UNIVERSITÀ
DI PARMA

DIPARTIMENTO DI
SCIENZE MATEMATICHE, FISICHE E
INFORMATICHE

Corso di Laurea in Informatica

Tesi di Laurea

Implementazione della specifica Z del progetto "Tokeneer ID Station" tramite l'utilizzo della liberaria JSetL

Relatore:
Prof. Gianfranco Rossi

Candidato:
Angela Calderini

Anno Accademico 2019/2020

A mia madre Endi, mio fratello Andrea, e alle mie amiche Alessandra, Marina, Loredana e Valentina, senza di voi non avrei raggiunto questo traguardo.

Indice

Introduzione

Scopo principale di questo lavoro di tesi è l'analisi e la codifica, in linguaggio Java, di una specifica Z di un sistema reale, utilizzando la libreria Java JSetL, al fine di ottenere un prototipo eseguibile.
Il sistema considerato è quello del progetto "Tokeneer ID Station" (TIS), progetto commissionato da NSA (National Security Agency) ad Altran e implementato da Praxis High Integrity Systems. Il punto di partenza è la specifica Z di TIS.

Tramite il linguaggio Z è possibile rappresentare un sistema mediante uno stato, e il suo comportamento mediante i cambiamenti di questo stato. Ogni stato è composto da variabili, insiemi e dalla relazioni fra queste. Per descrivere lo stato e i vari cambiamenti la notazione Z si avvale e si basa sulla teoria matematica degli insiemi.

Il linguaggio di implementazione scelto è Java, ma arricchito dalle funzionalità offerte dalla libreria JSetL. JSetL è una libreria sviluppata presso il Dipartimento di Matematica e Informatica dell'Università di Parma, che nasce con l'intenzione di modellare un sistema di programmazione con vincoli, un particolare tipo della programmazione dichiarativa, paradigma che prevede di descrivere la soluzione del problema e le sue proprietà invece che le singole azioni che portano a raggiungerla. In particolare, JSetL offre variabili insiemistiche ed intere, operazioni su insiemi, formulazione e risoluzione di vincoli logici insiemistici e non-determinismo.
La possibilità di operare in JSetL con insiemi e operazioni insiemistiche molto generali rende l'utilizzo di questa libreria particolarmente appropriata per l'implementazione di una specifica Z, diminuendo di fatto il "gap" esistente tra linguaggio di specifica e linguaggio di implementazione.
Inoltre, sfruttando il paradigma della programmazione dichiarativa a vincoli, in cui i vincoli si limitano a specificare le proprietà di cui deve essere dotata la soluzione da trovare piuttosto che specificare azioni singole da eseguire passo-passo, si ottiene un prototipo funzionale in cui si mette in evidenzia

il comportamento e i requisiti utente che deve possedere il sistema ponendo l'attenzione maggiormente su quali procedure deve fare il sistema piuttosto che la modalità utilizzata per svolgere tali procedure.

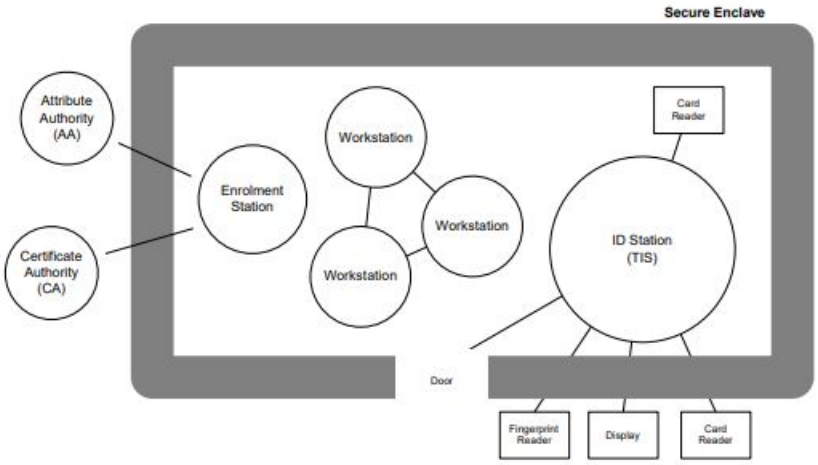
L'elaborato è strutturato come segue. La specifica Z del progetto "Tokeneer ID Station" è introdotta e spiegata nel capitolo 1. JSetL è presentato nel capitolo 2, mentre nel capitolo 3 viene sinteticamente descritto come ottenere, in generale, un programma Java+JSetL da una specifica Z. Il capitolo 4 mostra come è strutturato il prototipo Java analizzando alcune sue operazioni interne. Il capitolo 5 esamina la configurazione iniziale del sistema, e mostra un esempio di esecuzione del prototipo. Infine nel capitolo 6 sono presentate le conclusioni ed i possibili sviluppi futuri.

Capitolo 1

"Tokeneer ID Station" e la sua specifica

Il caso di studio da cui parte il lavoro di tesi è il progetto "Tokeneer ID Station"(TIS), progetto commissionato da NSA (National Security Agency) ad Altran e implementato da Praxis High Integrity Systems, con il fine di dimostrare che è possibile implementare sistemi altamente sicuri e rigorosi offrendo un grande vantaggio in termini di qualità e maggiori garanzie.

TIS è un componente di un sistema più grande già esistente chiamato "Tokeneer System". Il sistema Tokeneer nel suo insieme è formato da un enclave fisicamente sicuro in cui al suo interno sono conservate informazioni su una rete di postazioni di lavoro, da un insieme di componenti del sistema esterni e un insieme di componenti interni all' enclave.



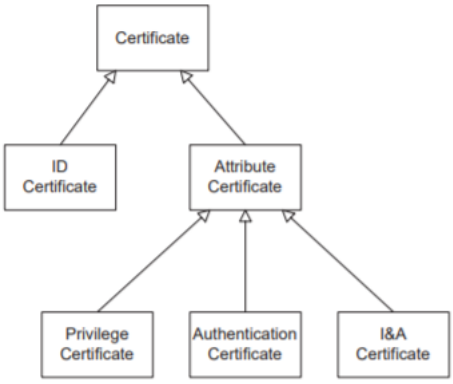
La parte del sistema "Tokeneer System" su cui si concentra il lavoro di tesi è la componente ID Station.
TIS è un'entità autonoma responsabile della verifica dell'utente. Per svolgere questo compito l'utente dovrà essere in possesso di un elemento chiamato "Token" che conterrà al suo interno dei certificati da convalidare. All'interno del Token ci saranno delle informazioni da controllare e verificare. Se l'identificazione ha avuto successo, presupponendo che l'utente disponga di un'autorizzazione sufficiente, TIS permetterà l'accesso all'enclave.

L'ID Station si interfaccia con 4 devices fisici:

- Fingerprint reader
- Token reader
- Door
- Display

Certificati contenuti nel Token e utilizzati per l'autenticazione:

- ID Certificate
- IeA Certificate
- Privilege Certificate
- Authorization Certificate



1.1 La Specifica in Z

La notazione Z è una notazione matematica che utilizza i tipi di dato e i predicati logici con lo scopo di descrivere il comportamento di un sistema realistico. Questo linguaggio di specifica formale utilizza gli schemi per strutturare la specifica in componenti che interagiranno tra di loro per descrivere il sistema nella sua interezza.
La specifica è composta da schemi relativamente semplici, che rappresentano stati e operazioni, utilizzati per costruire operazioni più complesse, dando così una descrizione priva di ambiguità per capire il funzionamento del sistema. Essa non si sofferma sull'implementazione e i vari dettagli ma offre descrizioni in modo astratto delle operazioni e delle componenti che permettono queste operazioni scegliendo un giusto livello di astrazione.

Le operazioni e il comportamento richiesto dell'ID Station è specificato utilizzando gli scenari, che definiscono l'interazione tra l'ID Station e i suoi sistemi collegati.
Ogni scenario si focalizza su un esito positivo, ma copre anche varie condizioni che non consentono di raggiungere il risultato positivo. Quindi l'intero comportamento del sistema, compresi i risultati positivi e falliti, costituiscono i requisiti di sistema.

Nei prossimi paragrafi verranno descritte le componenti principali coinvolte nelle operazioni e lo scenario affrontato per questo lavoro di tesi, ovvero l'operazione di autenticazione chiamata "User Entry".

