Oldest,
                     Newest,
                     Q1,
                     Q2,
                     Q3,
                      Iqr,
                      MinConsecutiveStrangeSamples,
                      StrangeSamples,
                      SenseFault,
                      Factory|TAIL],REST) ) ,
equality( SizeBefore , MaxSize ) ,
head( Oldest , VAR1 ) ,
equality( OldOldest , VAR1 ) ,
head( Newest , VAR2 ) ,
equality( OldNewest , VAR2 ) ,
readEnd( REF ) ,
! .

consumptionEstimatorImplNewDataImpl_2_NoRoomForSamples_2( OldOldest ,
                                                          OldNewest ,
                                                          Os ,
                                                          NewOldest ) :-
setNext( OldNewest , Os ) ,
getNext( OldOldest , NewOldest ) .

consumptionEstimatorImplNewDataImpl_2_NoRoomForSamples_3( REF ,
                                                          OldOldest ,
                                                          NewOldest ,
                                                          Os ) :-
readStart( REF , dl([This,
                     SampleWindow,
                     MaxSize,
                     Oldest,
                     Newest,
                     Q1,
                     Q2,
                     Q3,
                     Iqr,
                     MinConsecutiveStrangeSamples,
                     StrangeSamples,
                     SenseFault,
                     Factory|TAIL],REST) ) ,

remove( SampleWindow , OldOldest , VAR1 ) ,
add( VAR1 , Os , VAR2 ) ,
equality( SampleWindow_1 , VAR2 ) ,
sequenceBrackets( [NewOldest] , VAR3 ) ,
equality( Oldest_1 , VAR3 ) ,
sequenceBrackets( [Os] , VAR4 ) ,
equality( Newest_1 , VAR4 ) ,
STATE_1 = dl([This,
              SampleWindow_1,
              MaxSize,
```

```
                Oldest_1,
                Newest_1,
                Q1,
                Q2,
                Q3,
                Iqr,
                MinConsecutiveStrangeSamples,
                StrangeSamples,
                SenseFault,
                Factory|TAIL],REST) ,
writeEnd( REF , STATE_1 ) ,
! .

consumptionEstimatorImplNewDataImpl_2_NoRoomForSamples( REF ,
                                                        SizeBefore ,
                                                        Os ) :-
consumptionEstimatorImplNewDataImpl_2_NoRoomForSamples_1( REF ,
                                                          SizeBefore ,
                                                          OldOldest ,
                                                          OldNewest ) ,
consumptionEstimatorImplNewDataImpl_2_NoRoomForSamples_2( OldOldest ,
                                                          OldNewest ,
                                                          Os ,
                                                          NewOldest ) ,
consumptionEstimatorImplNewDataImpl_2_NoRoomForSamples_3( REF ,
                                                          OldOldest ,
                                                          NewOldest ,
                                                          Os ) .




consumptionEstimatorImplNewDataImpl_2( REF , SizeBefore , Os ) :-
consumptionEstimatorImplNewDataImpl_2_NoSamples( REF , SizeBefore , Os ) or
consumptionEstimatorImplNewDataImpl_2_RoomForSamples( REF , SizeBefore , Os ) or
consumptionEstimatorImplNewDataImpl_2_NoRoomForSamples( REF , SizeBefore , Os ) .



consumptionEstimatorImplNewDataImpl_3_1( REF , Data_In , SizeAfter ) :-
readStart( REF , dl([This,
                     SampleWindow,
                     MaxSize,
                     Oldest,
                     Newest,
                     Q1,
                     Q2,
                     Q3,
                     Iqr,
                     MinConsecutiveStrangeSamples,
                     StrangeSamples,
                     SenseFault,
                     Factory|TAIL],REST) ) ,
tupleSelection( SampleWindow , 1 , VAR1 ) ,
size( VAR1 , VAR2 ) ,
equality( SizeAfter , VAR2 ) ,
implication( (VAR3 is MaxSize*85/100,SizeAfter >= VAR3,equality(SenseFault , f)) ,
```

