UNIVERSITÀ
DI PARMA

DIPARTIMENTO DI SCIENZE
MATEMATICHE, FISICHE E INFORMATICHE

Corso di Laurea in Informatica

Tesi di Laurea

Generazione di programmi Java da specifiche formali Z tramite la libreria JSetL

Relatore:
Prof. Gianfranco Rossi

Candidato:
Matteo Mordonini

Anno Accademico 2019/2020

Indice

Introduzione

Lo scopo di questo lavoro consiste nell'individuare una procedura, non ambigua, per ottenere un programma Java funzionante, partendo da una specifica formale descritta in notazione Z. Il programma risultante dovrà rispecchiare la specifica di parteza "animandola", cioè dovrà permettere di eseguire operazioni su determinate variabili, in accordo con gli insiemi dello stato e delle operazioni descritti dalla specifica formale.

Una specifica formale utilizza nozioni matematiche per descrivere le proprietà che un sistema software deve possedere, senza stabilire a priori il modo con il quale ottenerle. In sostanza, i dettagli implementativi non vengono menzionati, lasciando libertà totale al programmatore circa le tecnologie software da utilizzare per realizzare il sistema. Il compito della specifica è di descrivere *cosa* il sistema deve fare senza stabilire il *come*.

La notazione Z è un linguaggio di specifica formale i cui aspetti chiave sono la teoria degli insiemi e i predicati logici. Una specifica descritta in notazione Z è da intendere come una macchina a stati composta da un insieme degli stati possibili del sistema e da un insieme delle operazioni applicabili sullo stato attuale della macchina.

Per semplificare la traduzione da notazione Z a Java, si farà uso della libreria JSetL, sviluppata all'Università di Parma, la quale permette di programmare in Java usando i paradigmi della programmazione dichiarativa. I costrutti di JSetL sono di natura molto simile a quelli utilizzati in Z, poiché entrambi fanno forte uso di teoria degli insiemi e predicati logici.

Le fondamenta di questo lavoro si basano sulla tesi di Lorenzo De Santis[1] il quale ha descritto regole di traduzione da Z a Java + JSetL e scritto una libreria, da considerarsi un'estensione di JSetL, per semplificare l'automazione di questa operazione. Il nostro intento sarà quello di portare avanti e migliorare il lavoro da lui svolto. Gli obiettivi principali sono quelli del raggiungimento di una minore ambiguità nella procedura di traduzione e di un consolidamento della libreria jsetl.z sviluppata da De Santis.

Capitolo 1

Notazione Z

La notazione Z è un linguaggio di specifica di sistemi software basato sulla teoria degli insiemi e i predicati logici. Un sistema viene modellato in termini di tipi di dato matematici, i quali non sono orientati ad una rappresentazione informatica, ma rispondono a determinate leggi matematiche consentendo di ragionare sul comportamento del sistema che si vuole descrivere.

Come dicevamo nell'introduzione, una specifica descritta tramite Z è concettualmente paragonabile ad una macchina a stati descritta tramite teoria degli insiemi e predicati logici. Precisamente, una specifica Z è composta da:

1. **Un insieme di stati** contenenti variabili di stato aventi un nome ed un tipo, perciò l'insieme di stati consiste di tutti i possibili valori che possono essere assunti da queste variabili.

2. **Un insieme di operazioni** le quali possono modificare lo stato corrente, oppure interrogarlo. Le prime descrivono il passaggio da uno stato verso un altro, mentre le seconde descrivono la sola lettura del valore di determinate variabili senza alterare lo stato corrente. Funzionamento analogo alle query di aggiornamento e di selezione all'interno di un database.

Nelle seguenti sezioni, verranno descritti gli elementi della notazione Z [2, 6] su cui si concentrerà maggiormente la nostra attenzione nella procedura di traduzione verso Java + JSetL.

1.1 Insiemi definiti per enumerazione

Un insieme definito per enumerazione non è altro che un elenco di costanti, utile e versatile per introdurre un aspetto del sistema senza legarlo a nessun concetto matematico o di programmazione. Eccone un esempio che potrebbe descrivere il

tipo di una variabile il cui scopo è quello di mandare un messaggio all'utente per segnalargli la presenza di un eventuale errore:

$$MESSAGE ::= ok \mid fail \mid error$$

1.2 Definizioni assiomatiche

Una definizione assiomatica in Z è un costrutto che permette di specificare variabili, e relativi vincoli su di esse, all'infuori di uno schema. Queste variabili sono accessibili all'interno di qualsiasi schema e vengono inserite per l'appunto nelle definizioni assiomatiche perché non sono vere e proprie variabili di stato, ma sono utili per il funzionamento del sistema.

Eccone un esempio che potrebbe descrivere il fatto che si abbia un numero massimo di download effettuabili giornalmente da una piattaforma di streaming musicale:

$$maxDownloadPerDay : \mathbb{Z}$$
$$\hline$$
$$maxDownloadPerDay = 10$$

1.3 Schemi

Il componente principale di una specifica scritta in Z è il cosiddetto schema, costrutto usato per descrivere lo stato, le operazioni su di esso oppure un nuovo tipo di dato.

Uno schema è identificato da un nome ed ha due componenti principali: una parte dichiarativa, dove vengono dichiarate variabili o riferimenti ad altri schemi, ed una parte predicativa, dove vengono dichiarati predicati logici usando le variabili introdotte nella parte dichiarativa o nelle definizioni assiomatiche. I predicati logici, che chiameremo *vincoli*, sono da considerarsi in congiunzione logica tra loro. Le variabili dichiarate negli schemi possono essere anche decorate con i simboli "′", "?", "!" posposti al loro nome, il che vuole dire, rispettivamente, che sarà il valore della variabile nello stato successivo, che si tratta di una variabile di input, che si tratta di una variabile di output.

Uno schema può rappresentare un nuovo tipo dato oppure un'operazione sullo stato. Quelli della prima tipologia li chiameremo *schemi di struttura*, mentre i secondi *schemi di azione*.

1.3.1 Schemi nella parte dichiarativa

All'interno della parte dichiarativa di uno schema, oltre a variabili, è possibile indicare anche il nome di altri schemi della specifica. Il significato di questa scrittura è che faranno parte, dello schema contenitore, tutte le variabili e tutti i vincoli presenti nello schema così indicato.

La citazione di uno schema può essere decorata anch'essa tramite i simboli "Ξ", "Δ" e "$'$". In generale, se *NomeSchema* è così definito:

$$
\begin{array}{|l}
\hline
NomeShema \\
\hline
variabile_1 : TIPO_1 \\
variabile_2 : TIPO_2 \\
... \\
variabile_n : TIPO_n \\
\hline
vincolo_1 \\
vincolo_2 \\
... \\
vincolo_n \\
\hline
\end{array}
$$

allora "$\Xi NomeSchema$" sarà:

$$
\begin{array}{|l}
\hline
\Xi NomeShema \\
\hline
\Delta NomeSchema \\
\hline
variabile_1 = variabile'_1 \\
variabile_2 = variabile'_2 \\
... \\
variabile_n = variabile'_n \\
\\
\hline
\end{array}
$$

invece "$\Delta NomeSchema$" è così definito:

$$
\begin{array}{|l}
\hline
\Delta NomeSchema \\
\hline
NomeSchema \\
NomeSchema' \\
\hline
\end{array}
$$

infine, "$NomeSchema'$" è così definito:

$NomeSchema'$

$variabile'_1 : TIPO_1$
$variabile'_2 : TIPO_2$
...
$variabile'_n : TIPO_n$

$\sim$ I vincoli di "NomeSchema",
ma tutte le variabili al loro interno sono decorate con "′" $\sim$

1.3.2 Operazione tra schemi

Per rendere più sintetica una specifica, è possibile indicare uno schema come il risultato di un'operazione fra schemi:

$$Schema == Schema_1 \, \pi \, Schema_2$$

Dove π è uno tra i seguenti operatori logici:

- $\vee$
- $\wedge$

Oppure nella forma:

$$Schema == \neg Schema_1$$

Nel primo caso, il risultato è uno schema in cui nella parte dichiarativa si ha l'unione delle variabili dichiarate in entrambi gli schemi, mentre nella parte dichiarativa si ha l'operazione definita dal connettivo logico "π" tra i vincoli dei due schemi.

Esempi completi di specifiche Z verranno mostrati nel capitolo 4.

Capitolo 2

Libreria JSetL

La libreria Java *JSetL*[3, 4, 5] permette di programmare usando i paradigmi della programmazione dichiarativa sfruttando le potenzialità della programmazione orientata agli oggetti. Il suo meccanismo interno non è dissimile da un interprete CLP (Constraint Programming). Infatti utilizza vincoli su variabili logiche risolvendoli tramite un risolutore di vincoli (solver). I vincoli utilizzabili sono relativi alle operazioni della teoria degli insiemi e sui numeri interi. Gli insiemi realizzabili possono essere anche parzialmente specificati e l'utente può definire nuovi vincoli.

Ogni oggetto logico è un'istanza della classe **LObject**, la quale fornisce metodi comuni per ognuno di essi. LObject ha due sottoclassi: **LVar** e **LCollection**, le quali consistono, rispettivamente, in variabili logiche di qualunque tipo e collezioni di valori di qualsiasi tipo. Entrambe hanno ulteriori sottoclassi

Di seguito andremo a descrivere brevemente alcune di esse oltre alle classi **Constraint** e **Solver**.

2.1 LVar

La classe LVar rappresenta variabili logiche che possono assumere un valore tramite l'utilizzo di vincoli. Quando il dominio di una variabile logica sarà ristretto ad un singolo valore, il **Solver** glielo assegnerà e potremo considerarla *legata* ad esso. Sarà considerata *slegata* fino a quel momento. Una volta assegnato, il valore di una variabile logica non sarà più modificabile.

Una variabile logica possiede anche un nome esterno costituito da un oggetto di tipo *java.lang.String* il quale può essere utile nelle operazioni di stampa.

2.2 IntLVar

Questa classe modella variabili logiche rappresentanti i numeri interi, è quindi un caso speciale di variabile logica, essendo sostanzialmente una LVar vincolata ad avere un valore appartenente al dominio dei numeri interi. Il dominio di una IntLVar può essere immediatamente ristretto nel costruttore oppure tramite vincoli applicati successivamente.

Un oggetto di tipo IntLVar può invocare metodi specifici per i numeri interi, come ad esempio le operazioni aritmetiche di base, le quali ritornano un oggetto IntLVar con associato un vincolo che lega il suo valore al risultato dell'operazione stessa.

2.3 LSet

Un oggetto s della classe LSet è un insieme logico il cui valore è un numero n di oggetti, con $n \geq 0$, e una coda, la quale è anch'essa un insieme logico che rappresenta il resto di s e può essere vuoto oppure slegato. Un LSet può essere fondamentalmente in tre stati:

- vuoto, se non ha elementi al suo interno
- non vuoto, avendo almeno un elemento, e con la coda vuota, il che significa che è un *insieme chiuso*
- non vuoto, avendo almeno un elemento, e con la coda slegata, il che significa che è un *insieme aperto*

2.4 IntLSet

Si tratta di una sottoclasse di LSet, quindi rappresenta anch'essa un insieme di oggetti logici, con la peculiarità che questi oggetti sono numeri interi o istanze di *IntLVar*. Dato che i numeri interi sono di fondamentale importanza per la notazione Z, questa classe è molto utile perché permette di maneggiarli più facilmente rispetto all'eventuale utilizzo di *LSet*.

2.5 LMap

Un oggetto *LMap* rapperesenta una funzione parziale, i cui elementi sono coppie logiche (*LPair*) con la peculiarità che non possono contenere due coppie aventi il primo elemento uguale e il secondo elemento diverso. L'insieme di appartenenza

degli oggetti del primo elemento viene chiamato dominio, mentre quello relativo al secondo elemento viene detto rango.

Le funzioni parziali, sono relazioni binarie molto utilizzate in Z, quindi LMap è una classe molto utile per implementarle.

2.6 Constraint

La classe *Constraint* rappresenta, per l'appunto, i vincoli, cioè le operazioni applicabili a variabili e collezioni logiche. Saranno fondamentali per tradurre i predicati logici presenti nelle parti predicative delle definizioni assiomatiche e degli schemi presenti nelle specifiche espresse in notazione Z.

Possono essere atomici, composti o di negazione. Un vincolo atomico rappresenta, per esempio, un'operazione sugli interi o sugli insiemi come l'uguaglianza o l'unione. Un vincolo composto implementa un connettivo logico, andando a mettere, per esempio, in congiunzione logica tra di loro due vincoli. Un vincolo di negazione effettua la negazione logica di un altro vincolo.

2.7 Solver

Un oggetto *Solver* ha il compito di accumulare vincoli e verificare la loro soddisfacibilità, andando ad assegnare un valore a tutte le variabili coinvolte, nel caso sia possibile.