1.2 Componenti Principali

Ogni schema all'interno della specifica rappresenta uno stato di un determinato componente di TIS, oppure un' operazione che cambia lo stato dei componenti inclusi. A livello macroscopico è possibile definire i due componenti principali che interagiscono all'interno del sistema: *IDStation* e *Real World*.

1.2.1 ID Station

L'*IDStation* rappresenta la componente del sistema che si occupa di prendere in input il Token inserito dall utente e di convalidarlo, autorizzandolo e permettendogli l'accesso all'enclave.

IDStation

$UserToken$
$AdminToken$
$Finger$
$DoorLatchAlarm$
$Floppy$
$Keyboard$
$Config$
$Stats$
$KeyStore$
$Admin$
$AuditLog$
$Internal$

$currentDisplay : DISPLAYMESSAGE$
$currentScreen : Screen$

$status \in \{ gotFinger, waitingFinger, waitingUpdateToken, waitingEntry \} \Rightarrow$
$((\exists\, ValidToken \bullet goodT(\theta ValidToken) = currentUserToken)$
$\vee\, (\exists\, TokenWithValidAuth \bullet goodT(\theta TokenWithValidAuth) = currentUserToken))$

$rolePresent \neq\ \Rightarrow$
$(\exists\, TokenWithValidAuth \bullet goodT(\theta TokenWithValidAuth) = currentAdminToken)$

$enclaveStatus \notin \{ notEnrolled, waitingEnrol, waitingEndEnrol \} \Rightarrow$
$(ownName \neq\ )$

$enclaveStatus \in \{ waitingStartAdminOp, waitingFinishAdminOp \}$
$\Leftrightarrow currentAdminOp \neq$

$(currentAdminOp \neq\ \wedge\ thecurrentAdminOp \in \{ shutdownOp, overrideLock \})$
$\Rightarrow enclaveStatus = waitingStartAdminOp$

$enclaveStatus = gotAdminToken \Rightarrow rolePresent =$

$currentScreen.screenStats = displayStats(\theta Stats)$
$currentScreen.screenConfig = displayConfigData(\theta Config)$

IDStation dispone di alcuni componenti di stato: *Config* rappresenta i dati di configurazione, *AuditLog* tiene traccia di tutte le interazioni che avvengono all' interno del sistema TIS, attraverso la componente *Stats* è possibile tenere traccia del numero di volte in cui è stato permesso o negato l'accesso all'enclave.
DoorLatchAlarm contiene a sua volta degli oggetti che permettono di moni-

torare e controllare la porta dell'enclave e quindi l'accesso ad esso. L'oggetto *DoorLatchAlarm* è composto al suo interno da oggetti che vengono azionati nel caso in cui lo stato del sistema sia potenzialmente insicuro, ad esempio nella circostanza in cui la porta(*Door*) è aperta e la serratura(*latch*) è bloccata.
UserToken e *Finger* sono rappresentazioni interne di *RealWorld*, indicano la presenza di input nelle periferiche del mondo reale(Token Reader e Finger Reader).
Infine l'entità *Internal* contiene oggetti che tengono traccia dell'avanzamento dell'elaborazione.
L'intera Token ID Station è costruita combinando i componenti di stato appena descritti.

1.2.2 Real World

RealWorld è un'entità costituita da componenti esterni all' *IDStation*, a cui dovrà fornire gli input per poter cominciare le varie elaborazioni.

$$RealWorld \mathrel{\widehat=} TISControlledRealWorld \land TISMonitoredRealWorld$$

Dallo schema precedentemente definito si può notare che la componente *RealWorld* è formata da altri due elementi: *TISControlledRealWorld* e *TISMonitoredRealWorld*.

TISControlledRealWorld

$latch : LATCH$
$alarm : ALARM$
$display : DISPLAYMESSAGE$
$screen : Screen$

I componenti di *TISControlledRealWorld* rappresentano le entità del mondo reale controllate da TIS. Il componente *display* si occupa di presentare dei brevi messaggi all' utente al di fuori dell'enclave, è utilizzato per comunicare un mancato inserimento di un token o di un'impronta digitale oppure per l'inserimento di dati non validi. Invece *screen* è usato per comunicare con l'utente all' interno dell'enclave.

TISMonitoredRealWorld

$now : TIME$
$door : DOOR$
$finger : FINGERPRINTTRY$
$userToken, adminToken : TOKENTRY$
$floppy : FLOPPY$
$keyboard : KEYBOARD$

Invece, i componenti di *TISMonitoredRealWorld* rappresentano le entità usate da TIS: *finger* e *userToken* sono le entità più importanti, identificano il token e l'impronta inserita dall'utente, *floppy e keyboard* rappresentano i dispositivi con cui l'utente potrà interagire, *door* rappresenta la porta dell'enclave e *now* rappresenta l'ora attuale.

1.2.3 Token

Il *Token* è un oggetto presente sia in *IDStation* che in *RealWorld*, principalmente è usato nell'operazione di autenticazione dell'utente, è composto da un identificativo e da quattro certificati. Solamente con la convalida di questi ultimi l'utente potrà accedere all' enclave.

Token

$tokenID : TOKENID$

$idCert : IDCert$
$privCert : PrivCert$
$iandACert : IandACert$
$authCert : OptionalAuthCert$

Tutti i certificati all' interno del *Token* sono composti a loro volta da un identificativo, da un periodo di validità e da attributi, alcuni di questi sono comuni a tutti i certificati mentre altri sono attributi che vengono definiti in base alla tipologia di certificato. Come si può notare, l'unico componente che non deve necessariamente essere presente è il certificato di autorizzazione($AuthCert$) dato che è etichettato dalla parola "Optional".

1.3 User Entry

Lo scenario chiamato "UserEntry" coinvolge la ricezione di un token e la sua validazione, la possibilità di leggere e validare informazioni biometriche, scrivere all' interno del Token e infine togliere il blocco alla porta per permettere all' user di accedere.
In ogni fase ci possono essere degli errori e quindi la specifica si occupa di descrivere sia il comportamento di fronte ai successi che agli insuccessi. Di seguito vengono analizzati tutti gli aspetti che riguardano questo procedimento.

Le fasi dell'accesso nell'enclave

- L'utente che vuole avere accesso all' enclave tenta di autenticarsi
- L'utente inserisce il Token dentro il token reader

Prerequisiti

- L'ID Station è in stato "quiescent" (non sono in corso altri tentativi di accesso, modifiche alla configurazione o attività di avvio)
- Utente fuori dall'enclave
- La componente Door è in stato "closed" e "locked"
- Il Token inserito contiene i certificati: ID Certificate, Privilege Certificate e I&A Certificate validi e che quest'ultimo, al suo interno, abbia un modello di impronta digitale che corrisponde all' effettiva impronta dell'utente che vuole accedere
- Il token può non avere l'Authorization Certificate valido

Fasi attraversate

- Inserimento del token
- Lettura dei certificati e dati dal token
- Validazione di ogni certificato
- Lettura dell' impronta digitale
- Confronto dell' impronta digitale all' interno del token con quella inserita dall' utente
- Scrittura di dati nel token
- Scrittura o modifica dell' authorization certificate
- Rimozione del token
- Scrittura di dati nel display
- Settare la porta in "unlocked"
- Apertura porta
- Settare la porta in "locked"
- Chiusura porta

Condizioni di successo

- L'utente ritorna in possesso del token che ha inizialmente inserito.
- Nel token è presente un Authorization Certificate valido e corrente.
- Il token ha gli stessi certificati: ID Certificate, IeA Certificate e Privilege Certificate che aveva inizialmente.
- L'utente è dentro all' enclave.
- La componente door è in stato "close" e "locked".

Condizioni di fallimento

- Token NON inserito in modo corretto.
- Token difettoso.
- Dati all' interno del token non validi.
- Token rimosso prima di aver letto tutte le informazioni al suo interno.