```
            (implication((VAR4 is Q2-Iqr/2,VAR5 is
Q2+Iqr/2,numberRange(VAR4,VAR5,VAR6),membership(Data_In,VAR6)) ,
                         equality(StrangeSamples_1 , 0)) ,
             implication((VAR7 is Q2-Iqr/2,VAR8 is
Q2+Iqr/2,numberRange(VAR7,VAR8,VAR9),nonMembership(Data_In,VAR9)) ,
                        (VAR10 is StrangeSamples + 1,equality(StrangeSamples_1 ,
VAR10)))) ) ,
implication( (VAR11 is MaxSize*85/100,(SizeAfter < VAR11 or equality(SenseFault ,
t))) ,
             equality(StrangeSamples_1 , StrangeSamples) ) ,
STATE_1 = dl([This,
              SampleWindow,
              MaxSize,
              Oldest,
              Newest,
              Q1,
              Q2,
              Q3,
              Iqr,
              MinConsecutiveStrangeSamples,
              StrangeSamples_1,
              SenseFault,
              Factory|TAIL],REST) ,
writeEnd( REF , STATE_1 ) ,
! .

consumptionEstimatorImplNewDataImpl_3( REF , Data_In , SizeAfter ) :-
consumptionEstimatorImplNewDataImpl_3_1( REF , Data_In , SizeAfter ) .

consumptionEstimatorImplNewDataImpl_4_1( REF , SizeAfter , W , Np ) :-
readStart( REF , dl([This,
                     SampleWindow,
                     MaxSize,
                     Oldest,
                     Newest,
                     Q1,
                     Q2,
                     Q3,
                     Iqr,
                     MinConsecutiveStrangeSamples,
                     StrangeSamples,
                     SenseFault,
                     Factory|TAIL],REST) ) ,
inorder( SampleWindow , VAR1 ) ,
equality( W , VAR1 ) ,
VAR2 is 25/100*SizeAfter,
VAR3 is 50/100*SizeAfter,
VAR4 is 75/100*SizeAfter,
sequenceBrackets( [VAR2,VAR3,VAR4] , VAR5 ) ,
equality( Np , VAR5 ) ,
readEnd( REF ) ,
! .

computeQ( W_In , Np_In , Q_Out ) :-
implication( not(natural(Np_In)) , (VAR1 is ceiling(Np_In),
                                    functionApplication(W_In,VAR1,VAR2),
                                    equality(S,VAR2),
                                    getValue(S,Q_Out)) ) ,
implication( natural(Np_In) , (functionApplication(W_In,Np_In,VAR3),
```

```
                                equality(S1 , VAR3),
                                VAR4 is Np_In + 1,
                                functionApplication(W_In,VAR4,VAR5),
                                equality(S2 , VAR5),
                                getValue(S1,V1),
                                getValue(S2,V2),
                                VAR6 is (V1+V2)/2,
                                equality(Q_Out , VAR6)) ) .


consumptionEstimatorImplNewDataImpl_4_2( W , Np , ComputedQ ) :-
implication( (sequenceBrackets([],VAR1),equality(Np,VAR1)) ,
             (sequenceBrackets([],VAR2),
             equality(ComputedQ,VAR2)) ) ,
implication( (sequenceBrackets([],VAR3),inequality(Np,VAR3)) ,
             (head(Np,VAR4),
             equality(Nph,VAR4),
             tail(Np,VAR5),
             equality(TNp,VAR5),
             computeQ(W,Nph,Qh),
             consumptionEstimatorImplNewDataImpl_4_2(W,TNp,TComputedQ),
             sequenceBrackets([Qh],VAR6),
             concatenation(VAR6,TComputedQ,VAR7),
             equality(ComputedQ,VAR7)) ) .


consumptionEstimatorImplNewDataImpl_4_3( REF , ComputedQ , SizeAfter , Q2_Out , Iqr_Out
) :-
readStart( REF , dl([This,
                     SampleWindow,
                     MaxSize,
                     Oldest,
                     Newest,
                     Q1,
                     Q2,
                     Q3,
                     Iqr,
                     MinConsecutiveStrangeSamples,
                     StrangeSamples,
                     SenseFault,
                     Factory|TAIL],REST) ) ,
functionApplication( ComputedQ , 1 , VAR1 ) ,
equality( Q1_1 , VAR1 ) ,
functionApplication( ComputedQ , 2 , VAR2 ) ,
equality( Q2_1 , VAR2 ) ,
functionApplication( ComputedQ , 3 , VAR3 ) ,
equality( Q3_1 , VAR3 ) ,
functionApplication( ComputedQ , 3 , VAR4 ) ,
functionApplication( ComputedQ , 1 , VAR5 ) ,
VAR6 is VAR4 - VAR5 ,
equality( Iqr_1 , VAR6 ) ,
implication( (StrangeSamples >= MinConsecutiveStrangeSamples,
             VAR7 is MaxSize*85/100,
             SizeAfter >= VAR7) ,
             equality( SenseFault_1 , t ) ) ,
implication( (StrangeSamples < MinConsecutiveStrangeSamples or (VAR8 is
MaxSize*85/100,SizeAfter < VAR8)) ,
             equality( SenseFault_1 , f ) ) ,
equality( Q2_Out , Q2_1 ) ,
```