Al suo interno il solver utilizza un accumulatore di vincoli (*constraint store*) che può essere popolato dall'utente, il quale rappresenta una congiunzione logica di tutti i vincoli inseriti al suo interno. Nel caso si voglia cercare una soluzione per questi vincoli, il Solver andrà a verificarne la soddisfacibilità e, se trovata, proporrà, una ad una, tutte le soluzioni possibili.

Traducendo uno schema rappresentante un'operazione sullo stato della macchina, andremo a creare un metodo Java il quale andrà a definire variabili e vincoli rispecchiando la logica dello schema stesso, per poi andare a risolverli tramite un Solver.

Capitolo 3

Il package jsetl.z

Per semplificare la traduzione dalla notazione Z verso Java, è stato realizzato il package *jsetl.z* [1], il quale sarà incluso, insieme a JSetL, in ogni programma Java derivante da questa procedura di traduzione.

Come già detto, uno scopo di questa traduzione consiste nell'"animare" una specifica descritta in notazione Z. In tal senso, lo scopo di questo package è l'automatizzare quelle operazioni che devono essere sempre effettuate indipendentemente dalla specifica di partenza. In questo modo, nella procedura di traduzione, ci si concentrerà principalmente sulla mappatura dei costrutti Z nei relativi costrutti JSetL.

Partendo da una specifica Z contenente insiemi definiti per enumerazione, definizioni assiomatiche e schemi, si otterrà, tramite la procedura di traduzione, una classe Java contenente determinati attributi e metodi. Gli attributi sono derivati da insiemi definiti per enumerazione, definizioni assiomatiche e schemi definenti lo stato, mentre i metodi sono derivati dagli schemi che rappresentano le operazioni sullo stato. La libreria, invece, si occupa dell'interazione con l'utente, fornendogli la possibilità di eseguire l'operazione desiderata, della gestione dei vincoli di stato e dei vincoli dinamici, ovvero quei vincoli validi solamente durante l'esecuzione di un determinato schema.

I componenti principali del package sono i seguenti:

- **ZDocument**, classe astratta più importante perché ha il compito di gestire lo stato del sistema, conservando i vincoli di invarianza di stato e applicando i vincoli dinamici.

- **ZInterface**, interfaccia che definisce i metodi per l'interazione dell'utente con il sistema.

- **ZDynamicState**, interfaccia che definisce i metodi per implementare i vincoli dinamici del sistema.

- **ZStaticState**, interfaccia che definisce i metodi per implementare i vincoli statici (invarianti) del sistema.

3.1 ZDocument

Il compito della classe astratta *ZDocument* consiste nel mantenere in memoria gli aspetti statici del sistema, ovvero le sue invarianti, ed applicare su di esso le operazioni descritte dagli schemi della specifica.

Le invarianti del sistema sono variabili accessibili da qualsiasi schema, quindi da qualsiasi metodo nel codice Java risultante, e determinati vincoli che sono sempre validi in qualunque stato.

Gli attributi principali della classe sono:

- un oggetto *ZInterface*, con il compito di gestire l'interfaccia utente.
- un oggetto *ZDynamicState*, con il compito di gestire i vincoli dinamici e aggiornare i valori delle variabili di stato che dovessero cambiare a seguito dell'esecuzione di uno schema.
- un oggetto *ZStaticState*, con il compito di gestire i vincoli statici.
- un oggetto *Solver*, con il compito di risolvere i vincoli ad ogni esecuzione di uno schema.
- un oggetto *Constraint*, con il compito di accumulare i vincoli temporanei dei vari schemi. Sarà utilizzato dal solver.
- quattro liste, che contengono rispettivamente i riferimenti agli insiemi definiti per enumerazione e alle variabili temporanee di input, di modifica e di output presenti nei vari schemi.

I metodi principali, invece, sono:

- **protected void tempPost(Constraint c)**, invocato nei metodi rappresentanti gli schemi, ha il compito di aggiungere il vincolo preso in input all'interno dell'accumulatore dei vincoli temporanei.
- **public void statPost(Constraint c)**, invocato nel costruttore della classe Java risultante dalla traduzione, aggiunge il vincolo preso in input all'accumulatore dei vincoli statici.
- **protected void execute()**, avvia il processo di input ed output per poi eseguire la risoluzione dei vincoli temporanei e statici. Sarà l'ultima istruzione di ogni metodo rappresentante uno schema di azione.

- **protected void askVar(Object o)**, popola la lista delle variabili temporanee di input, i cui valori andranno chiesti all'utente durante l'esecuzione del metodo rappresentante uno schema di azione.
- **protected void ansVar(Object o)**, popola la lista delle variabili temporanee di output, i cui valori saranno stampati al termine dell'esecuzione dei metodi rappresentanti uno schema di azione.
- **protected void prmVar(Object o, Object oPrm)**, tiene traccia del fatto che il valore della variabile *o* dovrà essere eguagliato a quello della variabile *oPrm* quando tutti i vincoli di uno schema saranno valutati. Consiste nell'implementazione del concetto delle variabili primed della notazione Z, le quali simboleggiano il cambiamento di stato, andando a tenere traccia dell'eventuale nuovo valore assunto da una variabile al termine di una operazione.

3.2 ZInterface

L'interfaccia *ZInterface* ha il compito di gestire l'interazione tra l'utente e il sistema derivante dalla specifica Z. L'interfaccia fa sì che all'avvio del programma, l'utente si trovi a scegliere quale operazione effettuare, ovvero quale metodo eseguire tra quelli risultanti dalla traduzione degli schemi di azione. Al termine di ogni operazione, l'interfaccia riproporrà all'utente l'elenco di quelle disponibili. L'interfaccia è testuale e l'utente sceglie l'operazione digitandone il nome.

Un secondo momento di interazione si ha durante l'esecuzione dei metodi quando viene chiesto all'utente di inserire il valore di una variabile di input e quando gli verrà restituito in output il valore di una variabile di output.

Nel main Java viene invocato il metodo **public void open(ZDocument doc, String initMethod)** tramite un oggetto della classe *TextInterface*, la quale implementa *ZInterface*. Dato che tutte le specifiche Z hanno uno schema iniziale, il quale conferisce un valore iniziale allo stato del sistema, questo metodo esegue *initMethod* il quale è il metodo rappresentante lo schema iniziale. Successivamente, *initMethod* verrà rimosso dalla lista dei metodi da proporre all'utente. L'esecuzione, all'avvio del sistema, del metodo rappresentante lo schema iniziale, è un'automatismo che è stato aggiunto alla versione originale di jsetl.z, poiché si tratta di un'operazione che deve essere sempre eseguita per prima al fine di rispettare la semantica della specifica Z.

3.3 ZDynamicState e ZStaticState

Come detto in precedenza, le interfacce ZDynamicState e ZStaticState hanno il compito, rispettivamente, di gestire i vincoli dinamici e statici.

Entrambe le interfacce contengono un accumulatore di vincoli i quali vengono popolati, rispettivamente, dai metodi *tempPost* e *statPost*. ZDynamicState ha anche il compito di aggiornare il valore alle variabili di stato aventi la controparte primed ($'$) in un determinato schema.

L'accumulatore dei vincoli dinamici viene svuotato al termine di ogni metodo, mentre quello dei vincoli statici viene popolato nel costruttore e resterà inalterato. Al contrario, quello dei vincoli statici non viene mai svuotato e mantiene i vincoli in forma non risolta, i quali verranno passato al solver ad ogni esecuzione di *execute()*.

Capitolo 4

Procedura di traduzione da Z a Java

La procedura di traduzione che seguirà sarà affiancata da un esempio concreto di specifica Z e per ogni algoritmo sarà mostrato il suo effetto su quest'ultimo. La traduzione, da Z verso un programma funzionante Java + JSetL, sarà divisa in tre passi consecutivi:

1. Manipolazione della specifica, portandola in una forma normalizzata in modo tale da rendere più pratica la traduzione verso java.

2. Traduzione da Z a Java + JSetL in cui i vari componenti della specifica vengono identificati e tradotti nel corrispettivo codice Java.

3. Modifica del codice Java risultante per migliorarne la lettura o rimuovere codice ridondante, senza modificare la semantica del programma.

4.1 Primo esempio di specifica Z

La seguente specifica Z modella un programma volto ad implementare un gioco in cui l'utente deve indovinare un numero generato casualmente. Prevede due modalità di gioco, *facile* e *difficile*, le quali definiscono l'insieme dal quale viene estratto il numero casuale. Lo stato iniziale imposta la modalità facile, successivamente, l'utente può provare ad indovinare, fornendo un numero intero in input, oppure incominciare una nuova partita scegliendo la modalità di gioco desiderata. Una nuova partita implica la generazione di un numero casuale, il quale deve essere differente da quello estratto in precedenza. Quando l'utente proverà ad indovinare, riceverà un messaggio che identificherà un successo o un fallimento.

La specifica:

1. Insiemi definiti per enumerazione

$$MESSAGGIO ::= vittoria \mid riprova$$

$$MODALITA ::= facile \mid difficile$$

2. Definizione assiomatica

$$\begin{array}{|l} limitefacile : \mathbb{Z} \\ limitedifficile : \mathbb{Z} \\ \hline limitefacile = 3 \\ limitedifficile = 5 \end{array}$$

3. Schemi di stato

$$\begin{array}{|l} \textit{PartitaFacile} \\ \hline difficolta : MODALITA \\ casuale : \mathbb{Z} \\ \hline difficolta = FACILE \\ casuale < limitefacile \\ casuale \geq 0 \end{array}$$

$$\begin{array}{|l} \textit{PartitaDifficile} \\ \hline difficolta : MODALITA \\ casuale : \mathbb{Z} \\ \hline difficolta = DIFFICILE \\ casuale < limitedifficile \\ casuale \geq 0 \end{array}$$

4. Schema di stato definito tramite operazione tra schemi

$$Partita == PartitaFacile \lor PartitaDifficile$$

5. Schema di inizializzazione (stato iniziale della macchina)

$$\begin{array}{|l}
\hline
InitPartita \\
\hline
Partita' \\
\hline
PartitaFacile' \\
\hline
\end{array}$$

6. Schema di azione Delta (cambiamento dello stato)

$$\begin{array}{|l}
\hline
NuovaPartita \\
\hline
\Delta Partita \\
difficolta? : MODALITA \\
\hline
difficolta' = difficolta? \\
casuale' \neq casuale \\
\hline
\end{array}$$

7. Schemi d'azione Xi (lettura dello stato)

$$\begin{array}{|l}
\hline
ProvaGiusto \\
\hline
\Xi Partita \\
casuale? : \mathbb{Z} \\
messaggio! : MESSAGGIO \\
\hline
casuale = causale? \\
messaggio! = VITTORIA \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
ProvaSbagliato \\
\hline
\Xi Partita \\
casuale? : \mathbb{Z} \\
messaggio! : MESSAGGIO \\
\hline
casuale \neq causale? \\
messaggio! = RIPROVA \\
\hline
\end{array}$$

8. Schema d'azione Xi definito tramite operazione tra schemi

$$Prova == ProvaGiusto \lor ProvaSbagliato$$

4.2 Terminologia negli algoritmi

Il documento che contiene tutta la specifica Z è chiamato ZDOCUMENT. Della specifica vengono identificati i seguenti componenti: insiemi definiti per enumerazione, definizioni assiomatiche e schemi. Gli schemi sono classificati in 4 categorie: schemi di azione, di stato, di inizializzazione e di struttura. Gli insiemi definiti per enumerazione contengono costanti, le definizioni assiomatiche e gli schemi contengono variabili e vincoli. Lo schema di inizializzazione deve essere unico ed il suo nome deve iniziare con la stringa "Init".

Le operazioni di traduzione saranno descritte tramite uno pseudocodice che farà forte ricorso alla notazione insiemistica e "dot notation". Per avere un livello di astrazione sufficientemente formale, si utilizzeranno:

i seguenti attributi:

- **d**: indica la parte dichiarativa, rispettivamente, di uno schema e di una definizione assiomatica.
- **p**: indica la parte predicativa rispettivamente, di uno schema e di una definizione assiomatica.
- **type**: indica il tipo di una variabile.
- **name / NAME**: indica il nome, rispettivamente, minuscolo e maiuscolo di uno schema, di una variabile, di una costante, o di un insieme definito per enumerazione.
- **operator**: indica l'operatore di una operazione tra schemi.

i seguenti insiemi:

- **Schema**: indica uno schema della notazione zeta.
- **AssDefinition**: indica una definizione assiomatica della notazione zeta.
- **XiSchema**: indica un riferimento, preceduto dalla lettera "Ξ" e all'interno di uno schema, ad un altro schema della specifica.
- **DeltaSchema**: indica un riferimento, preceduto dalla lettera "Δ" e all'interno di uno schema, ad un altro schema della specifica.
- **PrimedSchema**: indica un riferimento, susseguito dal simbolo "′" e all'interno di uno schema, ad un altro schema della specifica.
- **UndecoratedSchema**: indica un riferimento ad un altro schema della specifica.