1.3.1 Gli schemi dell'operazione "User Entry"

Lo scenario descritto nei paragrafi precedenti è racchiuso nello schema *TISUserEntryOp* all' interno della specifica Z.

$$TISUserEntryOp \mathrel{\widehat=} TISReadUserToken \lor TISValidateUserToken \lor TISReadFinger \lor TISValidateFinger \lor TISWriteUserToken \lor TISValidateEntry \lor TISUnlockDoor \lor TISCompleteFailedAccess$$

L'operazione *TISUserEntryOp* contiene altri otto schemi che rappresentano l'elaborazione che il sistema dovrà svolgere per autenticare l'utente. Di seguito verranno definiti gli schemi riguardo le operazioni interne di *TISUserEntryOp* e nei capitoli successivi verranno analizzate mostrando anche la loro relativa codifica in Java+JSetL.

$$TISReadUserToken \mathrel{\widehat=} ReadUserToken$$

$$TISValidateUserToken \mathrel{\widehat=} ValidateUserTokenOK \lor ValidateUserTokenFail \lor [\, UserTokenTorn \mid status = gotUserToken]$$

$$TISReadFinger \mathrel{\widehat=} ReadFingerOK \lor FingerTimeout \lor NoFinger \lor [\, UserTokenTorn \mid status = waitingFinger \,]$$

$$TISValidateFinger \mathrel{\widehat=} ValidateFingerOK \lor ValidateFingerFail \lor [\, UserTokenTorn \mid status = gotFinger \,]$$

$$TISWriteUserToken \mathrel{\widehat=} (WriteUserToken \fatsemi UpdateUserToken) \lor [\, UserTokenTorn \mid status = waitingUpdateToken \,]$$

$$TISValidateEntry \mathrel{\widehat=} EntryOK \lor EntryNotAllowed \lor [\, UserTokenTorn \mid status = waitingEntry \,]$$

$$TISUnlockDoor \mathrel{\widehat=} UnlockDoorOK \lor [WaitingTokenRemoval \mid status = waitingRemoveTokenSuccess] \lor TokenRemovalTimeout$$

$$TISCompleteFailedAccess \mathrel{\widehat=} FailedAccessTokenRemoved \lor [WaitingTokenRemoval \mid status = waitingRemoveTokenFail]$$

Capitolo 2

La libreria JSetL

Il principale obiettivo di JSetl è di sviluppare programmi più leggibili e affidabili utilizzando la teoria degli insiemi. L'aspetto interessante dell'utilizzo di astrazioni di insiemi è che il programma che viene scritto può essere molto vicino a una specifica formale basata su insiemi scritta in Z. In questo senso, possiamo vedere il programma finale come una specifica eseguibile basata su insiemi.

2.1 Caratteristiche principali

La libreria JSetL integra nozioni di logica e di insiemistica utilizzando un linguaggio dichiarativo basato su vincoli. JSetL offre strutture tipiche della programmazione logica con vincoli.

- Variabili logiche: possono essere inizializzate e possono assumere qualunque valore. Un valore può essere assegnato come risultato di un vincolo che coinvolge la variabile (ad esempio, un vincolo di uguaglianza), ma se l'assegnazione è già stata fatta non è possibile modificarla.
- Liste e insiemi: sono trattati come variabili ma assumono il valore di una lista o insieme di oggetti; la differenza tra i due tipi di dato è nell'ordine e nelle ripetizioni.
- Unificazione: è utilizzata per controllare l'uguaglianza tra variabili logiche, liste ed insiemi oppure per l'assegnazione di valori se l'oggetto non è inizializzato.
- Vincoli: le varie operazioni ed espressioni vengono svolte attraverso la definizione di vincoli.

- Non determinismo: la risoluzione di vincoli utilizza un algoritmo non deterministico.
- Vincoli definiti da utente: con la classe NewCostraint l'utente può definire nuovi vincoli.

2.2 Classi utilizzate

Di seguito sono elencate le classi utilizzate nel programma appartenenti alla libreria JSetL .

- **Solver** Gli oggetti di questa classe sono risolutori di congiunzioni di vincoli. I vincoli possono essere aggiunti al risolutore che li memorizza. Gli oggetti di tipo `Solver` sono in grado di determinare, se esistono, soluzioni per la data congiunzione di vincoli.

  Esempio:

```
LSet insiemeVar= new LSet("Insieme");
Solver solver= new Solver();
solver.add(insiemeVar.eq(LSet.empty()));
solver.solve();
```

  Nell'esempio appena definito vengono definiti due oggetti, un oggetto `LSet` e un oggetto `Solver`. Attraverso la funzione `add` della classe `Solver` è possibile inserire al suo interno il vincolo in cui l'oggetto identificato come `insiemeVar` deve essere uguale ad un oggetto sempre della classe `LSet` ma vuoto.
  La funzione `solve` permette di risolvere i vincoli contenuti all' interno dell'oggetto `Solver` su cui la funzione è invocata.

- **Constraint** Classe che offre la congiunzione di vincoli atomici.
  Esempio:

```
Constraint c = lista.eq(LList.empty().ins(var)).and
(insieme.eq(LSet.empty().ins(var,varInt)));
```

  L'oggetto `c` di tipo `Constraint` viene inizializzato come congiunzione di due vincoli. Un vincolo in cui `lista` è uguale ad un oggetto di tipo

`LList` contenente un oggetto chiamato `var`, e il vincolo in cui l'oggetto `insieme` deve essere uguale ad un oggetto di tipo `LSet` contenente al suo interno un oggetto identificato come `var` e un elemento identificato come `varInt`.

- **LList** Questa classe implementa liste logiche che possono essere completamente specificate, parzialmente specificate o completamente non specificate. Gli elementi delle liste logiche non devono necessariamente essere dello stesso tipo e possono essere a loro volta oggetti logici.
  Esempio:

```
LList lista= LList.empty().ins(new LSet(),o1,o2);
```

  L'oggetto di tipo `LList` chiamato `lista` viene inizializzato come una lista contenente un oggetto di tipo `LSet` non definito, e due oggetti chiamati `o1 e o2`.

- **LSet** Questa classe implementa insiemi logici, ovvero un tipo di raccolta logica in cui l'ordine e la ripetizione degli elementi non contano. Essi possono essere completamente specificati (tutti i loro elementi sono noti), parzialmente specificati (alcuni dei loro elementi sono sconosciuti) o completamente non specificati; inoltre i suoi componenti possono essere di diverso tipo.
  Esempio:

```
LSet insieme= LSet.empty();
```

  Oggetto di tipo `LSet` inizializzato come insieme vuoto.

- **LVar** Questa classe fornisce l'implementazione per le variabili logiche che possono essere inizializzate con qualsiasi valore o anche non inizializzate.
  Esempio:

```
LVar var= new LVar("Variabile","abcdef");
```

  Oggetto di tipo `Lvar` inizializzato come variabile logica con nominativo "Variabile" e valore "abcdef".

- **IntLvar** Gli oggetti di questa classe sono variabili logiche i cui valori possono essere solo numeri interi. Questi oggetti supportano vincoli aritmetici comuni come moltiplicazione, sottrazione, modulo ecc.
  Esempio:

```
IntLVar varInt= new IntLVar("Variabile",53);
```

  Oggetto di tipo `IntLvar` inizializzato come variabile logica intera con nominativo "Variabile" e valore 53.

- **IntLset** Insiemi logici i cui elementi possono essere solo numeri interi (o variabili logiche con valori interi, ovvero `IntLVar`).
  Esempio:

```
IntLSet intSet= IntLSet.empty().ins(1,2,3);
```

  Oggetto di tipo `IntLSet` inizializzato come insieme logico composto dai valori 1,2 e 3.

Capitolo 3

Da Z a Java

Scrivere un programma in linguaggio Java dalla specifica Z utilizzando i vincoli e le strutture dati insiemistiche messe a disposizione dal JSetL è molto più semplice e da punto di vista semantico più equivalente piuttosto che utilizzare soltanto i costrutti standard di Java. La tecnica utilizzata è più simile ad una codifica che ad una implementazione perché utilizzare i vincoli e le strutture di JSetL ci permette di codificare tutti gli operatori logici, insiemistici e relazionali utilizzati nella specifica di TIS.