```
equality( Iqr_Out , Iqr_1 ) ,
STATE_1 = dl([This,
              SampleWindow,
              MaxSize,
              Oldest,
              Newest,
              Q1_1,
              Q2_1,
              Q3_1,
              Iqr_1,
              MinConsecutiveStrangeSamples,
              StrangeSamples,
              SenseFault_1,
              Factory|TAIL],REST) ,
writeEnd( REF , STATE_1 ) ,
! .

consumptionEstimatorImplNewDataImpl_4( REF , SizeAfter , Q2_Out , Iqr_Out ) :-
consumptionEstimatorImplNewDataImpl_4_1( REF , SizeAfter , W , Np ) ,
consumptionEstimatorImplNewDataImpl_4_2( W , Np , ComputedQ ) ,
consumptionEstimatorImplNewDataImpl_4_3( REF , ComputedQ , SizeAfter , Q2_Out , Iqr_Out
) .


isSenseFault( reference(ID,consumptionEstimatorImpl) , SenseFault_Out ) :-
consumptionEstimatorImplIsSenseFaultImpl( reference(ID,consumptionEstimatorImpl) ,
SenseFault_Out ) .

consumptionEstimatorImplIsSenseFaultImpl( REF , SenseFault_Out ) :-
consumptionEstimatorImplIsSenseFaultImpl_1( REF , SenseFault_Out ) .

consumptionEstimatorImplIsSenseFaultImpl_1( REF , SenseFault_Out ) :-
readStart( REF , dl([This,
                     SampleWindow,
                     MaxSize,
                     Oldest,
                     Newest,
                     Q1,
                     Q2,
                     Q3,
                     Iqr,
                     MinConsecutiveStrangeSamples,
                     StrangeSamples,
                     SenseFault,
                     Factory|TAIL],REST) ) ,
equality( SenseFault_Out , SenseFault ) ,
readEnd( REF ) ,
! .
```

7.7.7 La teoria logica ConsumptionEstimatorFactory

```
% ConsumptionEstimatorFactory.pl

:- parents( 'ModelloDelDominio/ConsumptionEstimatorImpl.pl' ) .
```

```
% dl( [This,SampleFactory|REST] , REST )


consumptionEstimatorFactoryNew( reference(ID,consumptionEstimatorFactory) ,
                                                    SampleFactory_In) :-
newStart( reference(ID,consumptionEstimatorFactory) ) ,
consumptionEstimatorFactoryNewImpl( dl([This,
                                        SampleFactory|REST],REST) ,
                                    reference(ID,consumptionEstimatorFactory) ,
                                    SampleFactory_In ) ,
newEnd( reference(ID,consumptionEstimatorFactory) , dl([This,
                                                        SampleFactory|REST],REST) ) .
consumptionEstimatorFactoryNewImpl( STATE_1,
                                    This_In ,
                                    SampleFactory_In ) :-
consumptionEstimatorFactoryNewImpl_1( STATE_1,
                                      This_In ,
                                      SampleFactory_In ) .

consumptionEstimatorFactoryNewImpl_1_1( STATE_1,
                                        This_In ,
                                        SampleFactory_In ) :-
extension( consumptionEstimatorFactory , This_In ) ,
workingReference( SampleFactory_In ) ,
classInitImpl( STATE_1 ,
               This_In ) .

consumptionEstimatorFactoryNewImpl_1_2( dl([This_1,
                                            SampleFactory_1|TAIL],REST),
                                        SampleFactory_In ) :-
equality( SampleFactory_1 , SampleFactory_In ) .

consumptionEstimatorFactoryNewImpl_1( STATE_1,
                                      This_In ,
                                      SampleFactory_In ) :-
consumptionEstimatorFactoryNewImpl_1_1( STATE_1,
                                        This_In ,
                                        SampleFactory_In ) ,
consumptionEstimatorFactoryNewImpl_1_2( STATE_1,
                                        SampleFactory_In ) .