- **SchemaOperation**: indica un'operazione tra schemi.
- **Constant**: indica una costante di un insieme definito per enumerazione.
- **EnumeratedSet**: indica un insieme definito per enumerazione che verrà utilizzato come tipo di variabili.
- **Var**: indica una variabile all'interno di uno schema o di una definizione assiomatica.
- **Constraint**: indica un vincolo all'interno di uno schema o di una definizione assiomatica.

e le seguenti funzioni:

- **schema(Schema decoratedSchema) : UndecoratedSchema**: ritorna lo schema non decorato a cui fa riferimento *decoratedSchema*.
- **delta(Schema schema) : DeltaSchema**: ritorna lo schema *schema* decorato con "Δ".
- **primed(Schema schema) : PrimedSchema**: ritorna lo schema *schema* decorato con "′".
- **statement(String stm, String scp) : void**: Aggiunge lo statement *stm* nello scope specificato da *scp*, il quale può essere:
  - "CONSTRUCTOR" che indica il costruttore della classe principale
  - "INIT" che indica il metodo init() della classe rappresentate uno schema di struttura.
  - "ACTION" che indica il metodo, all'interno della classe principale, che rappresenta uno schema di azione.
  - "ATTRIBUTE" indica gli attributi della classe principale. Lo statement deve essere una stringa di testo che rappresenti la definizione di un attributo di classe java.
  - "STRUCTURE" indica gli attributi della classe relativa allo schema di struttura che si sta prendendo in considerazione.
  - "MAIN" indica il metodo main della classe principale.

  Dato che questo metodo prende in input uno statement java, viene considerato implicito il simbolo ";" finale.

- **domainConstraint(Var variable) : String**: ritorna il codice java che traduce il vincolo sul tipo della variabile zeta *variable* nel corrispondente vincolo java.
- **zToJavaConstraint(Constraint const) : String**: ritorna il codice java che traduce il vincolo zeta *const* nel corrispondente vincolo java + JSetL.
- **addMethod(String methodName) : void**: aggiunge il metodo *methodName* alla classe principale.
- **newClass(String className) : void** crea una nuova classe java, il cui nome è indicato da *className* e rappresentante uno schema di struttura, avente una parte di codice predefinita.

4.3 Traduzione di tipi e vincoli da Z a JSetL

Nella terminologia che verrà utilizzata negli algoritmi, sono presenti tre elementi che devono essere ulteriormente descritti per poterli implementare in un caso reale. Si tratta dell'attributo "type" e delle funzioni "domainConstraint()" e "zToJava-Constraint()". All'interno delle seguenti regole, saranno presenti alcuni attributi già mostrati in precedenza.

type attributo che indica il tipo di una variabile e viene usato tramite la sintassi *"nomevariabile.type"* e restituisce il suo tipo in JSetL in base alla sua dichiarazione all'interno della specifica Z.

- *IntLVar*: se applicato ad una variabile di tipo intero.

$$\frac{\text{VAR} : \mathbb{Z}}{\texttt{IntLVar}}$$

- *LVar*: se applicato ad una variabile di tipo insieme definito per enumerazione.

$$\frac{\text{VAR} : \text{SET}}{\texttt{LVar}}$$

- *IntLSet*: se applicato ad una variabile di tipo insieme di interi.

$$\frac{\text{VAR} : \mathbb{P}\,\mathbb{Z}}{\texttt{IntLSet}}$$

- *LSet*: se applicato ad una variabile di tipo insieme di non soli interi.

$$\frac{\text{VAR} : \mathbb{P}\,\text{SET}}{\texttt{LSet}}$$

- *LRel*: se applicato ad una variabile di tipo relazione binaria.

$$\frac{\text{VAR} : \text{SET1} \leftrightarrow \text{SET2}}{\texttt{LRel}}$$

- *LMap*: se applicato ad una variabile di tipo funzione parziale.

$$\frac{\text{VAR} : \text{SET1} \nrightarrow \text{SET2}}{\texttt{LMap}}$$

- *Schema*: se applicato ad una variabile il cui tipo è definito da Schema, ovvero uno schema di struttura, quindi il tipo java sarà il nome della classe definita a partire dallo schema di struttura.

$$\frac{\text{VAR} : \text{SCHEMA}}{\texttt{SCHEMA.NAME}}$$

domainConstraint(Var variable) indica il vincolo relativo al dominio della variabile in input. Restituisce un vincolo JSetL in accordo con il tipo Z, della variabile stessa, secondo le regole seguenti.

- tipo definito da un insieme definito per enumerazione:

$$\frac{\text{VAR} : \text{SET}}{\texttt{VAR.name + ".in(" + SET.name + ")"}}$$

- tipo definito da uno schema di struttura:

$$\frac{\text{VAR} : \text{SCHEMA}}{\texttt{VAR.name + ".in(" + SCHEMA.NAME + ".type)"}}$$

- tipo definito dal powerset di un insieme definito per enumerazione:

$$\frac{\text{VAR} : \mathbb{P}\, SET}{\texttt{VAR.name + ".subset(" + SET.name + ")"}}$$

- tipo definito dal powerset di uno schema di struttura:

$$\frac{\text{VAR} : \mathbb{P}\, SCHEMA}{\texttt{VAR.name + ".subset(" + SCHEMA.NAME + ".type)"}}$$

- tipo definito dal powerset dell'insieme dei numeri interi:

$$\frac{\text{VAR} : \mathbb{P}\,\mathbb{Z}}{\texttt{VAR.name + ".subset(new IntLSet())"}}$$

- tipo definito da una funziona parziale, o da una relazione binaria:

VAR : $TIPO1 \nrightarrow TIPO2$ *oppure* VAR : $TIPO1 \leftrightarrow TIPO2$
`VAR.name+".dom("+X+").and("+VAR.name+".ran("+Y+"))"`

  dove `X` è una delle seguenti stringhe in accordo, rispettivamente, alla tipologia di insieme a cui appartiene $TIPO1$:

  - **Schema di struttura**:
    - `X` = TIPO1.NAME + ".type"
  - **Insieme definito per enumerazione**:
    - `X` = TIPO1.name
  - $\mathbb{Z}$:
    - `X` = "new IntLSet()"

  e dove `Y` è una delle seguenti stringhe in accordo, rispettivamente, alla tipologia di insieme a cui appartiene $TIPO2$:

  - **Schema di struttura**:
    - `Y` = TIPO2.NAME + ".type"
  - **Insieme definito per enumerazione**:
    - `Y` = TIPO2.name
  - $\mathbb{Z}$:
    - `Y` = "new IntLSet()"

zToJavaConstraint(Constraint const) ritorna il codice java che traduce il vincolo Z *const* nel corrispondente vincolo java + JSetL. Ogni vincolo Z può essere tradotto in uno o più corrispondenti vincoli JSetL. In Z, un vincolo esprime una relazione fra due espressioni, quindi la sua forma generica è la seguente:

$$ESPRESSIONE1 \quad OPERATORE \quad ESPRESSIONE2$$

I vincoli possono essere combinati tramite l'operatore logico di negazione, gli operatori logici binari di disgiunzione, congiunzione, implicazione e coimplicazione e gli operatori logici ternari di quantificazione universale e quantificazione esistenziale. Il risultato della funzione *zToJavaConstraint()*, in accordo al vincolo Z preso in input, sarà il seguente:

- OPERATORI INSIEMISTICI

– uguaglianza insiemistica:

$$\frac{\text{ESPRESSIONE1} = \text{ESPRESSIONE2}}{\texttt{ESPRESSIONE1 + ".eq(" + ESPRESSIONE2 + ")"}}$$

– disuguaglianza insiemistica:

$$\frac{\text{ESPRESSIONE1} \neq \text{ESPRESSIONE2}}{\texttt{ESPRESSIONE1 + ".neq(" + ESPRESSIONE2 + ")"}}$$

– appartenenza:

$$\frac{\text{ESPRESSIONE1} \in \text{ESPRESSIONE2}}{\texttt{ESPRESSIONE1 + ".in(" + ESPRESSIONE2 + ")"}}$$

– non-appartenenza:

$$\frac{\text{ESPRESSIONE1} \notin \text{ESPRESSIONE2}}{\texttt{ESPRESSIONE1 + ".nin(" + ESPRESSIONE2 + ")"}}$$

– inclusione:

$$\frac{\text{ESPRESSIONE1} \subseteq \text{ESPRESSIONE2}}{\texttt{ESPRESSIONE1 + ".subset(" + ESPRESSIONE2 + ")"}}$$

– inclusione stretta:

$$\frac{\text{ES1} \subset \text{ES2}}{\texttt{ES1 + ".subset(" + ES2 + ").and(" + ES1 + ".neq(" + ES2 + "))"}}$$

- OPERATORI SU RELAZIONI BINARIE

– restrizione di dominio:

$$\frac{\text{SET} \lhd \text{REL}}{\texttt{REL.name + ".dres(" + SET.name + ", " + REL.name +")"}}$$

– restrizione di rango:

$$\frac{\text{REL} \rhd \text{SET}}{\texttt{REL.name + ".rres(" + SET.name + ", " + REL.name +")"}}$$

– anti-restrizione di dominio:

$$\frac{\text{SET} \lhd\!\!\!\!- \text{REL}}{\texttt{REL.name + ".ndres(" + SET.name + ", " + REL.name +")"}}$$

– anti-restrizione di rango:

$$\frac{\text{REL} \rhd \text{SET}}{\texttt{REL.name + ".nrres(" + SET.name + ", " + REL.name +")"}}$$

- OPERATORI ALGEBRICI
  - equazione:

$$\frac{\text{ESPRESSIONE1} = \text{ESPRESSIONE2}}{\texttt{ESPRESSIONE1 + ".eq(" + ESPRESSIONE2 + ")"}}$$

  - disequazione:

$$\frac{\text{ESPRESSIONE1} \neq \text{ESPRESSIONE2}}{\texttt{ESPRESSIONE1 + ".neq(" + ESPRESSIONE2 + ")"}}$$

$$\frac{\text{ESPRESSIONE1} \leq \text{ESPRESSIONE2}}{\texttt{ESPRESSIONE1 + ".le(" + ESPRESSIONE2 + ")"}}$$

$$\frac{\text{ESPRESSIONE1} < \text{ESPRESSIONE2}}{\texttt{ESPRESSIONE1 + ".lt(" + ESPRESSIONE2 + ")"}}$$

$$\frac{\text{ESPRESSIONE1} \geq \text{ESPRESSIONE2}}{\texttt{ESPRESSIONE1 + ".ge(" + ESPRESSIONE2 + ")"}}$$

$$\frac{\text{ESPRESSIONE1} > \text{ESPRESSIONE2}}{\texttt{ESPRESSIONE1 + ".gt(" + ESPRESSIONE2 + ")"}}$$

- OPERATORI LOGICI
  - negazione:

$$\frac{\neg\ \text{VINCOLO}}{\texttt{VINCOLO + ".notTest()"}}$$

  - disgiunzione:

$$\frac{\text{VINCOLO1} \vee \text{VINCOLO2}}{\texttt{VINCOLO1 + ".or(" + VINCOLO2 + ")"}}$$

  - congiunzione:

$$\frac{\text{VINCOLO1} \wedge \text{VINCOLO2}}{\texttt{VINCOLO1 + ".and(" + VINCOLO2 + ")"}}$$

  - implicazione:

$$\frac{\text{VINCOLO1} \Rightarrow \text{VINCOLO2}}{\texttt{VINCOLO1 + ".impliesTest(" + VINCOLO2 + ")"}}$$

– coimplicazione:

$$\frac{\text{V1} \Leftrightarrow \text{V2}}{\texttt{V1+".impliesTest("+V2+").and("+V2+".impliesTest("+V1+"))"}}$$

– quantificazione universale:

$$\frac{\forall \text{VAR: INSIEME} \bullet \text{VINCOLO}}{\texttt{INSIEME + ".forallElems(" + VAR + ", " + VINCOLO + ")"}}$$

– quantificazione esistenziale:

$$\frac{\exists \text{VAR: INSIEME} \bullet \text{VINCOLO}}{\texttt{INSIEME+".forallElems("+VAR+", "+VINCOLO+".notTest()).notTest()"}}$$

Le espressioni all'interno dei vincoli vengono valutate nei modi seguenti:

- ESPRESSIONI INSIEMISTICHE

  – insieme vuoto:

$$\frac{\varnothing}{\texttt{"LSet.empty()"}}$$

  – variabile in uno schema di struttura:

$$\frac{\text{VARIABILE}}{\texttt{"d." + VARIABILE.name)}}$$

  – variabile fuori da uno schema di struttura:

$$\frac{\text{VARIABILE)}}{\texttt{VARIABILE.name)}}$$

  – variabile decorata con "'":

$$\frac{\text{VARIABILE'}}{\texttt{VARIABILE'.name + "Prm"}}$$

  – variabile decorata con "?":

$$\frac{\text{VARIABILE?}}{\texttt{VARIABILE?.name + "Ask"}}$$

– variabile decorata con "!":

$$\frac{\text{VARIABILE!}}{\texttt{VARIABILE!.name + "Ans"}}$$

– unione:

$$\frac{\text{ESPRESSIONE1} \cup \text{ESPRESSIONE2}}{\texttt{"union(" + ESPRESSIONE1 + ", " + ESPRESSIONE2 + ")"}}$$

– intersezione:

$$\frac{\text{ESPRESSIONE1} \cap \text{ESPRESSIONE2}}{\texttt{"inters(" + ESPRESSIONE1 + ", " + ESPRESSIONE2 + ")"}}$$

– sottrazione:

$$\frac{\text{ESPRESSIONE1} \setminus \text{ESPRESSIONE2}}{\texttt{"diff(" + ESPRESSIONE1 + ", " + ESPRESSIONE2 + ")"}}$$

– dominio di una relazione binaria

$$\frac{dom\text{REL}}{\texttt{"lSet"}}$$

inserendo la seguente stringa prima dello statement domREL

$$\frac{}{\texttt{"LSet lSet = new LSet();" + REL + ".dom(lSet)"}}$$

- ESPRESSIONI ALGEBRICHE

– numero intero:

$$\frac{\text{NUMERO}}{\texttt{NUMERO}}$$

– somma:

$$\frac{\text{ESPRESSIONE1} + \text{ESPRESSIONE2}}{\texttt{ESPRESSIONE1 + ".sum(" + ESPRESSIONE2 + ")"}}$$

– sottrazione:

$$\frac{\text{ESPRESSIONE1} - \text{ESPRESSIONE2}}{\texttt{ESPRESSIONE1 + ".sub(" + ESPRESSIONE2 + ")"}}$$

– moltiplicazione:

$$\frac{\text{ESPRESSIONE1} * \text{ESPRESSIONE2}}{\texttt{ESPRESSIONE1 + ".mul(" + ESPRESSIONE2 + ")"}}$$

– divisione esatta:

$$\frac{\text{ESPRESSIONE1 / ESPRESSIONE2}}{\texttt{ESPRESSIONE1 + ".div(" + ESPRESSIONE2 + ")"}}$$

– cardinalità di un insieme:

$$\frac{\#\text{ESPRESSIONE}}{\texttt{"size(" + ESPRESSIONE + ")"}}$$

4.4 Algoritmi di traduzione da Z a Z

ZDOCUMENT viene "normalizzato" tramite le seguenti regole, prima della effettiva traduzione in codice java, affinché la traduzione diventi più semplice. Gli schemi, utilizzati all'interno di altri schemi, vengono espansi tramite le variabili e i vincoli contenuti nello schema stesso. Gli schemi, definiti tramite operazioni tra schemi, conterranno variabili e vincoli risultanti dall'operazione logica che li definisce.

4.4.1 Espansione operazione tra schemi

Input: ZDOCUMENT, insieme dei componenti della specifica Z
1 **foreach** *SchemaOperation* $s \in ZDOCUMENT$ **do**
2 **switch** $s.operator$ **do**
3 **case** "$\neg$" **do**
4 /* Nego i vincoli */
5 $s.schema.d \leftarrow s.schema1.d$ $s.schema.p \leftarrow \neg(s.schema1.p)$
6 **end**
7 **case** "$\vee$" **do**
8 /* Unione parte dichiarativa e disgiunzione dei vincoli */
9 $s.schema.d \leftarrow (s.schema1.d \cup s.schema2.d)$
10 $s.schema.p \leftarrow (s.schema1.p \vee s.schema2.p)$
11 **end**
12 **case** "$\wedge$" **do**
13 /* Unione parte dichiarativa e congiunzione dei vincoli */
14 $s.schema.d \leftarrow (s.schema1.d \cup s.schema2.d)$
15 $s.schema.p \leftarrow (s.schema1.p \wedge s.schema2.p)$
16 **end**
17 **end**
18 $ZDOCUMENT \leftarrow (ZDOCUMENT \setminus s)$
19 $ZDOCUMENT \leftarrow (ZDOCUMENT \cup s.schema)$
20 **end**

Algorithm 1: Espansione operazione tra schemi

Esempio Espansione dello schema 11 "$Prova == ProvaGiusto \vee ProvaSbagliato$":

Prova

$\Xi Partita$
$casuale? : \mathbb{Z}$
$messaggio! : MESSAGGIO$

$(casuale = casuale?$
$messaggio! = vittoria)$
$\vee$
$(casuale \neq casuale?$
$messaggio! = riprova)$

La parte dichiarativa è l'unione delle parti dichiarative degli operandi, mentre la parte predicativa ne è la disgiunzione logica.

4.4.2 Espansione degli schemi esterni

Schema

$SchemaDaEspandere$
$SchemaDaEspandere'$
$\Xi SchemaDaEspandere$
$\Delta SchemaDaEspandere$

In caso di decorazione "Ξ", lo schema viene sostituito dal medesimo, ma decorato con "Δ", e un vincolo di uguaglianza, per ogni variabile al suo interno, tra la variabile stessa e la sua versione decorata con "$'$", la quale rappresenta la variabile nello stato successivo della macchina.

Uno schema decorato con "Δ" viene sostituito da un suo riferimento decorato "$'$" e da un riferimento non decorato.

Infine, uno schema decorato con "$'$" viene sostituito dalle sue variabili e dai suoi vincoli, decorando tutte le variabili presenti con "$'$".

```
   Input: ZDOCUMENT, insieme dei componenti della specifica Z
 1 foreach Schema s ∈ ZDOCUMENT do
 2 |   foreach XiSchema xi ∈ s.d do
 3 |   |   /* Lo scema Xi diventa Delta                                  */
 4 |   |   s.d ← ((s.d \ xi) ∪ delta(xi))
 5 |   |   /* Aggiungo vincolo v = v' per ogni v                        */
 6 |   |   foreach Var v ∈ xi.d do
 7 |   |   |   s.p ← (s.p ∪ {v = v'})
 8 |   |   end
 9 |   end
10 |   foreach DeltaSchema delta ∈ s do
11 |   |   /* Lo schema Delta diventa uno schema non decorato e uno primed */
12 |   |   s.d ← ((s.d \ delta) ∪ schema(delta) ∪ primed(delta))
13 |   end
14 |   foreach PrimedSchema primed ∈ s do
15 |   |   s.d ← (s.d \ primed)
16 |   |   /* Aggiungo a s le variabili di primed e le decoro           */
17 |   |   foreach Var v ∈ primed.d do
18 |   |   |   s.d ← (s.d ∪ {v'})
19 |   |   end
20 |   |   /* Aggiungo a s i vincoli di primed e decoro le variabili al loro
   |   |      interno                                                   */
21 |   |   temp ← primed.p
22 |   |   foreach Var v ∈ temp do
23 |   |   |   v.name ← v'
24 |   |   end
25 |   |   s.p ← (s.p ∪ temp)
26 |   end
27 |   foreach UndecoratedSchema undecorated ∈ s do
28 |   |   s.d ← ((s.d \ undecorated) ∪ undecorated.d)
29 |   |   s.p ← (s.p ∪ undecorated.p)
30 |   end
31 end
```

Algorithm 2: Espansione schemi esterni

Esempio Espansione dello schema esterno ΞPartita all'interno dello schema *Prova*

Prova

$$\Xi Partita$$
$$casuale? : \mathbb{Z}$$
$$messaggio! : MESSAGGIO$$

$$(casuale = casuale?$$
$$messaggio! = vittoria)$$
$$\lor$$
$$(casuale \neq casuale?$$
$$messaggio! = riprova)$$

ricavato nell'esempio precedente:

Prova

$$difficolta : MODALITA$$
$$casuale : \mathbb{Z}$$
$$difficolta' : MODALITA$$
$$casuale' : \mathbb{Z}$$
$$casuale? : \mathbb{Z}$$
$$messaggio! : MESSAGGIO$$

$$difficolta' = difficolta$$
$$casuale' = casuale$$
$$casuale \geq 0$$
$$(difficolta = facile \land casuale < limitefacile) \lor$$
$$(difficolta = difficile \land casuale < limitedifficile)$$
$$casuale' \geq 0$$
$$(difficolta' = facile \land casuale' < limitefacile) \lor$$
$$(difficolta' = difficile \land casuale' < limitedifficile)$$
$$casuale? \geq 0$$
$$(casuale = casuale? \land messaggio! = vittoria) \lor$$
$$(casuale \neq casuale? \land messaggio! = riprova)$$

4.4.3 Specifica dopo le operazioni preliminari

La specifica normalizzata tramite le operazioni preliminari di traduzione da Z a Z:

1. Insiemi definiti per enumerazione

$$MESSAGGIO ::= vittoria \mid riprova$$

$$MODALITA ::= facile \mid difficile$$

2. Definizione assiomatica

$$\begin{array}{|l} limitefacile : \mathbb{Z} \\ limitedifficile : \mathbb{Z} \\ \hline limitefacile = 3 \\ limitedifficile = 5 \\ \\ \end{array}$$

3. Schema di stato

$$\begin{array}{|l} \textit{Partita} \\ \hline difficolta : MODALITA \\ casuale : \mathbb{Z} \\ \hline casuale \geq 0 \\ (difficolta = facile \land casuale < limitefacile) \lor \\ \quad (difficolta = difficile \land casuale < limitedifficile) \\ \hline \end{array}$$

4. Schema di inizializzazione

$$\begin{array}{|l} \textit{InitPartita} \\ \hline difficolta' : MODALITA \\ casuale' : \mathbb{Z} \\ \hline casuale' \geq 0 \\ difficolta' = facile \\ casuale' < limitefacile \\ (difficolta' = facile \land casuale' < limitefacile) \lor \\ \quad (difficolta' = difficile \land casuale' < limitedifficile) \\ \hline \end{array}$$

5. Schemi di azione

NuovaPartita

$$
\begin{array}{l}
difficolta : MODALITA \\
casuale : \mathbb{Z} \\
difficolta' : MODALITA \\
casuale' : \mathbb{Z} \\
difficolta? : MODALITA \\
\hline
casuale \geq 0 \\
(difficolta = facile \land casuale < limitefacile) \lor \\
\quad (difficolta = difficile \land casuale < limitedifficile) \\
casuale' \geq 0 \\
(difficolta' = facile \land casuale' < limitefacile) \lor \\
\quad (difficolta' = difficile \land casuale' < limitedifficile) \\
difficolta' = difficolta? \\
casuale' \neq casuale
\end{array}
$$

Prova

$$
\begin{array}{l}
difficolta : MODALITA \\
casuale : \mathbb{Z} \\
difficolta' : MODALITA \\
casuale' : \mathbb{Z} \\
casuale? : \mathbb{Z} \\
messaggio! : MESSAGGIO \\
\hline
difficolta' = difficolta \\
casuale' = casuale \\
casuale \geq 0 \\
(difficolta = facile \land casuale < limitefacile) \lor \\
\quad (difficolta = difficile \land casuale < limitedifficile) \\
casuale' \geq 0 \\
(difficolta' = facile \land casuale' < limitefacile) \lor \\
\quad (difficolta' = difficile \land casuale' < limitedifficile) \\
casuale? \geq 0 \\
(casuale = casuale? \land messaggio! = vittoria) \lor \\
\quad (casuale \neq casuale? \land messaggio! = riprova)
\end{array}
$$

4.5 Algoritmi di traduzione da Z a Java

Scheletro della classe principale

Il seguente codice java rappresenta la classe principale derivante dalla specifica Z, la quale verrà completata tramite l'applicazione dei successivi algoritmi. "ZDOCUMENT.NAME" verrà sostituito con il nome della specifica Z.

```
1  import JSetL.*;
2  import JSetL.z.*;
3  @SuppressWarnings("rawtypes")
4  // ZDOCUMENT.NAME = nome del documento z diventa il nome della classe
5  public class ZDOCUMENT.NAME extends ZDocument{
6      ZDOCUMENT.NAME(){
7          super();
8          execute();
9      }
10     public static void main(String[] args)
11     {
12     //crea il documento e lo apre tramite l' interfaccia
13     ZDOCUMENT.NAME doc = new ZDOCUMENT.NAME();
14     }
15 }
```

4.5.1 Individuazione dello stato iniziale

Lo schema iniziale rappresenta lo stato iniziale del programma. Grazie a questa informazione, il metodo risultante dalla traduzione dello schema di inizializzazione verrà eseguito all'avvio del programma e non sarà più proposto all'utente tra le operazioni eseguibili.