3.1 Un esempio di codifica

Di seguito, un esempio semplice di codifica da Z a Java+JSetL come dimostrazione:

ValidToken

Token

$privCert.baseCertId = idCert.id$
$iandACert.baseCertId = idCert.id$

$privCert.tokenID = tokenID$
$iandACert.tokenID = tokenID$

```
public static Constraint validToken(LList token) {

    LVar tokenID= new LVar();
    LVar id= new LVar();
    LList l1= LList.empty().ins(id,new IntLSet(),new LSet());
    LList l2= LList.empty().ins(new LVar(),new LVar());
    LList idCert= LList.empty().ins(l1,l2);
    IntLSet pValidPeriod= new IntLSet();
    LVar  pBaseCertId= new LVar(),pTokenId= new LVar();
    LList privCert= LList.empty().ins(new LVar(),pValidPeriod,new
              LSet(),pBaseCertId,pTokenId,new LVar(),new LSet());

    IntLSet iValidPeriod= new IntLSet();
    LVar iBaseCertId= new LVar(),iTokenId= new LVar();
    LList iandaCert= LList.empty().ins(new LVar(),iValidPeriod,new
               LSet(),iBaseCertId,iTokenId,new LVar());
    LSet authCert= new LSet();
    Solver solver= new Solver();
    solver.add(token.eq(LList.empty().ins(tokenID,idCert,
    privCert,iandaCert,authCert)));
    solver.add(pBaseCertId.eq(id));
    solver.add(iBaseCertId.eq(id));
    solver.add(pTokenId.eq(tokenID));
    solver.add(iTokenId.eq(tokenID));

    return solver.getConstraint();
  }
```

Nella prima parte possiamo vedere la definizione dell'operazione *ValidToken* in notazione Z e subito sotto la sua corrispondente implementazione in Java+JSetL. La prima cosa che possiamo notare è che in generale, la codifica in Java+JSetL è più lunga del codice Z corrispondente, infatti in ogni metodo si dovrà definire al suo interno la struttura dell'oggetto che viene passato nei parametri per utilizzare le componenti al suo interno.
Il metodo `validToken` ritorna un oggetto di tipo `Constraint`. Questo tipo di dato rappresenta il vincolo, e racchiude tutti i vincoli che il `solver` ha memorizzato all' interno della funzione.
Un altro fattore interessante è che nella specifica lo schema include un ulteriore schema: *Token*. Nel metodo l'entità `Token` viene passato come argomento di tipo `LList`.

Dato che come abbiamo visto precedentemente *Token* è uno schema che dichiara cinque variabili, esso è codificato come una lista composta da 5 elementi e la corrispondenza tra le variabili Z e quelle di JSetL viene implementato in base all'ordine di dichiarazione.

Un token è valido se tutti i certificati su di esso sono strutturati correttamente e ogni certificato fa un riferimento incrociato all' ID certificate e all' ID token.

3.2 Regole generali per la codifica

In generale ogni schema in Z è codificato come metodo in Java e possiamo individuare delle semplici regole da applicare per effettuare la codifica.

- Le variabili dichiarate nello schema diventano parametri della funzione e solitamente se lo schema dichiara variabili attraverso l'inclusione di un altro schema, allora gli argomenti del metodo sono tali schemi anziché le variabili dichiarate al loro interno.
- Gli schemi che non hanno al loro interno il predicato sono codificati come Liste attraverso la dichiarazione dell'oggetto `LList` fornito dalla libreria JSetL.
- Il metodo ritorna sempre un oggetto di tipo `Constraint`.
- Viene definito un oggetto di tipo `Solver` che si occuperà di "immagazzinare" i vincoli definiti all' interno della funzione e di restituire come risultato della funzione un oggetto `Constraint` che racchiuderà tutti i vincoli al suo interno.
- Le componenti di un' entità che non vengono utilizzate, vengono "nascoste" definendo il tipo in modo generico, ad esempio come `new LList(), new LSet()o new LVar()`.
- Le assegnazioni avvengono tramite unificazione: l'oggetto `Solver` definito nel metodo forza l'unificazione tramite un vincolo di uguaglianza (Esempio: `solver.add(token.eq(LList.empty().ins(tokenID,idCert, privCert,iandaCert,authCert)))`).
  Nel caso in cui ciò non sia vero, la clausola avrà esito negativo.

Capitolo 4

Autenticazione dell'utente

In questo capitolo verranno analizzate le operazioni principali che permettono l'autorizzazione dell'utente. Per ogni operazione sarà specificato il suo corrispondete codice in Z attraverso la definizione del suo schema e la sua implementazione in Java+JSetL.

4.1 L'operazione "UserEntryOp"

UserEntryOp è l'operazione che racchiude tutte le altre operazioni che permettono al sistema di verificare e di conseguenza di permettere l'accesso all' utente, oppure di non permetterlo a causa di errori o dati errati.
Il processo di autenticazione completo viene attivato dall' inserimento del token da parte dell'utente e la conseguente lettura di esso da parte di TIS.
Questo procedimento coinvolge la lettura e validazione del token, lettura e validazione dell'impronta digitale e la scrittura del certificato di autorizzazione nel token.
Se queste fasi saranno soddisfatte allora l'utente potrà togliere il token dal lettore e successivamente la porta dell'enclave si aprirà.
Nel caso si verificasse un errore, il sistema dovrà tornare in uno stato in cui possa ammettere l'inserimento di un token da parte di un altro utente e quindi ricominciare il processo di autenticazione.

$$TISUserEntryOp \mathrel{\widehat{=}} TISReadUserToken \lor TISValidateUserToken \lor TISReadFinger \lor TISValidateFinger \lor TISWriteUserToken \lor TISValidateEntry \lor TISUnlockDoor \lor TISCompleteFailedAccess$$

```
public static Constraint tisUserEntryOp(LList idStation,LList
    realWorld,LList realWorld_,LList idStation_) {

        Solver solver= new Solver();

        LList realWorld1= new LList();
        LList idStation1= new LList();
        LList realWorld2= new LList();
        LList idStation2= new LList();
        LList realWorld3= new LList();
        LList idStation3= new LList();
        LList realWorld4= new LList();
        LList idStation4= new LList();
        LList realWorld5= new LList();
        LList idStation5= new LList();
        LList realWorld6= new LList();
        LList idStation6= new LList();

        solver.add(tisReadUserToken(idStation, realWorld,
            realWorld1, idStation1).or
            (tisValidateUserToken(idStation1, realWorld1,
                realWorld2, idStation2)).or
            (tisReadFinger(idStation2, realWorld2, realWorld3,
                idStation3)).or
            (tisValidateFinger(idStation3, realWorld3, realWorld4,
                idStation4)).or
            (tisWriteUserToken(idStation4, realWorld4, realWorld5,
                idStation5)).or
            (tisValidateEntry(idStation5, realWorld5, realWorld6,
                idStation6)).or
            (tisUnlockDoor(idStation6, realWorld6, realWorld_,
                idStation_)).or
            (tisCompleteFailedAccess(idStation, realWorld,
                realWorld_, idStation_)));
        return solver.getConstraint();


    }
```

Gli argomenti della funzione `TisUserEntryOp` sono `idStation` e `realWorld`. Essi rappresentano rispettivamente lo stato dell'ID Station e di Real World all' inizio della procedura, mentre `idStation_` e `realWorld_` rappresentano sempre le due entità, ma allo stato finale, quindi al termine dell'operazione di autenticazione. All'interno del `solver` viene definito il vincolo in cui tutte le varie fasi vengono legate tra loro attraverso operazioni di disgiunzione. Analogamente anche le operazioni all' interno di `tisUserEntryOp` sono strutturate come appena detto in precedenza: gli argomenti di tutti i metodi riguardano lo stato iniziale e finale delle entità ID Station e Real World.