consumptionEstimatorFactoryDelete( reference(ID,consumptionEstimatorFactory) ) :-
readStart( reference(ID,consumptionEstimatorFactory) , dl([This,
                                                           SampleFactory|
REST],REST) ) ,
consumptionEstimatorFactoryDeleteImpl( dl([This,SampleFactory|REST],REST) ) ,
deleteEnd( reference(ID,consumptionEstimatorFactory) ) ,
! .

consumptionEstimatorFactoryDeleteImpl( STATE ) :-
consumptionEstimatorFactoryDeleteImpl_1( STATE ) .

consumptionEstimatorFactoryDeleteImpl_1( STATE ) :- classFinImpl( STATE ) .
```

```
consumptionEstimatorFactoryGetConsumptionEstimator( reference(ID,consumptionEstimatorFa
ctory) ,
                                                     MaxSize_In ,
                                                    MinConsecutiveStrangeSamples_In ,
                                                    Oc_Out ) :-
strictlyPositiveNatural( MaxSize_In ) ,
strictlyPositiveNatural( MinConsecutiveStrangeSamples_In ) ,
consumptionEstimatorFactoryGetConsumptionEstimatorImpl( reference(ID,consumptionEstimat
orFactory) ,
                                                         MaxSize_In ,

MinConsecutiveStrangeSamples_In ,
                                                          Oc_Out ) .


consumptionEstimatorFactoryGetConsumptionEstimatorImpl( REF ,
                                                         MaxSize_In ,
                                                        MinConsecutiveStrangeSamples_In
,
                                                        Oc_Out ) :-
consumptionEstimatorFactoryGetConsumptionEstimatorImpl_1( REF ,
                                                           MaxSize_In ,

MinConsecutiveStrangeSamples_In ,
                                                           Oc_Out ) .


consumptionEstimatorFactoryGetConsumptionEstimatorImpl_1_1( REF ,
                                                             MaxSize_In ,

MinConsecutiveStrangeSamples_In ,
                                                             Sf ) :-
readStart( REF , dl([This,
                     SampleFactory|TAIL],REST) ) ,
MaxSize_In >= 7 ,
VAR1 is MaxSize_In*5/100 ,
VAR2 is MaxSize_In*15/100 ,
MinConsecutiveStrangeSamples_In > VAR1 ,
MinConsecutiveStrangeSamples_In =< VAR2 ,
equality( Sf , SampleFactory ) ,
readEnd( REF ) ,
! .

consumptionEstimatorFactoryGetConsumptionEstimatorImpl_1_2( MaxSize_In ,

MinConsecutiveStrangeSamples_In ,
                                                             Sf ,
                                                             Oc_Out ) :-
consumptionEstimatorImplNew( Oc_Out , MaxSize_In , MinConsecutiveStrangeSamples_In , Sf
) .
```

```
consumptionEstimatorFactoryGetConsumptionEstimatorImpl_1( REF ,
                                                          MaxSize_In ,

MinConsecutiveStrangeSamples_In ,
                                                          Oc_Out ) :-
consumptionEstimatorFactoryGetConsumptionEstimatorImpl_1_1( REF ,
                                                            MaxSize_In ,

MinConsecutiveStrangeSamples_In ,
                                                            Sf ) ,
consumptionEstimatorFactoryGetConsumptionEstimatorImpl_1_2( MaxSize_In ,