Input: ZDOCUMENT, insieme dei componenti della specifica Z

```
1 initName ← null
2 foreach Schema s ∈ ZDOCUMENT do
3 |   /* Schema iniziale il cui nome inizia con "Init"                    */
4 |   if s.name = ("Init" + anyString) then
5 |   |   initName ← s.name
6 |   |   break
7 |   end
8 end
9 statement("initz.open(doc, " + initName + ");", "MAIN")
```

Algorithm 3: Acquisizione nome dello schema di inizializzazione

4.5.2 Individuazione degli insiemi definiti per enumerazione

Questa procedura ha il compito di individuare, all'interno della specifica, gli insiemi definiti per enumerazione, supponendo di non avere un parser in grado di dedurlo da solo tramite la specifica scritta in formato latex.

Per ogni variabile della specifica, la procedura controlla se il suo tipo consiste in uno schema, in caso contrario si è in presenza di un insieme definito per enumerazione. Verrà istanziato un insieme *enumeratedSetSet* che sarà utilizzato dagli algoritmi successivi.

Input: ZDOCUMENT, insieme dei componenti della specifica Z

```
1  /* Rilevamento degli enumerated set presenti nella specifica          */
2  enumeratedSetSet ← ∅
3  foreach Schema s ∈ ZDOCUMENT do
4  |  foreach Var v ∈ s do
5  |  |  isASchema ← FALSE
6  |  |  type ← v.type
7  |  |  foreach Schema s1 ∈ ZDOCUMENT do
8  |  |  |  if type = s1.NAME then
9  |  |  |  |  isASchema ← TRUE
10 |  |  |  end
11 |  |  end
12 |  |  if isASchema = FALSE then
13 |  |  |  enumeratedSetSet ← (enumeratedSetSet ∪ {type})
14 |  |  end
15 |  end
16 end
```

Algorithm 4: Individuazione insiemi definiti per enumerazione

4.5.3 Individuazione degli schemi di stato e di azione

```
   Input: ZDOCUMENT, insieme dei componenti della specifica Z
 1 ActionSchemaSet ← ∅
 2 StateSchemaSet ← ∅
 3 foreach Schema s ∈ ZDOCUMENT do
 4 |   IsStateSchema ← TRUE
 5 |   foreach Var v ∈ s do
 6 |   |   if v.name = (anyString + "'") or
 7 |   |   v.name = (anyString + "?") or
 8 |   |   v.name = (anyString + "!") then
 9 |   |   |   IsStateSchema ← FALSE
10 |   |   end
11 |   end
12 |   if IsStateSchema = TRUE then
13 |   |   StateSchemaSet ← (StateSchemaSet ∪ s)
14 |   else
15 |   |   ActionSchemaSet ← (ActionSchemaSet ∪ s)
16 |   end
17 end
```

Algorithm 5: Traduzione schemi di stato

4.5.4 Individuazione degli schemi di struttura

Questa procedura ha il compito di individuare, all'interno della specifica, gli schemi di struttura. Verrà istanziato un insieme *SStruttura* che sarà utilizzato dagli algoritmi successivi.

Input: ZDOCUMENT, insieme dei componenti della specifica Z

```
1  /* Per ogni variabile di ogni schema, se il tipo della variabile è
      uguale al nome di uno schema, quello schema è uno schema di struttura
      */
2  SStruttura ← ∅
3  foreach Schema s ∈ ZDOCUMENT do
4  |  foreach Var v ∈ s do
5  |  |  foreach Schema s1 ∈ ZDOCUMENT do
6  |  |  |  if v.type = s1.name then
7  |  |  |  |  SStruttura ← (SStruttura ∪ {s1})
8  |  |  |  end
9  |  |  end
10 |  end
11 end
```

Algorithm 6: Individuazione schemi di struttura

4.5.5 Traduzione insiemi definiti per enumerazione

Input: enumeratedSetSet, insieme degli insiemi definiti per enumerazione

```
1  foreach EnumeratedSet es ∈ enumeratedSetSet do
2  |  statement("givenSet(" + es.name + ")", "CONSTRUCTOR")
3  |  statement("public LSet " + es.name + " = new LSet(" + es.NAME
   |    + ")", "ATTRIBUTE")
4  |  foreach Constant c ∈ es do
5  |  |  /* Vincolo che indica il dominio della variabile.  Viene aggiunto
   |  |     lo statement nel costruttore della classe principale        */
6  |  |  statement("statPost(" + c.name + ".in(" + es.name + "))",
   |  |    "CONSTRUCTOR")
7  |  |  /* Viene aggiunto l'attributo alla classe principale           */
8  |  |  statement("public LVar "+ c.name + " = new LVar" + "("+
   |  |    c.NAME + "," + c.NAME + ")", "ATTRIBUTE")
9  |  end
10 end
```

Algorithm 7: Traduzione insiemi definiti per enumerazione

Esempio: l'applicazione dell'algoritmo 5 all'insieme *MESSAGGIO* produce il seguente codice alle 3-5 e 10-12:

```
1 public class GIOCO extends ZDocument{
2     ...
```

```
3       public LSet messaggio = new LSet("MESSAGGIO");
4       public LVar vittoria = new LVar("VITTORIA", "VITTORIA");
5       public LVar riprova = new LVar("RIPROVA", "RIPROVA");
6       ...
7
8       GIOCO(){
9           ...
10          statPost(vittoria.in(messaggio));
11          statPost(riprova.in(messaggio));
12          givenSet(messaggio);
13          ...
14      }
15      ...
16  }
```

4.5.6 Traduzione schemi di struttura

Correlato all'algoritmo per la traduzione di uno schema di struttura viene generato inizialmente il seguente codice java rappresentante lo scheletro della classe derivata dalla traduzione stessa.

```
1   import JSetL.*;
2   import JSetL.z.*;
3   @SuppressWarnings("serial")
4   public class SCHEMA_DI_STRUTTURA.NAME<T extends P(DocumentoZ)>
5       extends SubLVar{
6       @SuppressWarnings("unchecked")
7       public SCHEMA_DI_STRUTTURA.NAME(ZDocument d, String name){
8           static public LSet type = new LSet("SCHEMA_DI_STRUTTURA.NAME"
        );
9           super(name,d);
10          init((T) d);
11      }
12      void init(T d){
13          d.statPost(this.in(type));
14      }
15  }
```

```
   Input: SStruttura, insieme contenente gli schemi di struttura della
          specifica
 1 foreach Schema s ∈ SStruttura do
 2 |   statement("givenSet(" + s.NAME +".type)", "CONSTRUCTOR")
 3 |   /* Nuova classe con lo stesso nome dello schema di struttura      */
 4 |   newClass(s.NAME)
 5 |   foreach var v ∈ s do
 6 |   |   /* Attributi pubblici di questa classe                        */
 7 |   |   if v.type ∈ SStruttura then
 8 |   |   |   statement( "public " + v.type + v.name + " = new " + v.type
   |   |   |    + "(this, " + v.NAME + ")", "STRUCTURE")
 9 |   |   else
10 |   |   |   statement( "public " + v.type + v.name + " = new " + v.type
   |   |   |    + "("+ v.NAME + ")", "STRUCTURE")
11 |   |   end
12 |   |   /* Vincolo sul tipo della variabile                          */
13 |   |   if v.type ≠ ℤ then
14 |   |   |   statement("d.statPost(" + domainConstraint(v) + ")", "INIT")
15 |   |   end
16 |   end
17 |   foreach Constraint c ∈ s do
18 |   |   /* Vincolo aggiunto allo store statico dei vincoli           */
19 |   |   statement("d.statPost(" + zToJavaConstraint(c) + ")", "INIT")
20 |   end
21 end
```

Algorithm 8: Espansione schemi di struttura

Esempio Nella specifica in esempio non sono presenti schemi di struttura, quindi si supponga una specifica contenente il seguente schema di struttura che vada a definire un nuovo tipo dato *Viaggio*:

Viaggio

$$Partenza : CONTINENTI$$
$$Arrivo : CONTINENTI$$

$$Partenza \neq Arrivo$$

Supponendo che "CONTINENTI" sia un insieme definito per enumerazione,la classe java risultante sarà:

```
1 import JSetL.*;
```

```
2  import JSetL.z.*;
3  @SuppressWarnings("serial")
4  public class Viaggio<T extends ZDOCUMENT>
5      extends SubLVar{
6          static public LSet type = new LSet("VIAGGIOTYPE");
7          public LVar partenza = new LVar("PARTENZA");
8          public LVar arrivo = new LVar("ARRIVO");
9
10     @SuppressWarnings("unchecked")
11     public Viaggio(ZDOCUMENT d, String name){
12         super(name,d);
13         init((T) d);
14     }
15     void init(T d){
16
17         d.statPost(this.in(type));
18
19         d.statPost(partenza.in(d.continenti));
20         d.statPost(arrivo.in(d.continenti));
21
22         d.statPost(partenza.neq(arrivo));
23
24     }
25 }
```

4.5.7 Traduzione definizioni assiomatiche

Input: ZDOCUMENT, insieme dei componenti della specifica Z

```
1  foreach AssDefinition a ∈ ZDOCUMENT do
2      foreach Var v ∈ a do
3          /* Viene aggiunto l'attributo alla classe principale          */
4          if v.type ∈ SStruttura then
5              statement("public " + v.type + v.name + " = new " + v.type
                + "(this, " + v.NAME + ")", "ATTRIBUTE")
6          else
7              statement("public " + v.type + v.name + " = new " + v.type
                + "("+ v.NAME + ")", "ATTRIBUTE")
8          end
9          if v.type ≠ ℤ then
10             /* Vincolo che indica il dominio della variabile          */
11             statement("statPost(" + domainConstraint(v) + ")",
                "CONSTRUCTOR")
12         end
13     end
14     foreach Constraint c ∈ a do
15         /* Vincolo aggiunto allo store statico.                       */
16         statement("statPost(" + zToJavaConstraint(c) + ")",
            "CONSTRUCTOR")
17     end
18 end
```

Algorithm 9: Traduzione definizioni assiomatiche

Esempio La definizione assiomatica

$$\begin{array}{|l} limitefacile : \mathbb{Z} \\ limitedifficile : \mathbb{Z} \\ \hline limitefacile = 3 \\ limitedifficile = 5 \\ \end{array}$$

genera le righe di codice 3-4 e 9-10 seguenti:

```
1 public class GIOCO extends ZDocument{
2   ...
3   public IntLVar limitefacile = new IntLVar("LIMITEFACILE");
4   public IntLVar limitedifficile = new IntLVar("LIMITEDIFFICILE");
5   ...
```

```
6
7   GIOCO(){
8    ...
9    statPost(limitefacile.eq(3));
10   statPost(limitedifficile.eq(5));
11   ...
12   }
13   ...
14 }
```

4.5.8 Traduzione schemi di stato

Input: StateSchemaSet, insieme degli schemi di stato
Input: SStruttura, insieme contenente gli schemi di struttura della specifica

```
1  foreach Schema s ∈ StateSchemaSet do
2      foreach var v ∈ s do
3          if v.type ∈ SStruttura then
4              statement( "public " + v.type + v.name + " = new " + v.type
                + "(this, " + v.NAME + ")", "ATTRIBUTE")
5          else
6              statement( "public " + v.type + v.name + " = new " + v.type
                + "(" + v.NAME + ")", "ATTRIBUTE")
7          end
8          if v.type ≠ ℤ then
9              /* Vincolo che indica il dominio della variabile          */
10             statement("statPost(" + domainConstraint(v) + ")",
                "CONSTRUCTOR")
11         end
12     end
13     foreach constraint c ∈ s do
14         /* Vincolo aggiunto allo store statico.                        */
15         statement("statPost(" + zToJavaConstraint(c) + ")",
            "CONSTRAINT")
16     end
17 end
```

Algorithm 10: Traduzione schemi di stato

Esempio Lo schema di stato *Partita*

Partita

$difficolta : MODALITA$
$casuale : \mathbb{Z}$

$casuale \geq 0$
$(difficolta = facile \land casuale < limitefacile) \lor$
$(difficolta = difficile \land casuale < limitedifficile)$

produce le seguenti righe di codice:

```
public class GIOCO extends ZDocument{
    ...
    public LVar difficolta = new LVar("DIFFICOLTA");
    public IntLVar casuale = new IntLVar("CASUALE");
    ...

    GIOCO(){
        ...
        statPost(difficolta.in(modalita));
        statPost(casuale.ge(0));
        statPost((difficolta.eq(facile).and
                (casuale.lt(limitefacile)))
            .or(difficolta.eq(difficile).and
                (casuale.lt(limitedifficile))));
        ...
    }
    ...
}
```