4.2 Lettura del token

Partendo dalla prima operazione di *UserEntryOp* analizziamo *TisReadUserToken* che nella specifica è dichiarata come di seguito:

$$TISReadUserToken \mathrel{\widehat{=}} ReadUserToken$$

ReadUserToken

$$UserEntryContext$$

$$\Xi UserToken$$
$$\Xi DoorLatchAlarm$$
$$\Xi Stats$$

$$AddElementsToLog$$

$$enclaveStatus \in \{enclaveQuiescent, waitingRemoveAdminTokenFail\}$$

$$status = quiescent$$
$$userTokenPresence = present$$

$$currentDisplay' = wait$$
$$status' = gotUserToken$$

Come si può notare l'operazione è definita tramite lo schema *ReadUserToken*. L'operazione "UserEntry" viene avviata quando TIS si trova nello stato *quiescent* e rileva la presenza di un *Token* nel token reader (che risiede all'esterno dell'enclave). Un'operazione di autorizzazione dell'utente può iniziare mentre *enclaveStatus* ha valore *enclaveQuiescent* o l'enclave è in attesa della rimozione di un token (*waitingRemoveAdminTokenFail*).
Quando il token utente viene rilevato per la prima volta, la sua presenza viene controllata e lo stato interno cambia.

```
public static Constraint tisReadUserToken(LList idStation,LList
    realWorld,LList realWorld_,LList idStation_) {

      Solver s= new Solver();
      s.add(readUserToken(idStation, realWorld, realWorld_,
          idStation_));
      return s.getConstraint();

    }
```

```
public static Constraint readUserToken(LList idStation,LList
    realWorld,LList realWorld_,LList idStation_) {

      LVar userTokenPresence= new LVar("present");
      LList userToken= LList.empty().ins(new
          LList(),userTokenPresence);
      LList doorLatchAlarm= new LList();
      LList config= new LList();
      LList stats= new LList();
      LVar status=new LVar("quiescent");
      LVar enclaveStatus= new LVar();
      LList internal= LList.empty().ins(status,enclaveStatus,new
          IntLVar());
      LVar currentDisplay= new LVar();
      LVar screenStats= new LVar(),screenConfig=new LVar();
      LList currentScreen= LList.empty().ins(screenStats,new
          LVar(),screenConfig);
      LList auditLog= new LList();
      LList userToken_= new LList();
      LList doorLatchAlarm_= new LList();
      LList currentFloppy_= new LList();
      LVar floppyPresence_= new LVar();
      LList config_= new LList();
      LList stats_= new LList();
      LVar status_=new LVar();
      LList internal_= LList.empty().ins(status_,new LVar(),new
          IntLVar());
      LVar currentDisplay_= new LVar();
      LVar screenStats_= new LVar(),screenConfig_=new LVar();
```

```
    LList currentScreen_= LList.empty().ins(screenStats_,new
        LVar(),screenConfig_);
    LList auditLog_= new LList();
    LList TISControlledRealWorld_= new LList();
    LList TISControlledRealWorld= new LList();
    LList tisFloppy= new LList();

    LList TISMonitoredRealWorld_= LList.empty().ins(new
        IntLVar(),new LVar(),new LList(),new LList(),
        new LList(),tisFloppy,new LList());

    Solver solver= new Solver();
    solver.add(idStation.eq(LList.empty().ins(userToken,new
        LList(),new LList(),
          doorLatchAlarm,new LList(),new LList(),config,stats,new
              LList(),new LList(),auditLog,internal,
          new LVar(),currentScreen)));
    solver.add(idStation_.eq(LList.empty().ins(userToken_,new
        LList(),new LList(),
          doorLatchAlarm_,new LList(),new LList(),new
              LList(),stats_,new LList(),new
              LList(),auditLog_,internal_,
          currentDisplay_,currentScreen_)));

    solver.add(userEntryContext(idStation, realWorld,realWorld_,
        idStation_));

solver.add(userToken_.eq(userToken).and(doorLatchAlarm_.eq(doorLatchAlarm)).and
    (stats_.eq(stats)));

    solver.add(screenStats.eq(displayStats(stats)).and
    (screenConfig.eq(displayConfigData(config))));

    solver.add(enclaveStatus.in(LSet.empty().ins("enclaveQuiescent",
    "waitingRemoveAdminTokenFail")));
    solver.add(status.eq("quiescent"));
    solver.add(userTokenPresence.eq("present"));
    solver.add(currentDisplay_.eq("wait"));
    solver.add(status_.eq("gotUserToken"));
    solver.add(screenStats_.eq(screenStats).and(screenConfig_.eq(screenConfig)));

      return solver.getConstraint();
    }
```

Come è stato detto nel capito riguardo la codifica da Z a Java+JSetL, in ogni metodo è necessario definire la struttura interna delle variabili che sono state passate come argomenti della funzione. Quindi è stato necessario definire le variabili che vengono utilizzate nell'operazione `readUserToken`, fare assegnazioni tramite vincolo di uguaglianza, e poi usare tali variabili per definire i vincoli necessari per lo svolgimento dell'operazione.
Gli ultimi quattro vincoli aggiunti dal `solver` tramite la funzione `add` che coinvolgono `status, userTokenPresence, currentDispaly_` e `status_` rappresentano la parte del predicato che è presente nello schema Z *ReadUserToken*.

4.3 Validazione del token

Una volta che TIS ha letto un token utente, deve convalidare il suo contenuto. Questo procedimento può essere svolto in due modi:

- Un token utente è valido per l'ingresso nell'enclave, senza la necessità di controlli biometrici se: il token contiene un Authorization certificate che sia correttamente congruente con l'ID token e l'ID certificate, ha un periodo di validità corretto, e sia il certificato di autorizzazione che il certificato ID possono essere convalidati utilizzando le chiavi contenute nel KeyStore di TIS. Nel caso in cui vi sia un certificato di autorizzazione valido, i controlli biometrici vengono ignorati e si passerà direttamente alla validazione dell'entrata.

- Un token utente è valido per l'ingresso nell'enclave se: il token è coerente, ha un periodo di validità corretto, ed è possibile convalidare il Privilege certificate e il I&A Certificate. Questo indipendentemente dalla presenza o dallo stato del certificato di autorizzazione. Tuttavia, in questa circostanza saranno necessari controlli biometrici.

Quindi i controlli biometrici sono necessari solo se il certificato di autorizzazione non è presente o non è valido. In questo caso i certificati rimanenti devono essere controllati.

$$TISValidateUserToken \mathrel{\widehat=} ValidateUserTokenOK \lor ValidateUserTokenFail \lor [\, UserTokenTorn \mid status = gotUserToken]$$

```
public static Constraint tisValidateUserToken(LList
    idStation,LList realWorld,LList realWorld_,LList idStation_) {

        Solver solver= new Solver();
        LVar status= new LVar();
        LList internal= LList.empty().ins(status,new LVar(),new
            IntLVar());

        solver.add(idStation.eq(LList.empty().ins(new LList(),new
            LList(),new LList(),new LList(),new LList(),new LList(),
            new LList(),new LList(),new LList(),new LList(),new
                LList(),internal,new LVar(),new LList())));

      solver.add((validateUserTokenOK(idStation, realWorld,
          realWorld_, idStation_)).or
          (validateUserTokenFail(idStation, realWorld, realWorld_,
              idStation_)).or
            (userTokenTorn(idStation, realWorld, realWorld_,
                idStation_)));

      solver.add(status.eq("gotUserToken"));
      return solver.getConstraint();

      }
```

La definizione in Z di *TISValidateUserToken* corrisponde al metodo Java `tisValidateUserToken`.
Nello schema l'operazione di validazione è formato da tre diverse operazioni legate tra loro: *ValidateUserTokenOK*, *ValidateUserTokenFail* e *UserTokenTorn*.
Nel metodo Java è stata fatta la stessa operazione: sono state legate le tre funzioni `validateUserTokenOK`, `validateUserTokenFail` e `userTokenTorn` tramite disgiunzione(or). Attraverso l'oggetto `solver` è stato aggiunto il vincolo in cui `status`, entità che rappresenta lo stato attuale del sistema, deve assumere il valore `gotUserToken`, come specificato anche nel codice in Z.
La funzione `validateUserTokenOK` rappresenta lo scenario in cui avviene la corretta validazione del token, `validateUserTokenFail` rappresenta lo stato in cui il token non viene validato correttamente e infine `userTokenTorn` definisce la circostanza in cui l'utente estrae il token prima che tutti i dati al suo interno vengano letti.

$$ValidateUserTokenOK \mathrel{\widehat{=}} BioCheckRequired \lor BioCheckNotRequired$$

```
public static Constraint validateUserTokenOK(LList idStation,LList
    realWorld,LList realWorld_,LList idStation_) {

    Solver solver= new Solver();
    solver.add((bioCheckRequired(idStation, realWorld, realWorld_,
        idStation_)).or
      (bioCheckNotRequired(idStation, realWorld, realWorld_,
          idStation_)));

    return solver.getConstraint();
  }
```

Lo schema *ValidateUserTokenOK* comprende due operazioni: *BioCheckRequirede* e *BioCheckNotRequired*, procedura che è stata fatta anche nel codice Java+JSetL.
Il metodo Java `bioCheckRequired`, che verrà chiamato dall'oggetto `Solver`, restituirà un vincolo che verrà soddisfatto nel caso in cui il token, che si trova all' interno dell'entità `IDStation`, abbia tutti i certificati validi, tralasciando il certificato di autorizzazione, in questo caso si passerà alla fase dei controlli biometrici.
Il metodo Java `bioCheckNotRequired` restituirà un vincolo che verrà soddisfatto nel caso in cui il token abbia tutti i certificati validi, compreso certificato di autorizzazione, in questo caso si passerà direttamente alla fase di validazione dell'entrata.