MinConsecutiveStrangeSamples_In ,
                                                            Sf ,
                                                            Oc_Out ) .
```

7.8 Conclusioni e sviluppi futuri

In questo capitolo si è definita la semantica operazionale del linguaggio Concepts Z e si è visto come sia possibile costruire il suo supporto all'esecuzione. Questo completa la definizione del linguaggio e mostra a tutti gli effetti l'esistenza di un nuovo linguaggio per la costruzione del software, ad un livello di astrazione più elevato degli attuali strumenti di programmazione.
Al momento della scrittura il componente Concepts Z M2C è ad uno stato di completamento del 50% e sarà presto disponibile un prototipo funzionante che supporti tutte le caratteristiche esposte in questo capitolo.
Per concludere si vogliono delineare alcuni temi che saranno oggetto di investigazioni future:

- Si è visto come Concepts Z costituisca a tutti gli effetti un nuovo linguaggio per la costruzione di sistemi software più astratto degli strumenti attualmente disponibili. Il linguaggio è stato costruito prendendo le basi dalla notazione Z. Come si è visto questo introduce complessità nella realizzazione del componente che si occupa della generazione automatica di codice. Tuttavia la scelta ha consentito una definizione precisa dei costrutti del linguaggio che traggono il loro fondamento da una notazione formale di ampio uso. Viene da chiedersi a questo punto se sia possibile andare oltre Z, ossia si valuta la possibilità di introdurre una sintassi concreta ad hoc per la notazione Concepts Z, eliminando tutte le problematiche dovute alla definizione dei suoi costrutti tramite il linguaggio Z. Questo

consentirebbe una più agevole specifica dei concetti tramite l'introduzione di una sintassi adeguata e una più semplice costruzione del generatore vista la possibilità di semplificare la sintassi astratta del linguaggio.

- Si è visto come Concepts Z consenta di costruire una serie di modelli a granularità crescente per il sistema supportando tutte le fasi del processo di sviluppo. Completato un modello del sistema, viene da chiedersi se sia possibile dimostrare, auspicabilmente in modo automatizzato, che esso soddisfa tutti i vincoli imposti dal modello individuato nella fase precedente. Ciò consentirebbe di provare con certezza, al termine della fase di progetto, di aver sintetizzato un sistema conforme ai requisiti iniziali. Ciò costituirebbe uno strumento di verifica di validità maggiore degli attuali codici di test.
- Nel paragrafo 4 si è utilizzato Concepts Z per definire una base concettuale a disposizione dell'analista. E' possibile sviluppare tali concetti fino alla costituzione di una serie di librerie pronte per l'uso che aumentano considerevolmente le capacità del linguaggio
- Si è visto come l'output del generatore di codice presenti, in certa misura, margini di miglioramento in termini di efficienza. Sarebbe possibile considerare l'output del componente M2C come un formato intermedio, soggetto ad ulteriori elaborazioni da parte di un back-end di ottimizzazione analogo a quello presente nei moderni compilatori per linguaggi di programmazione di alto livello.

Ringraziamenti

Sono ormai al termine della mia esperienza universitaria. Se mi guardo indietro non posso che scorgere una persona radicalmente diversa da ciò che sono ora. Si è trattato di un cammino lungo e non privo di ostacoli. Molte sono state le persone che mi hanno segnato, cambiando il mio modo di vedere le cose o di agire. Sono sempre stato sinceramente convinto che quando si intraprende qualcosa, o la si fa per bene, o non ha alcun senso intraprenderla. “Bene” e “Giusto” a volte sono difficili da discernere, soprattutto in un mondo al contrario.

Alla mia famiglia va il primo ringraziamento, per avermi sostenuto con ogni mezzo possibile fino al traguardo, con sacrifici difficili da rendere in poche righe. Dunque grazie Domenica, grazie Angelo, grazie Giacomo, grazie ai nonni Elio Ada e Angelina e a tutti gli altri parenti che mi hanno dato il loro aiuto in questo lungo periodo.

Un ringraziamento particolare va anche al mio medico e amico di famiglia, l'aiuto e i consigli da lui ricevuti sono andati ben oltre i suoi doveri come medico generico. Se oggi raggiungo questi risultati è anche grazie a lui. Dunque grazie Aurelio.

Un altro ringraziamento va ai miei piccoli allievi, la loro costante allegria, voglia di correre e imparare divertendosi, mi rasserena ogni volta che metto piede in palestra e mi stimola a ricercare modi sempre nuovi ed efficaci per insegnare la pallavolo.