4.5.9 Traduzione schemi di azione

I casi primed, ask e ans del seguente algoritmo sono stati suddivisi in tre sezioni separate tra di loro per motivi di lunghezza del codice.

```
   Input: ActionSchemaSet, insieme degli schemi di azione
 1 foreach Schema s ∈ ActionSchemaSet do
 2 |   addMethod(s.name)
 3 |   foreach Var v ∈ s do
 4 |   |   switch v.decorazione do
 5 |   |   |   case nonDecorata do
 6 |   |   |   |   /* variabile già compare come attributo:  do nothing     */
 7 |   |   |   end
 8 |   |   |   case primed do
 9 |   |   |   |   /* Vedi algoritmo 12                                     */
10 |   |   |   end
11 |   |   |   case ask do
12 |   |   |   |   /* Vedi algoritmo 13                                     */
13 |   |   |   end
14 |   |   |   case ans do
15 |   |   |   |   /* Vedi algoritmo 14                                     */
16 |   |   |   end
17 |   |   end
18 |   end
19 |   foreach Constraintc ∈ s do
20 |   |   /* tempPost() aggiunge il vincolo allo store temporaneo dei
   |   |      vincoli                                                       */
21 |   |   statement( "tempPost(" + zToJavaConstraint(c) + ")",
   |   |    "ACTION")
22 |   end
23 |   statement("execute()", "ACTION")
24 end
```

Algorithm 11: Traduzione schemi di azione

```
1  oldName ← v.name
2  v.name ← (v.name + "Prm")
3  if v.type ∈ SStruttura then
4  |  statement( v.type + v.name + " = new " + v.type + "(this, " +
   |    v.name + ")", "ACTION")
5  else
6  |  statement( v.type + v.name + " = new " + v.type + "("+ v.name +
   |    ")", "ACTION")
7  end
8  if v.type ≠ ℤ then
9  |  statement("tempPost(" + domainConstraint(v) + ")", "ACTION")
10 end
11 statement("prmVar(" + oldName + ", " + v.name + ")", "ACTION")
```

Algorithm 12: Variabile primed

```
1  v.name ← (v.name + "Ask")
2  if v.type ∈ SStruttura then
3  |  statement( v.type + v.name + " = new " + v.type + "(this, " +
   |    v.name + ")", "ACTION")
4  else
5  |  statement( v.type + v.name + " = new " + v.type + "("+ v.name +
   |    ")", "ACTION")
6  end
7  if v.type ≠ ℤ then
8  |  statement("tempPost(" + domainConstraint(v) + ")", "ACTION")
9  end
10 statement("askVar(" + v.name + ")", "ACTION")
```

Algorithm 13: Variabile di input

```
1  v.name ← (v.name + "Ans")
2  if v.type ∈ SStruttura then
3  |  statement( v.type + " " + v.name + " = new " + v.type + "(this, " +
   |    v.name + ")", "ACTION")
4  else
5  |  statement( v.type + " " + v.name + " = new " + v.type + "("+
   |    v.name + ")", "ACTION")
6  end
7  if v.type ≠ ℤ then
8  |  statement("tempPost(" + domainConstraint(v) + ")", "ACTION")
9  end
10 statement("ansVar(" + v.name + ")", "ACTION")
```

Algorithm 14: Variabile di output

L'applicazione dell'algoritmo 12 allo schema d'azione *Prova* produce le seguenti righe di codice:

```
public class GIOCO extends ZDocument{
    ...
    public void prova(){
        // variabili di input, valore chiesto all'utente
        IntLVar casualeAsk = new IntLVar("CASUALEASK");
        askVar(casualeAsk);

        // variabili primed
        LVar difficoltaPrm = new LVar("DIFFICOLTAPRM");
        IntLVar casualePrm = new IntLVar("CASUALEPRM");
        // "difficolta" verra' uguagliato a difficoltaPrm
        prmVar(difficolta, difficoltaPrm);
        tempPost(difficoltaPrm.in(modalita));
        prmVar(casuale, casualePrm);

        // variabili di output
        LVar messaggioAns = new LVar("MESSAGGIOANS");
        ansVar(messaggioAns);
        tempPost(messaggioAns.in(messaggio));

        // accumulo vincoli temporanei per questa procedura
        tempPost(difficoltaPrm.eq(difficolta));
        tempPost(casualePrm.eq(casuale));
        tempPost(casuale.ge(0));
        tempPost((difficolta.eq(facile).and
                (casuale.lt(limitefacile)))
            .or(difficolta.eq(difficile).and
                (casuale.lt(limitedifficile))));
        tempPost(casualePrm.ge(0));
        tempPost((difficoltaPrm.eq(facile).and
                (casualePrm.lt(limitefacile)))
            .or(difficoltaPrm.eq(difficile).and
                (casualePrm.lt(limitedifficile))));
        tempPost(casualeAsk.ge(0));
        tempPost((casuale.eq(casualeAsk).and
                (messaggioAns.eq(vittoria)))
            .or(casuale.neq(casualeAsk).and
                (messaggioAns.eq(riprova))));
        //esecuzione operazione
        execute();
    }
    ...
}
```

4.6 Programma Java risultante

```
1  package jsetl.z_esempi;
2
3  import jsetl.*;
4  import jsetl.z.*;
5  @SuppressWarnings("rawtypes") // per lt sottoclassi (parametriche) di
       SubLVar
6  public class GIOCO extends ZDocument {
7      // definizioni di insiemi dati(strutture jsetl pubbliche)
8      public LSet messaggio = new LSet("MESSAGGIO");
9      public LSet modalita = new LSet("MODALITA");
10
11     // definizioni di variabili globali e di stato(strutture jsetl
       pubbliche)
12     public IntLVar limitefacile = new IntLVar("LIMITEFACILE");
13     public IntLVar limitedifficile = new IntLVar("LIMITEDIFFICILE");
14     public LVar vittoria = new LVar("VITTORIA", "VITTORIA");
15     public LVar riprova = new LVar("RIPROVA", "RIPROVA");
16     public LVar facile = new LVar("FACILE", "FACILE");
17     public LVar difficile = new LVar("DIFFICILE", "DIFFICILE");
18     public LVar difficolta = new LVar("DIFFICOLTA");
19     public IntLVar casuale = new IntLVar("CASUALE");
20
21     GIOCO() {
22         super();
23         // vincoli di variabili globali e di stato, vincoli globali
       di stato
24         // vincoli statici, aspetti dello stato del sistema che non
       possono
25         // essere alterati.
26         statPost(limitefacile.eq(3));
27         statPost(limitedifficile.eq(5));
28
29         statPost(vittoria.in(messaggio));
30         statPost(riprova.in(messaggio));
31         statPost(facile.in(modalita));
32         statPost(difficile.in(modalita));
33
34         statPost(difficolta.in(modalita));
35         statPost(difficolta.in(modalita));
36
37         statPost(casuale.ge(0));
38         statPost((difficolta.eq(facile)
39                 .and(casuale.lt(limitefacile)))
40                         .or(difficolta.eq(difficile).and(
41                                 casuale.lt(limitedifficile))));
42
```

```
43          // vincoli di insiemi definiti per enumerazione
44          givenSet(messaggio);
45          givenSet(modalita);
46          execute();
47      }
48
49      public static void main(String[] args) {
50          // crea il documento e lo apre tramite l'interfaccia
51          GIOCONO doc = new GIOCO();
52          doc.partitainiziale();
53          intz.open(doc, "InitPartita");
54      }
55
56      // traduzione schema iniziale "InitPartita"
57      public void InitPartita() {
58          // variabili primed
59          LVar difficoltaPrm = new LVar("DIFFICOLTAPRM");
60          IntLVar casualePrm = new IntLVar("CASUALEPRM");
61          // "difficolta" verra' uguagliato a difficoltaPrm
62          prmVar(difficolta, difficoltaPrm);
63
64          tempPost(difficoltaPrm.in(modalita));
65          prmVar(casuale, casualePrm);
66
67          // accumulo vincoli temporanei per questa procedura
68
69          tempPost(casualePrm.ge(0));
70          tempPost(difficoltaPrm.eq(facile));
71          tempPost(casualePrm.lt(limitefacile));
72          tempPost((difficoltaPrm.eq(facile)
73          .and(casualePrm.lt(limitefacile)))
74              .or(difficoltaPrm.eq(difficile)
75                  .and(casualePrm.lt(limitedifficile))));
76          tempPost(casualePrm.label());
77
78          // esecuzione operazione
79          execute();
80      }
81
82      // operazione per lo schema d'azione "nuovapartita"
83      public void nuovapartita() {
84          // variabili di input, valore chiesto all'utente
85          LVar difficoltaAsk = new LVar("DIFFICOLTAASK");
86          askVar(difficoltaAsk);
87          tempPost(difficoltaAsk.in(modalita));
88
89          // variabili primed
90          LVar difficoltaPrm = new LVar("DIFFICOLTAPRM");
91          IntLVar casualePrm = new IntLVar("CASUALEPRM");
```

```
92          // "difficolta" verra' uguagliato a difficoltaPrm
93          prmVar(difficolta, difficoltaPrm);
94          tempPost(difficoltaPrm.in(modalita));
95          prmVar(casuale, casualePrm);
96
97          // accumulo vincoli (usare tempPost)
98          tempPost(casuale.ge(0));
99          tempPost((difficolta.eq(facile)
100         .and(casuale.lt(limitefacile)))
101             .or(difficolta.eq(difficile).and(
102                 casuale.lt(limitedifficile))));
103         tempPost(casualePrm.ge(0));
104         tempPost((difficoltaPrm.eq(facile)
105         .and(casualePrm.lt(limitefacile)))
106             .or(difficoltaPrm.eq(difficile).and(
107                 casualePrm.lt(limitedifficile))));
108         tempPost(difficoltaPrm.eq(difficoltaAsk));
109         tempPost(casualePrm.neq(casuale));
110         tempPost(casualePrm.label());
111
112
113
114         // esecuzione operazione
115         execute();
116
117     }
118
119     // operazione per lo schema d'azione "prova"
120     public void prova() {
121         // variabili di input, valore chiesto all'utente
122         IntLVar casualeAsk = new IntLVar("CASUALEASK");
123         askVar(casualeAsk);
124
125         // variabili primed
126         LVar difficoltaPrm = new LVar("DIFFICOLTAPRM");
127         IntLVar casualePrm = new IntLVar("CASUALEPRM");
128
129         // "difficolta" verra' uguagliato a difficoltaPrm
130         prmVar(difficolta, difficoltaPrm);
131         tempPost(difficoltaPrm.in(modalita));
132         prmVar(casuale, casualePrm);
133
134         // variabili di output
135         LVar messaggioAns = new LVar("MESSAGGIOANS");
136         ansVar(messaggioAns);
137         tempPost(messaggioAns.in(messaggio));
138
139         // accumulo vincoli temporanei per questa procedura
140         tempPost(difficoltaPrm.eq(difficolta));
```

```
141         tempPost(casualePrm.eq(casuale));
142         tempPost(casuale.ge(0));
143         tempPost((difficolta.eq(facile)
144         .and(casuale.lt(limitefacile)))
145             .or(difficolta.eq(difficile).and(
146                 casuale.lt(limitedifficile))));
147         tempPost(casualePrm.ge(0));
148         tempPost((difficoltaPrm.eq(facile)
149         .and(casualePrm.lt(limitefacile)))
150             .or(difficoltaPrm.eq(difficile).and(
151                 casualePrm.lt(limitedifficile))));
152         tempPost(casualeAsk.ge(0));
153         tempPost((casuale.eq(casualeAsk)
154         .and(messaggioAns.eq(vittoria)))
155             .or(casuale.neq(casualeAsk)
156                 .and(messaggioAns.eq(riprova))));
157 
158         // esecuzione operazione
159         execute();
160     }
161 }
```

4.7 Modifiche sul codice Java risultante

4.7.1 Modifiche automatiche

Nel codice Java ottenuto dalla traduzione dalla specifica Z, è possibile che siano presenti vincoli ridondanti, la cui rimozione non modificherebbe la semantica del programma. Inoltre si otterrebbe maggiore leggibilità del codice e un miglioramento, anche se molto ridotto, della velocità del programma.

Vincoli di uguaglianza Nel costruttore della classe principale, vengono definiti i vincoli riguardanti lo schema di stato, le definizioni assiomatiche e gli insiemi definiti per enumerazione. Può capitare spesso che nelle definizioni assiomatiche ci siano vincoli di uguaglianza, in cui si attribuisce immediatamente un valore noto ad una variabile, perché il loro scopo è proprio quello di sottospecificare un aspetto del sistema che non è fondamentale per il suo funzionamento.

Nell'esempio visto in precedenza, sono presenti due vincoli di uguaglianza che impostano il valore massimo del numero casuale da indovinare; sarebbe stato verosimile aggiungere anche un limite massimo ai tentativi a disposizione dell'utente per indovinare il numero corretto.