ValidateUserTokenFail

$UserEntryContext$

$\Xi UserToken$
$\Xi DoorLatchAlarm$
$\Xi Stats$

$AddElementsToLog$

$status = gotUserToken$
$userTokenPresence = present$

$\neg\ UserTokenWithOKAuthCert \land \neg\ UserTokenOK$

$currentDisplay' = removeToken$
$status' = waitingRemoveTokenFail$

```
public static Constraint validateUserTokenFail(LList
    idStation,LList realWorld,LList realWorld_,LList idStation_) {
        LVar userTokenPresence= new LVar();

        LList userToken= LList.empty().ins(new
            LList(),userTokenPresence);
        LList config= new LList(),stats= new LList(),keyStore= new
            LList();
        IntLVar currentTime= new IntLVar();
        LList doorLatchAlarm= LList.empty().ins(currentTime,new
            LVar(),new LVar(),new LVar(),new LVar(),new LVar());
        LVar status= new LVar();
        LList internal= LList.empty().ins(status,new LVar(),new
            IntLVar());
        LVar screenStats= new LVar();
        LVar screenConfig= new LVar();
        LList currentScreen= LList.empty().ins(screenStats,new
            LVar(),screenConfig);
        LList auditLog= new LList();
        IntLVar now= new IntLVar();
        LList tisMonitoredRealWorld= LList.empty().ins(now,new
            LVar(),new LList(),new LList(),new LList(),new LList());
        LList userToken_= new LList();
        LList doorLatchAlarm_= new LList();
        LList stats_= new LList();
        LList auditLog_= new LList();
```

```
LVar status_= new LVar();
LList internal_= LList.empty().ins(status_,new LVar(),new
    IntLVar());
LVar screenStats_= new LVar();
LVar screenConfig_= new LVar();
LVar currentDisplay_= new LVar();
LList currentScreen_= LList.empty().ins(screenStats_,new
    LVar(),screenConfig_);
Solver solver= new Solver();
solver.add(idStation.eq(LList.empty().ins(userToken,new
    LList(),new LList(),doorLatchAlarm,new LList(),new
    LList(),
      config,stats,keyStore,new
          LList(),auditLog,internal,new
          LVar(),currentScreen)));
solver.add(realWorld.eq(LList.empty().ins(new
    LList(),tisMonitoredRealWorld)));

solver.add(idStation_.eq(LList.empty().ins(userToken_,new
    LList(),new LList(),doorLatchAlarm_,new LList(),new
    LList(),
      new LList(),stats_,new LList(),new
          LList(),auditLog_,internal_,currentDisplay_,currentScreen_)));

  userEntryContext(idStation, realWorld, realWorld_,
      idStation_);
  solver.add(userToken_.eq(userToken));
  solver.add(doorLatchAlarm_.eq(doorLatchAlarm));
  solver.add(stats_.eq(stats));
  solver.add(screenStats.eq(displayStats(stats)));
  solver.add(screenConfig.eq(displayConfig(config)));
  solver.add(status.eq("gotUserToken"));
  solver.add(userTokenPresence.eq("present"));
  solver.add(currentDisplay_.eq("removeToken"));
  solver.add(status_.eq("waitingRemoveTokenFail"));
  solver.add(screenStats_.eq(screenStats));
  solver.add(screenConfig_.eq(screenConfig));
  return solver.getConstraint();

}
```

Guardando lo schema *ValidateUserTokenFail* si nota che nel caso in cui l'o-

perazione di validazione non abbia avuto esito positivo, la maggior parte dei componenti di TIS rimangono inalterati, solamente le componenti *currentDisplay* e *status* cambiano, segnalando all' utente di rimuovere il token e cambiando lo stato di TIS in *waitingRemoveTokenFail*. Anche nel metodo Java `validateUserTokenFail` è stato fatto lo stesso procedimento: i vincoli di uguaglianza impongo che i componenti che devono rimanere invariati(`userToken, doorLatchAlarm, stats`) rimangano uguali nel loro stato successivo(`userToken_, doorLatchAlarm_, stats_`). Attraverso l'utilizzo di vincoli vengono cambiati anche i valori delle componenti `status_` e `dispaly_`, che rappresentano lo stato successivo delle componenti `status` e `dispaly`.

4.4 Lettura dell'impronta

Un'impronta verrà letta se il sistema è attualmente in attesa dell'oggetto Finger e il Token utente è stato rilevato, quindi presente.
TIS potrebbe attendere di ottenere un'impronta valida per troppo tempo, in questo caso l'utente è invitato a rimuovere il token e l'operazione viene terminata senza successo.

$$TISReadFinger \mathrel{\widehat=} ReadFingerOK \lor FingerTimeout \lor NoFinger \lor [\, UserTokenTorn \mid status = waitingFinger \,]$$

```
public static Constraint tisReadFinger(LList idStation,LList
    realWorld,LList realWorld_,LList idStation_) {

      Solver solver= new Solver();
      LVar status= new LVar();

      LList internal= LList.empty().ins(status,new LVar(),new
          IntLVar());

      solver.add(idStation.eq(LList.empty().ins(new LList(),new
          LList(),new LList(),new LList(),new LList(),new LList(),
          new LList(),new LList(),new LList(),new LList(),new
              LList(),internal,new LVar(),new LList())));
      solver.add((readFingerOk(idStation, realWorld, realWorld_,
          idStation_)).or
          (fingerTimeout(idStation, realWorld, realWorld_,
              idStation_)).or
          (userTokenTorn(idStation, realWorld, realWorld_,
              idStation_)));
      solver.add(status.eq("waitingFinger"));
      return solver.getConstraint();

  }
```

L'operazione di lettura dell'impronta digitale è formata da tre diverse operazioni legate tra loro: *ReadFingerOK*, *FingerTimeout* e *UserTokenTorn*.
Nel metodo Java si ha analogamente le tre funzioni definite nel codice in Z: `readFingerOk`, `fingerTimeout` e `userTokenTorn`, legate tramite disgiunzione(or). In questo caso status deve essere assumere il valore `waitingFinger`, come specificato anche nel codice in Z.

La funzione `readFingerOk` rappresenta lo scenario in cui avviene la corretta lettura del impronta digitale, `fingerTimeout` rappresenta lo stato in cui il sistema aspetta per un tempo indefinito l'inserimento dell'impronta da parte dell'utente. Infine `userTokenTorn` definisce la circostanza in cui l'utente estrae il token prima che tutti i dati al suo interno vengano letti come è stato illustrato in precedenza.