Un'altra persona che vorrei ringraziare è il mio Maestro di arti marziali, sempre pronto a mostrarmi le cose da un'altra ottica ed a ricordarmi che la correttezza assoluta è una parola senza significato nella realtà in cui viviamo. Dal suo modo sincero e originale di insegnarmi e cercare di migliorarmi non posso che prendere esempio. Dunque grazie Alberto.

Durante la mia carriera universitaria ho incontrato molti docenti, ad alcuni dei quali sento di dover dare un ringraziamento particolare, poiché la loro onestà nello svolgere il loro compito e nell'insegnare ai ragazzi nel modo migliore possibile, mi ha consentito di giungere dove sono ora. A loro va la mia stima, per avermi trasmesso nel modo più onesto e chiaro possibile la loro conoscenza e comprensione della materia. Dunque grazie professoressa Mello, grazie professor Laschi.

Ultimo ma non meno importante, vorrei ringraziare il mio relatore. Oltre a quanto detto sopra a lui vanno meriti aggiuntivi, in quanto senza la sua disponibilità, pazienza e comprensione questa tesi non sarebbe mai esistita. Dunque grazie professor Denti.

Riferimenti Bibliografici

1. E.W. Dijkstra. *Structured Programming*. Academic Press New York 1972. Notes on StructuredProgramming.
2. ISO/IEC 19505 , 19502 , 19507
3. ISO/IEC 13568
4. B. Potter , J. Sinclair , D. Till. *An Introduction to Formal Specification and Z*. 2nd edition Prentice Hall
5. P. Malik , M. Utting. *CZT: A Framework for Z Tools*. The University of Waikato, Hamilton, New Zealand
6. S. Stepney , F. Polack , I. Toyn. *A Z Patterns Catalogue I*. University of York.
7. A. Hall. *Using Z as a Specification Calculus for Object-Oriented Systems*. Praxis plc.
8. A. Hall. *Specifying and Interpreting Class Hierachies in Z*. Praxis Bath UK
9. M. Utting , S. Wang. *Object Orientation without Extending Z*. The University of Waikato, Hamilton, New Zealand .
10. N. Amàlio , F. Polack , S. Stepney. *An Object-Oriented Structuring for Z based on Views*. Department of Computer Science, University of York.
11. E.Gamma , R. Helm , R. Johnson , J. Vlissides. *Design Patterns: elementi per il riuso di software a oggetti* . Pearson 2002
12. R. Duke , P. King , G.A. Rose , G. Smith. *The Object-Z specification Language: Version 1*. Technical Report 91-1, The University of Queensland, St. Lucia 4072, Australia , 1991
13. T. Stahl , M. Voelter. *Model-Driven Software Development: technology engineering management*. Wiley 2006
14. B.Liskov , J. Wing. *A behavioral notion of subtyping*. ACM Transactions on Programming Languages adn Systems , 1994
15. K. K. Dhara , G.T. Leavens. *Forcing behavioural subtyping through specification inheritance*. IEEE Computer Society editor ICSE18 , 1996
16. B. Meyer. *The Eiffel Language*. Prentice Hall 1992
17. B. Meyer. *Object Oriented Software Construction*. Prentice Hall 1997
18. N. Amàlio. *Generative frameworks for rigorous model-driven development*. PhD thesys Department of Computer Science University of York , 2006
19. J. Rumbaugh , I. Jacobson , G. Booch. *Unified Modeling Language Reference Manual*. Addison Wesley Object Technology Series 1998
20. K. Beck. *Extreme Programming Explained*. Addison Wesley 2000
21. F. Buschmann , K. Henney , D.C. Schmidt. *Pattern Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing*. Wiley 2007
22. P. Ancillotti , M.Boari. *Programmazione concorrente e distribuita*. McGraw-Hill 2007
23. G. Coulouris , J. Dollimore , T. Kindberg. *Distributed Systems Concepts and Design*. 4th edition Addison Wesley 2005
24. R.C. Martin. *The Dependency Inversion Principle*. C++ Report Engineering Notebook May 1996
25. M. Bramanti. *Calcolo delle Probabilità e Statistica*. Progetto Leonardo
26. P.T. Breuer , J.P. Bowen. *Towards Correct Executable Semantics for Z*. Proc. 8th Z Users Workshop (ZUM94), Workshops in Computing , Cambridge , 1994 Springer-Verlag.
27. L. Sterling , P. Ciancarini , T. Turnidge. *On the animation of “not executable”*

specifications by Prolog. International Journal on Software Engineering and Knowledge Engineering 6 , 1996
28. ISO/IEC 13211-1 , 13211-2
29. L. Console , E.Lamma , P. Mello , M. Milano. *Programmazione Logica e Prolog.* Nuova edizione UTET
30. K. Arnold , J. Gosling , D. Holmes. *The Java Programming Language*. 4th edition Addison Wesley Professional
31. E. Denti. *tuProlog Manual*. tuProlog version 2.8 Alma Mater Studiorum Università di Bologna Italy
32. M. Utting , I. Toyn , J. Sun , A. Martin , J.S.Dong , N. Daley , D. Currie. *ZML: XML Support for Standard Z*. ZB 2003: Formal Specification and Development in Z and B: Third International Conference of B and Z Users, Turku , Finland Proceedings Springer-Verlag Heidelberg
33. Eclipse.org . *Eclipse Modeling Framework* . http://www.eclipse.org/emf
34. A. Appel. *Modern Compiler Implementation in Java*. 2nd edition Cambridge University Press 2002