Quindi i vincoli seguenti:

```
1 statPost(limitefacile.eq(3));
2 statPost(limitedifficile.eq(5));
```

vengono sostituiti con l'assegnazione del valore direttamente nel costruttore delle variabili in questione:

```
public IntLVar limitefacile = new IntLVar("LIMITEFACILE", 3);
public IntLVar limitedifficile = new IntLvar("LIMITEDIFFICILE", 5);
```

Questa modifica può essere formalizzata come segue:

Input: Java program
foreach *Vincolo nella forma v.eq() nel costruttore* **do**
  if *v è uguagliata ad un intero i* **then**
    elimina il vincolo
    aggiungi *i* come secondo parametro del costruttore di *v*
  end
end

Algorithm 15: Rimozione vincoli di uguaglianza

4.7.2 Modifiche manuali

Alcuni aspetti della programmazione sono troppo complicati da esprimere con la specifica Z, quindi non è possibile ritrovarli nel codice java da essa derivante.

Nell'esempio fornito in precedenza, il concetto di numero casuale non è stato realmente espresso nella specifica. La variabile *casuale* è stata definita come un intero che può assumere un valore all'interno di un sottoinsieme di $\mathbb{Z}$ e che, quando si effettua una nuova partita, quel valore dovrà cambiare.

L'alternativa sarebbe stata quella di definire uno schema, da usare come tipo della variabile *casuale* il quale implementasse un algoritmo di generazione casuale di un numero all'interno di un certo intervallo. Si tratta di una complicazione troppo eccessiva della specifica, per questo motivo può essere utile fare una modifica, non automatizzabile, sul codice java, per avvicinarsi maggiormente ai requisiti su cui si basa la specifica.

Eseguendo il programma ottenuto dall'esempio preso in esame, può apparire che *casuale* sia generato casualmente, in realtà gli viene assegnato il primo valore ritenuto valido da JSetL. Per attenersi ai requisiti, si può modificare alcuni vincoli inserendo l'uguaglianza tra *casuale* e un intero generato da Math.random(), in accordo con gli altri vincoli che ne determinano il dominio, per esempio:

```
//Metodo: nuovapartita
(difficoltaPrm.eq(facile).and
(casualePrm.lt(limitefacile)))
.or
(difficoltaPrm.eq(difficile).and
(casualePrm.lt(limitedifficile)))
```

diventa

```
//Metodo: nuovapartita
int randomFacile = (int)(Math.random() * limitefacile.getValue());
int randomDifficile = (int)(Math.random() * limitedifficile.getValue
    ());
while(randomFacile == casuale.getValue()) {
    randomFacile = (int)(Math.random() * limitefacile.getValue());
}
while(randomDifficile == casuale.getValue()) {
    randomDifficile = (int)(Math.random() * limitedifficile.getValue
    ());
}
(difficoltaprm.eq(facile)
    .and(casualeprm.lt(limitefacile)).and(casualeprm.eq(randomFacile)
    ))
        .or(difficoltaprm.eq(difficile)
    .and(casualeprm.lt(limitedifficile))
    .and(casualeprm.eq(randomDifficile)))
```

Tale modifica permette di generare un valore, da attribuire alla variabile *casualeprm*, realmente casuale e diverso rispetto al valore attuale di tale variabile.

4.8 Secondo esempio di specifica Z

L'esempio seguente [2] è una specifica Z che rappresenta un sistema bancario in cui si possono effettuare alcune operazioni di base. I requisiti sono i seguenti:

1. Ogni cliente deve essere identificato dal codice della propria carta di identità (NIC).
2. Ogni cliente può avere al massimo un solo conto corrente.
3. Ogni conto corrente deve essere identificato dal NIC del suo proprietario.
4. In ogni conto corrente viene registrato il saldo corrente.
5. Quando un conto corrente viene aperto, il suo saldo è zero.
6. Su un conto corrente esistente è possibile depositare una quantità positiva di denaro, ritirarne una quantità positiva e non superiore a quella disponibile e controllare il saldo corrente.
7. Un conto corrente può essere chiuso solamente se il suo saldo è zero.

4.8.1 Specifica iniziale

1. Identificativo clienti

   $$[NIC]$$

2. Messaggi di output

   $$MSG ::= ok \mid nicExists \mid nicNotExists \mid amountError \mid insufficientFunds \mid balanceNotZero$$

3. Stato del sistema

   $$Bank$$
   $$sa : NIC \pfun \mathbb{Z}$$

4. Stato iniziale del sistema

   $$InitBank$$
   $$Bank'$$

   $$sa' = \varnothing$$

5. Apertura di un conto corrente senza errori

   $$OpenAccountOK$$
   $$\Delta Bank$$
   $$n? : NIC$$
   $$msg! : MSG$$

   $$n? \notin \operatorname{dom} sa$$
   $$sa' = sa \cup \{n? \mapsto 0\}$$
   $$msg! = ok$$

6. Errore di utente già esistente nel sistema

   $$AccountAlreadyExists$$
   $$\Xi Bank$$
   $$n? : NIC$$
   $$msg! : MSG$$

   $$n? \in \operatorname{dom} sa$$
   $$msg! = nicExists$$

7. Apertura di un conto corrente

$$OpenAccount == OpenAccountOK \lor AccountAlreadyExists$$

8. Operazione di deposito senza errori

DepositOK

$\Delta Bank$
$n? : NIC$
$a? : \mathbb{Z}$
$msg! : MSG$

$n? \in \operatorname{dom} sa$
$0 < a?$
$sa' = sa \oplus \{n? \mapsto sa(n?) + a?\}$
$msg! = ok$

9. Errore di utente non esistente

AccountNotExists

$\Xi Bank$
$n? : NIC$
$msg! : MSG$

$n? \notin \operatorname{dom} sa$
$msg! = nicNotExists$

10. Errore di importo non valido

IncorrectAmount

$\Xi Bank$
$a? : \mathbb{Z}$
$msg! : MSG$

$a? \leq 0$
$msg! = amountError$

11. Operazione di deposito

$$Deposit == DepositOK \lor AccountNotExists \lor IncorrectAmount$$

12. Operazione di prelievo senza errori

$$
\begin{array}{|l}
\hline
WithdrawOK \\
\Delta Bank \\
n? : NIC \\
a? : \mathbb{Z} \\
msg! : MSG \\
\hline
n? \in \operatorname{dom} sa \\
0 < a? \\
a? \leq sa(n?) \\
sa' = sa \oplus \{n? \mapsto sa(n?) - a?\} \\
msg! = ok \\
\hline
\end{array}
$$

13. Errore di saldo insufficiente

$$
\begin{array}{|l}
\hline
InsufficientFunds \\
\Xi Bank \\
n? : NIC \\
a? : \mathbb{Z} \\
msg! : MSG \\
\hline
a? > sa(n?) \\
n? \in \operatorname{dom} sa \\
msg! = insufficientFunds \\
\hline
\end{array}
$$

14. Operazione di prelievo

$$Withdraw == WithdrawOK \lor AccountNotExists \lor IncorrectAmount \lor InsufficientFunds$$

15. Operazione di chiusura di un conto senza errori

$$
\begin{array}{|l}
\hline
CloseAccountOK \\
\Delta Bank \\
n? : NIC \\
msg! : MSG \\
\\
\hline
n? \in \operatorname{dom} sa \\
sa(n?) = 0 \\
sa' = \{n?\} \lhd sa \\
msg! = ok \\
\hline
\end{array}
$$

16. Errore di saldo diverso da 0

$$
\begin{array}{|l}
\hline
BalanceNotZero \\
\Xi Bank \\
n? : NIC \\
msg! : MSG \\
\hline
n? \in \operatorname{dom} sa \\
sa(n?) \neq 0 \\
msg! = balanceNotZero \\
\hline
\end{array}
$$

17. Operazione di chiusura di un conto

$$CloseAccount == CloseAccountOK \lor AccountNotExists \lor BalanceNotZero$$

18. Operazione di controllo del saldo senza errori

$$
\begin{array}{|l}
\hline
CheckBalanceOk \\
\Xi Bank \\
n? : NIC \\
bal! : \mathbb{Z} \\
msg! : msg \\
\hline
n? \in \operatorname{dom} sa \\
bal! = sa(n?) \\
msg! = ok \\
\hline
\end{array}
$$

19. Operazione di controllo del saldo

$$CheckBalance == CheckBalanceOk \lor AccountNotExists$$

4.8.2 Specifica dopo le operazioni preliminari

1. Identificativi dei clienti della banca

$$[NIC]$$

2. Messaggi di output

$$
\begin{aligned}
&MSG ::= \\
&\quad ok \mid nicExists \mid nicNotExists \mid amountError \mid \\
&insufficientFunds \mid balanceNotZero
\end{aligned}
$$

3. Stato del sistema (Conti correnti nella banca)

$$
\begin{array}{|l}
\hline
Bank \\
sa : NIC \pfun \mathbb{Z} \\
\hline
\end{array}
$$

4. Stato iniziale del sistema

$$
\begin{array}{|l}
\hline
InitBank \\
sa' : NIC \pfun \mathbb{Z} \\
\hline
sa' = \varnothing \\
\hline
\end{array}
$$

5. Operazione di apertura di un nuovo conto corrente

$$
\begin{array}{|l}
\hline
OpenAccount \\
sa : NIC \pfun \mathbb{Z} \\
sa' : NIC \pfun \mathbb{Z} \\
n? : NIC \\
msg! : MSG \\
\hline
(n? \notin \dom sa \\
sa' = sa \cup \{n? \mapsto 0\} \\
msg! = ok) \\
\lor \\
(n? \in \dom sa \\
sa' = sa \\
msg! = nicExists) \\
\hline
\end{array}
$$

6. Operazione di deposito

Deposit

$$
\begin{array}{l}
sa : NIC \pfun \mathbb{Z} \\
sa' : NIC \pfun \mathbb{Z} \\
n? : NIC \\
a? : \mathbb{Z} \\
msg! : MSG \\
\hline
(n? \in \dom sa \\
0 < a? \\
sa' = sa \oplus \{n? \mapsto sa(n?) + a?\} \\
msg! = ok) \\
\lor \\
(sa = sa' \\
n? \notin \dom sa \\
msg! = nicNotExists) \\
\lor \\
(sa = sa' \\
a? \leq 0 \\
msg! = amountError)
\end{array}
$$

7. Operazione di prelievo

$$
\begin{array}{|l}
\hline
WithDraw \\
sa : NIC \pfun \mathbb{Z} \\
sa' : NIC \pfun \mathbb{Z} \\
n? : NIC \\
a? : \mathbb{Z} \\
msg! : MSG \\
\hline
(n? \in \operatorname{dom} sa \\
0 < a? \\
a? \leq sa(n?) \\
sa' = sa \oplus \{n? \mapsto sa(n?) - a?\} \\
msg! = ok) \\
\lor \\
(sa = sa' \\
n? \notin \operatorname{dom} sa \\
msg! = nicNotExists) \\
\lor \\
(sa = sa' \\
a? \leq 0 \\
msg! = amountError) \\
\lor \\
(sa = sa' \\
a? > sa(n?) \\
n? \in \operatorname{dom} sa \\
msg! = insufficientFunds) \\
\hline
\end{array}
$$

8. Operazione di chiusura di un conto

CloseAccount

$$
\begin{array}{l}
sa : NIC \pfun \mathbb{Z} \\
sa' : NIC \pfun \mathbb{Z} \\
n? : NIC \\
msg! : MSG \\
\\
\hline
(n? \in \operatorname{dom} sa \\
sa(n?) = 0 \\
sa' = \{n?\} \ndres sa \\
msg! = ok) \\
\lor \\
(sa = sa' \\
n? \notin \operatorname{dom} sa \\
msg! = nicNotExists) \\
\lor \\
(sa = sa' \\
n? \in \operatorname{dom} sa \\
sa(n?) \neq 0 \\
msg! = balanceNotZero)
\end{array}
$$

9. Operazione di controllo del saldo corrente

CheckBalance

$$
\begin{array}{l}
sa : NIC \pfun \mathbb{Z} \\
sa' : NIC \pfun \mathbb{Z} \\
n? : NIC \\
bal! : \mathbb{Z} \\
msg! : MSG \\
\hline
sa = sa' \\
(n? \in \operatorname{dom} sa \\
bal! = sa(n?) \\
msg! = ok) \\
\lor \\
(n? \notin \operatorname{dom} sa \\
msg! = nicNotExists)
\end{array}
$$