ReadFingerOK

$UserEntryContext$

$\Xi DoorLatchAlarm$
$\Xi UserToken$
$\Xi Stats$

$AddElementsToLog$

$status = waitingFinger$
$fingerPresence = present$
$userTokenPresence = present$

$currentDisplay' = wait$
$status' = gotFinger$

```
public static Constraint readFingerOk(LList idStation,LList
    realWorld,LList realWorld_,LList idStation_) {

    LVar userTokenPresence= new LVar();
    LList userToken= LList.empty().ins(new
        LList(),userTokenPresence);
    LVar fingerPresence= new LVar();
    LList finger= LList.empty().ins(new LList(),fingerPresence);
    LList config= new LList(),stats= new LList(),keyStore= new
        LList();
    IntLVar currentTime= new IntLVar();
    LList doorLatchAlarm= LList.empty().ins(currentTime,new
        LVar(),new LVar(),new LVar(),new LVar(),new LVar());
    LVar status= new LVar();
    LList internal= LList.empty().ins(status,new LVar(),new
        IntLVar());
    LVar screenStats= new LVar();
    LVar screenConfig= new LVar();
    LList currentScreen= LList.empty().ins(screenStats,new
        LVar(),screenConfig);
```

```
LList auditLog= new LList();
LList userToken_= new LList();
LList doorLatchAlarm_= new LList();
LList stats_= new LList();
LList auditLog_= new LList();
LVar status_= new LVar();
LList internal_= LList.empty().ins(status_,new LVar(),new
    IntLVar());
LVar screenStats_= new LVar();
LVar screenConfig_= new LVar();
LVar currentDisplay_= new LVar();
LList currentScreen_= LList.empty().ins(screenStats_,new
    LVar(),screenConfig_);

Solver solver= new Solver();
  solver.add(idStation.eq(LList.empty().ins(userToken,new
      LList(),finger,doorLatchAlarm,new LList(),new LList(),
      config,stats,keyStore,new LList(),auditLog,internal,new
          LVar(),currentScreen)));
  solver.add(idStation_.eq(LList.empty().ins(userToken_,new
      LList(),new LList(),doorLatchAlarm_,new LList(),new
      LList(),
      new LList(),stats_,new LList(),new
          LList(),auditLog_,internal_,currentDisplay_,currentScreen_)));
  solver.add(userEntryContext(idStation, realWorld, realWorld_,
      idStation_));
  solver.add(userToken_.eq(userToken));
  solver.add(doorLatchAlarm_.eq(doorLatchAlarm));
  solver.add(stats_.eq(stats));
  solver.add(screenStats.eq(displayStats(stats)));
  solver.add(screenConfig.eq(displayConfig(config)));
  solver.add(status.eq("waitingFinger"));
  solver.add(userTokenPresence.eq("present"));
  solver.add(fingerPresence.eq("present"));
  solver.add(userTokenWithOKAuthCert(keyStore, userToken,
      currentTime));
  solver.add(status_.eq("gotFinger"));
  solver.add(currentDisplay_.eq("wait"));
  solver.add(screenStats_.eq(screenStats));
  solver.add(screenConfig_.eq(screenConfig));

  return solver.getConstraint();
```

```
}
```

Analizzando lo schema *ReadFingerOK* possiamo osservare che i componenti *DoorLatchAlarm*, *UserToken* e *Stats* rimangono invariati, quindi il loro stato con cambia. L'entità *status* all' inizio dell'operazione dovrà avere il valore *waitingFinger* mentre allo stato finale cambierà in *gotFinger* perché si presuppone che la lettura dell'impronta sia avvenuta. Analogamente nel codice in Java+JSetL del metodo `readFingerOk`, tramite l'utilizzo di vincoli viene imposto che `status`, sia uguale a `waitingFinger` e `status_`, che è la sua condizione successiva, uguale a `gotFinger`. Sempre all' interno della funzione `readFingerOk` viene imposto il vincolo in cui sia `userTokenPresence` che `fingerPresence` dovranno essere valorizzati con `present` dato che sono variabili che rappresentano la rilevazione della presenza di un token o di una impronta nell'apposito lettore.
L'entità token, come quella finger possono essere inserite o rimosse dal sistema, ma tramite le variabili `tokenPresence` e `fingerPresence` posso monitorarle, quando saranno valorizzate come `present` saranno presenti, invece quando assumeranno il valore `absent` saranno assenti.

4.5 Validazione dell'impronta

Un'impronta ha una sua rappresentazione all'interno di TIS, essa deve corrispondere alle informazioni contenute nell'`userToken` affinché sia considerata accettabile e corretta. L'impronta digitale convalidata correttamente è un prerequisito per generare un certificato di autorizzazione e aggiungerlo al token utente ma se questo non accade viene chiesto di rimuovere il token.

$$TISValidateFinger \mathrel{\widehat{=}} ValidateFingerOK \lor ValidateFingerFail \lor [\, UserTokenTorn \mid status = gotFinger \,]$$

```
public static Constraint tisValidateFinger(LList idStation,LList
    realWorld,LList realWorld_,LList idStation_) {

      Solver solver= new Solver();
      LVar status= new LVar();
      LList internal= LList.empty().ins(status,new LVar(),new
          IntLVar());
      solver.add(idStation.eq(LList.empty().ins(new LList(),new
          LList(),new LList(),new LList(),new LList(),new LList(),
```

```
        new LList(),new LList(),new LList(),new LList(),new
            LList(),internal,new LVar(),new LList())));
    solver.add((validateFingerOK(idStation, realWorld,
        realWorld_, idStation_)).or
        (validateFingerFail(idStation, realWorld, realWorld_,
            idStation_)).or
        (userTokenTorn(idStation, realWorld, realWorld_,
            idStation_)));

    solver.add(status.eq("gotFinger"));

    return solver.getConstraint();

}

}
```

L'operazione di validazione dell'impronta digitale rappresentata dallo schema *TISValidateFinger* è formata da tre diverse operazioni legate tra loro: *ValidateFingerOK*, *ValidateFingerFail* e *UserTokenTorn*.
Nel metodo Java si hanno analogamente le tre funzioni definite nel codice in Z: `validateFingerOK`, `validateFingerFail` e `userTokenTorn`, legate tramite disgiunzione(or). In questo caso `status` deve essere assumere il valore `gotFinger`, come specificato anche nel codice in Z.
La funzione `validateFingerOK` rappresenta lo scenario in cui avviene la corretta validazione dell'impronta digitale e `ValidateFingerFail` rappresenta lo stato in cui il sistema non riesce a validare l'impronta.

ValidateFingerOK

$UserEntryContext$

$\Xi DoorLatchAlarm$
$\Xi UserToken$

$AddSuccessfulBioCheckToStats$
$AddElementsToLog$

$status = gotFinger$
$userTokenPresence = present$

$FingerOK$

$status' = waitingUpdateToken$
$currentDisplay' = wait$

```
public static Constraint validateFingerOK(LList idStation,LList
    realWorld,LList realWorld_,LList idStation_) {
    LVar userTokenPresence= new LVar();
    LList userToken= LList.empty().ins(new
        LList(),userTokenPresence);
    LVar fingerPresence= new LVar();
    LList finger= LList.empty().ins(new LList(),fingerPresence);
    LList config= new LList(),stats= new LList(),keyStore= new
        LList();
    IntLVar currentTime= new IntLVar();
    LList doorLatchAlarm= LList.empty().ins(currentTime,new
        LVar(),new LVar(),new LVar(),new LVar(),new LVar());
    LVar status= new LVar();
    LList internal= LList.empty().ins(status,new LVar(),new
        IntLVar());
    LVar screenStats= new LVar();
    LVar screenConfig= new LVar();
    LList currentScreen= LList.empty().ins(screenStats,new
        LVar(),screenConfig);
    LList auditLog= new LList();
    LList userToken_= new LList();
    LList doorLatchAlarm_= new LList();
    LList stats_= new LList();
    LList auditLog_= new LList();
    LVar status_= new LVar();
```

```
    LList internal_= LList.empty().ins(status_,new LVar(),new
        IntLVar());
    LVar screenStats_= new LVar();
    LVar screenConfig_= new LVar();
    LVar currentDisplay_= new LVar();
    LList currentScreen_= LList.empty().ins(screenStats_,new
        LVar(),screenConfig_);

      Solver solver= new Solver();
      solver.add(idStation.eq(LList.empty().ins(userToken,new
          LList(),finger,doorLatchAlarm,new LList(),new LList(),
          config,stats,keyStore,new LList(),auditLog,internal,new
              LVar(),currentScreen)));

      solver.add(idStation_.eq(LList.empty().ins(userToken_,new
          LList(),new LList(),doorLatchAlarm_,new LList(),new
          LList(),
          new LList(),stats_,new LList(),new
              LList(),auditLog_,internal_,currentDisplay_,currentScreen_)));

      userEntryContext(idStation, realWorld, realWorld_,
          idStation_);
      solver.add(userToken_.eq(userToken));
      solver.add(doorLatchAlarm_.eq(doorLatchAlarm));
          solver.add(addSuccessfulBioCheckToStats(stats,stats_));
      solver.add(screenStats.eq(displayStats(stats)));
      solver.add(screenConfig.eq(displayConfig(config)));
      solver.add(status.eq("gotFinger"));
      solver.add(userTokenPresence.eq("present"));
      fingerOk(finger, userToken);
      solver.add(status_.eq("waitingUpdateToken"));
      solver.add(currentDisplay_.eq("wait"));
      solver.add(screenStats_.eq(displayStats(stats_)));
      solver.add(screenConfig_.eq(screenConfig));

      return solver.getConstraint();


  }
```

Analizzando lo schema *ValidateFingerOK* possiamo osservare che i componenti *DoorLatchAlarm* e *UserToken* rimangono invariati, quindi il loro stato

non cambia. Il componente *status* all'inizio dell'operazione dovrà assumere il valore *gotFinger* mentre allo stato finale cambierà in *waitingUpdateToken* perché il sistema dovrà scrivere il certificato di autorizzazione nel token. Analogamente nel codice in Java del metodo `validateFingerOk`, tramite l'utilizzo di vincoli viene imposto che `status`, sia uguale a `gotFinger` e `status_` uguale a `waitingUpdateToken`. Il metodo `addSuccessfulBioCheckToStats` cambierà le componenti che riguardano le statistiche del sistema, aggiungendo il successo della validazione, anche nello schema in Z è inclusa questa procedura, tramite l'inclusione dello *AddSuccessfulBioCheckToStats*.