4.8.3 Codice Java risultante

```
1  package banca;
2
3  import jsetl.*;
4  import jsetl.z.*;
5
6  @SuppressWarnings("rawtypes")
7  public class BANCA extends ZDocument{
8      //definizioni di insiemi dati(strutture JSetL pubbliche)
9      public LSet nic = new LSet("NIC");
10     public LSet msg = new LSet("MSG");
11
12     //definizioni di variabili globali e di stato(strutture JSetL
       pubbliche)
13     public LVar ok = new LVar("OK", "OK");
14     public LVar nicExists = new LVar("NICEXISTS", "NICEXISTS");
15     public LVar amountError = new LVar("AMOUNTERROR", "AMOUNTERROR");
16     public LVar insufficientFunds = new LVar("INSUFFICIENTFUNDS", "
       INSUFFICIENTFUNDS");
17     public LVar balanceNotZero = new LVar("BALANCENOTZERO", "
       BALANCENOTZERO");
18     public LVar nicNotExists = new LVar("NICNOTEXISTS", "NICNOTEXISTS
       ");
19     public LMap sa = new LMap("SA");
20
21
22     BANCA(){
23         super();
24         //vincoli di variabili globali e di stato, vincoli globali di
        stato
25         statPost(ok.in(msg));
26         statPost(nicExists.in(msg));
27         statPost(amountError.in(msg));
28         statPost(insufficientFunds.in(msg));
29         statPost(balanceNotZero.in(msg));
30         statPost(nicNotExists.in(msg));
31         statPost(sa.dom(nic).and(sa.ran(z)));
32         statPost(sa.pfun());
33
34         //vincoli di insiemi dati
35         givenSet(nic);
36         givenSet(msg);
37
38     }
39     public static void main(String[] args){
40         //crea il documento e lo apre tramite l interfaccia
41         BANCA doc = new BANCA();
42         intz.open(doc, "initbank");
43     }
44
```

```
45      public void Initbank(){
46          tempPost(sa.eq(LSet.empty()));
47          execute();
48      }
49
50      public void openaccount(){
51          // sa' : NIC -> Z
52          LMap saPrm = new LMap("SAPRM");
53          tempPost(saPrm.dom(nic).and(saPrm.ran(z).and(saPrm.pfun())));
54          prmVar(sa, saPrm);
55
56          // n? : NIC
57          LVar nAsk = new LVar("NASK");
58          tempPost(nAsk.in(nic));
59          askVar(nAsk);
60
61          // msg! : MSG
62          LVar msgAns = new LVar("MSGANS");
63          tempPost(msgAns.in(msg));
64          ansVar(msgAns);
65
66          // Vincoli
67          tempPost(
68              sa.dom(new LSet().ins(nAsk)).notTest(). // n? nin dom(sa)
69              and(sa.union(LMap.empty().ins(new LPair(nAsk, 0)), saPrm)
    .  // sa' = sa U {n? -> 0}
70              and(msgAns.eq(ok))).     // msg! = ok
71              or(sa.dom(new LSet().ins(nAsk)).     // n? in dom(sa)

72              and(saPrm.eq(sa)).  // sa' = sa
73              and(msgAns.eq(nicExists))));     // msg! = nicExists
74          execute();
75      }
76
77      public void deposit(){
78          // sa' : NIC -> Z
79          LMap saPrm = new LMap("SAPRM");
80          tempPost(saPrm.dom(nic).and(saPrm.ran(z).and(saPrm.pfun())));
81          prmVar(sa, saPrm);
82
83          // n? : NIC
84          LVar nAsk = new LVar("NASK");
85          tempPost(nAsk.in(nic));
86          askVar(nAsk);
87
88          // a? : Z
89          IntLVar aAsk = new IntLVar("A");
90          tempPost(aAsk.in(z));
91          askVar(aAsk);
```

```
92
93          // msg! : MSG
94          LVar msgAns = new LVar("MSGANS");
95          tempPost(msgAns.in(msg));
96          ansVar(msgAns);
97
98          // Vincoli
99          /*
100          * A = {nAsk -> sa(nAsk) + aAsk}
101          * B = dom A = {nAsk}
102          * C = B ndres sa
103          * D = sa(nASk) + aAsk = E + aAsk
104          * E = sa(nAsk)
105          *
106          * sa' = sa oplus {nAsk -> sa(nAsk) + aAsk}
107          * sa' = (dom({nAsk -> sa(nAsk) + aAsk}) ndres sa) U {nAsk ->
     sa(nAsk) + aAsk}
108          * sa' = (dom(A) ndres sa) U A
109          * sa' = (B ndres sa) U A
110          * sa' = C U A
111
112          */
113          LVar D = new LVar();
114          IntLVar E = new IntLVar();
115          LRel A = LRel.empty().ins(new LPair(nAsk, D)); // A = {n? ->
     D}
116          LSet B = new LSet();
117          LMap C = new LMap();
118
119          tempPost(
120                  sa.dom(new LSet().ins(nAsk)).
121
122                  and(aAsk.gt(0)).
123
124                  and(new LPair(nAsk, E).in(sa)). // E = sa(n?)
125                  and(D.eq(E.sum(aAsk))). // D = E + aAsk = sa(n?) + a?
126                  and(A.dom(B)).   // B = dom(A)
127                  and(sa.ndres(B, C)).     // C = B ndres sa
128                  and(C.union(A, saPrm)). // sa' = C \cup A
129
130                  and(msgAns.eq(ok)).
131
132                      or(sa.eq(saPrm).
133                  and(sa.dom(new LSet().ins(nAsk)).notTest()).
134                  and(msgAns.eq(nicNotExists))).
135                      or(sa.eq(saPrm).
136                  and(aAsk.le(0)).
137                  and(msgAns.eq(amountError))));
138          execute();
```

```
139     }
140
141     public void withdraw(){
142         // sa' : NIC -> Z
143         LMap saPrm = new LMap("SAPRM");
144         tempPost(saPrm.dom(nic).and(saPrm.ran(z).and(saPrm.pfun())));
145         prmVar(sa, saPrm);
146
147         // n? : NIC
148         LVar nAsk = new LVar("NASK");
149         tempPost(nAsk.in(nic));
150         askVar(nAsk);
151
152         // a? : Z
153         IntLVar aAsk = new IntLVar("A");
154         tempPost(aAsk.in(z));
155         askVar(aAsk);
156
157         // msg! : MSG
158         LVar msgAns = new LVar("MSGANS");
159         tempPost(msgAns.in(msg));
160         ansVar(msgAns);
161
162         // Vincoli
163
164         LVar D = new LVar();
165         IntLVar E = new IntLVar();
166         LRel A = LRel.empty().ins(new LPair(nAsk, D)); // A = {n? ->
    D}
167         LSet B = new LSet();
168         LMap C = new LMap();
169
170         tempPost(
171             sa.dom(new LSet().ins(nAsk)).
172
173             and(aAsk.gt(0)).
174
175             and(new LPair(nAsk, E).in(sa)).
176             and(aAsk.le(E)).
177
178             and(new LPair(nAsk, E).in(sa)). // E = sa(n?)
179             and(D.eq(E.sub(aAsk))). // D = E - aAsk = sa(n?) - a?
180             and(A.dom(B)).   // B = dom(A)
181             and(sa.ndres(B, C)).     // C = B ndres sa
182             and(C.union(A, saPrm)). // sa' = C U A
183
184             and(msgAns.eq(ok)).
185             or(sa.eq(saPrm).
186
```

```
187             and(sa.dom(new LSet().ins(nAsk)).notTest()).
188
189             and(msgAns.eq(nicNotExists))).
190             or(sa.eq(saPrm).
191             and(aAsk.le(0)).
192             and(msgAns.eq(amountError))).
193             or(sa.eq(saPrm).
194             and(new LPair(nAsk, E).in(sa)).
195             and(aAsk.gt(E)).
196
197             and(sa.dom(new LSet().ins(nAsk))).
198
199             and(msgAns.eq(insufficientFunds))));
200         execute();
201     }
202
203     public void closeaccount(){
204         // sa' : NIC -> Z
205         LMap saPrm = new LMap("SAPRM");
206         tempPost(saPrm.dom(nic).and(saPrm.ran(z).and(saPrm.pfun())));
207         prmVar(sa, saPrm);
208
209         // n? : NIC
210         LVar nAsk = new LVar("NASK");
211         tempPost(nAsk.in(nic));
212         askVar(nAsk);
213
214         // msg! : MSG
215         LVar msgAns = new LVar("MSGANS");
216         tempPost(msgAns.in(msg));
217         ansVar(msgAns);
218
219         // Vincoli
220
221         IntLVar E = new IntLVar();
222
223         tempPost(
224             sa.dom(new LSet().ins(nAsk)).
225
226             and(new LPair(nAsk, E).in(sa)).
227             and(E.eq(0)).
228
229             and(sa.ndres(LSet.empty().ins(nAsk), saPrm)).
230
231             and(msgAns.eq(ok)).
232             or(sa.eq(saPrm).
233             and(sa.dom(new LSet().ins(nAsk)).notTest()).
234
235             and(msgAns.eq(nicNotExists))).
```

```
236
237             or(sa.eq(saPrm).
238
239             and(sa.dom(new LSet().ins(nAsk))).
240
241             and(new LPair(nAsk, E).in(sa)).
242             and(E.neq(0)).
243
244             and(msgAns.eq(balanceNotZero))));
245         execute();
246     }
247
248     public void checkbalance(){
249         // sa' : NIC -> Z
250         LMap saPrm = new LMap("SAPRM");
251         tempPost(saPrm.dom(nic).and(saPrm.ran(z).and(saPrm.pfun())));
252         prmVar(sa, saPrm);
253
254         // n? : NIC
255         LVar nAsk = new LVar("NASK");
256         tempPost(nAsk.in(nic));
257         askVar(nAsk);
258
259         // bal! : Z
260         LVar balAns = new LVar("BALANS");
261         tempPost(balAns.in(z));
262         ansVar(balAns);
263
264         // msg! : MSG
265         LVar msgAns = new LVar("MSGANS");
266         tempPost(msgAns.in(msg));
267         ansVar(msgAns);
268
269         // Vincoli
270
271         IntLVar E = new IntLVar();
272
273         tempPost(
274             sa.eq(saPrm).
275             and(sa.dom(new LSet().ins(nAsk))).
276
277             and(new LPair(nAsk, E).in(sa)).
278             and(balAns.eq(E)).
279
280             and(msgAns.eq(ok)).
281
282             or(sa.dom(new LSet().ins(nAsk)).notTest().
283             and(msgAns.eq(nicNotExists))));
284         execute();
```

```
285         }
286 }
```

Conclusioni e lavori futuri

In questa tesi è stata maggiormente formalizzata la procedura di traduzione da notazione Z verso Java + JSetL, descritta da Lorenzo De Santis [1], tramite algoritmi scritti in pseudocodice e sono state apportate modifiche al package jsetl.z volte principalmente a supportare la versione 3.0 di JSetL ed eseguire automaticamente il metodo relativo allo schema iniziale.

L'utilizzo di pseudocodici corredati dalla descrizione di attributi e di metodi presenti al loro interno, a nostro avviso, risulta meno ambigua e di più semplice comprensione.

La libreria *jsetl.z* è stata migliorata implementando l'esecuzione automatica dello schema iniziale, in questo modo si mantiene maggiore fedeltà rispetto alla semantica della notazione Z. Oltre a questa modifica, la libreria è stata aggiornata consentendo la compatibilità con l'ultima versione di JSetL.

In conclusione, riteniamo che in futuro possa essere utile implementare un'interfaccia grafica, che permetta un'interazione con l'utente più semplice, e un programma che prenda effettivamente in input una specifica Z e la traduca in un programma Java + JSetL, in accordo con la procedura che abbiamo descritto.

Ringraziamenti

Ringrazio per avermi offerto questo lavoro di tesi ***il Professor Gianfranco Rossi*** il quale mi ha fornito tutto il supporto tecnico ed informativo necessario, durante il tirocinio, per portare a termine il lavoro di tesi, dandomi spunti di riflessione e risolvendo tutti i miei dubbi in merito a JSetL.

Ringrazio per la disponibilità e il tempo dedicatomi ***il Professor Maximiliano Cristià*** il quale, data la sua notevole competenza in merito alla notazione Z, ha avuto la cortesia di rispondere alle mie domande aiutandomi nella comprensione di questo linguaggio di specifica di un sistema software, fornendomi anche l'idea di eseguire automaticamente lo schema iniziale.

Ringrazio infine ***i miei genitori e i miei amici***, in particolare ***Alessio e Davide***, per il supporto morale durante il periodo di lavoro alla tesi.

Bibliografia

[1] Lorenzo De Santis
Da Specifiche Z a Programmi Java tramite JSetL Tesi di laurea, Università degli Studi di Parma, A.A. 2016-2017

[2] Maximiliano Cristiá
Introduction to the Z notation
Licenciatura en Ciencias de la Computación, Universidad Nacional de Rosario, 2019

[3] Gianfranco Rossi, Elio Panegai, Elisabetta Poleo
JSetL: a Java library for supporting declarative programming in Java
Software Practice & Experience, 2007

[4] Gianfranco Rossi, Roberto Amadini, Andrea Fois
JSetL User's Manual
http://www.clpset.unipr.it/jsetl/downloads/jsetl-3-0-manual.pdf

[5] JSetL online documentation
http://www.clpset.unipr.it/jsetl/jsetl-3-0-1-javadoc/overview-summary.html

[6] Jim Woodcock, Jim Davies
Using Z: Specification, Refinement, and Proof
http://www.usingz.com/usingz.pdf