4.6 Scrittura del Token

Dopo i controlli biometrici il sistema dovrà scrivere sul token il certificato di autorizzazione(Authorization Certificate).
Se questa operazione avrà esito positivo, l'utente sarà successivamente ammesso nell'enclave.

$$TISWriteUserToken \mathrel{\widehat=} (WriteUserToken \fatsemi UpdateUserToken) \\ \lor [\, UserTokenTorn \mid status = waitingUpdateToken \,]$$

```
public static Constraint tisWriteUserToken(LList idStation,LList
    realWorld,LList realWorld_,LList idStation_) {

      Solver solver= new Solver();
      LVar status= new LVar();
      new LList();
      LList internal= LList.empty().ins(status,new LVar(),new
          IntLVar());
      new LList();
      solver.add(idStation.eq(LList.empty().ins(new LList(),new
          LList(),new LList(),new LList(),new LList(),new LList(),
          new LList(),new LList(),new LList(),new LList(),new
              LList(),internal,new LVar(),new LList())));
      solver.add((writeUserToken(idStation, realWorld, realWorld_,
          idStation_).and
          (updateUserToken(idStation, realWorld, realWorld_,
              idStation_)).or
          (userTokenTorn(idStation, realWorld, realWorld_,
              idStation_))));
```

```
        solver.add((writeUserToken(idStation, realWorld, realWorld_,
            idStation_)));
        solver.add(status.eq("waitingUpdateToken"));

        return solver.getConstraint();

}
```

L'operazione di scrittura rappresentata dallo schema *TISWriteUserToken* coinvolge tre diverse operazioni: *WriteUserToken* e *UpdateUserToken* legate da una operazione di congiunzione, e *UserTokenTorn*.
Nel relativo metodo Java si ha analogamente le tre funzioni definite nel codice in Z: `writeUserToken`, `updateUserToken` e `userTokenTorn`. In questa operazione la procedura dovrà iniziare da uno stato valorizzato come `waitingUpdateToken`, come specificato anche nel codice in Z.
Tramite `writeUserToken` e `updateUserToken` avviene l'operazione di scrittura dell'Authorization Certificate e aggiornamento del token. Può presentarsi l'eventualità in cui il token venga estratto prematuramente, in quel caso il vincolo restituito da `userTokenTorn` verrà soddisfatto.

4.7 Validazione per l'accesso

La porta verrà sbloccata solo se l'attuale configurazione di TIS consente all'utente di entrare nell'enclave.
Se all'utente non è consentito l'ingresso, verrà richiesto di rimuovere il token.

$$TISValidateEntry \mathrel{\widehat=} EntryOK$$
$$\lor\ EntryNotAllowed$$
$$\lor\ [\, UserTokenTorn \mid status = waitingEntry \,]$$

```
public static Constraint tisValidateEntry(LList idStation,LList
    realWorld,LList realWorld_,LList idStation_) {

      Solver solver= new Solver();
      LVar status= new LVar();
      new LList();
      LList internal= LList.empty().ins(status,new LVar(),new
          IntLVar());
      new LList();
```

```
    solver.add(idStation.eq(LList.empty().ins(new LList(),new
        LList(),new LList(),new LList(),new LList(),new LList(),
        new LList(),new LList(),new LList(),new LList(),new
            LList(),internal,new LVar(),new LList())));
    solver.add((entryOK(idStation, realWorld, realWorld_,
        idStation_)).or
        ((entryNotAllowed(idStation, realWorld, realWorld_,
            idStation_))).or
        ((userTokenTorn(idStation, realWorld, realWorld_,
            idStation_))));

    solver.add(status.eq("waitingEntry"));
    return solver.getConstraint();
}
```

L'operazione di validazione dell'accesso all' enclave rappresentata dallo schema *TISValidateEntry* è formata da tre diverse operazioni legate tra loro: *EntryOK*, *EntryNotAllowed* e *UserTokenTorn*.
Nel metodo Java si ha analogamente le tre funzioni definite nel codice in Z: `entryOK`, `entryNotAllowed` e `userTokenTorn`, legate tramite disgiunzione(or). In questo caso `status` , entità che rappresenta lo stato attuale del sistema, dovrà contenere il valore `waitingEntry`, come specificato anche nel codice in Z.
La funzione `entryOK` rappresenta lo scenario in cui avviene la corretta validazione dell'entrata, `entryNotAllowed` rappresenta lo stato in cui il sistema non autorizza l'utente per l'accesso e infine `userTokenTorn` definisce la circostanza in cui l'utente estrae il token prima che tutti i dati al suo interno vengano letti e validati.

EntryOK

UserEntryContext

$\Xi DoorLatchAlarm$
$\Xi UserToken$
$\Xi Stats$

AddElementsToLog

$status = waitingEntry$
$userTokenPresence = present$

UserAllowedEntry

$currentDisplay' = openDoor$
$status' = waitingRemoveTokenSuccess$
$tokenRemovalTimeout' = currentTime + tokenRemovalDuration$

```
public static Constraint entryOK(LList idStation,LList
    realWorld,LList realWorld_,LList idStation_) {

    LVar userTokenPresence= new LVar();
    LList userToken= LList.empty().ins(new
        LList(),userTokenPresence);
    IntLVar tokenRemovalDuration= new IntLVar();
    LList config= LList.empty().ins(new IntLVar(),new
        IntLVar(),tokenRemovalDuration,new LSet(),new LSet(),new
        LSet(),
        new IntLVar(),new IntLVar());
    LList stats= new LList(),keyStore= new LList();
    IntLVar currentTime= new IntLVar();
    LList doorLatchAlarm= LList.empty().ins(currentTime,new
        LVar(),new LVar(),new LVar(),new LVar(),new LVar());
    LVar status= new LVar();
    LList internal= LList.empty().ins(status,new LVar(),new
        IntLVar());
    LVar screenStats= new LVar();
    LVar screenConfig= new LVar();
    LList currentScreen= LList.empty().ins(screenStats,new
        LVar(),screenConfig);
    LList auditLog= new LList();
    LList userToken_= new LList();
```

```
IntLVar tokenRemovalDuration_= new IntLVar();
LList doorLatchAlarm_= new LList();
LList stats_= new LList();
LList auditLog_= new LList();
LVar status_= new LVar();
LList internal_= LList.empty().ins(status_,new
    LVar(),tokenRemovalDuration_);
LVar screenStats_= new LVar();
LVar screenConfig_= new LVar();
LVar currentDisplay_= new LVar();
LList currentScreen_= LList.empty().ins(screenStats_,new
    LVar(),screenConfig_);
  Solver solver= new Solver();
  new LList();
  solver.add(idStation.eq(LList.empty().ins(userToken,new
      LList(),new LList(),
      doorLatchAlarm,new LList(),new LList(),config,stats,new
          LList(),new LList(),auditLog,internal,
      new LVar(),currentScreen)));

  new LList();
  solver.add(idStation_.eq(LList.empty().ins(userToken_,new
      LList(),new LList(),
      doorLatchAlarm_,new LList(),new LList(),new
          LList(),stats_,new LList(),new
          LList(),auditLog_,internal_,
      new LVar(),currentScreen_)));

  userEntryContext(idStation, realWorld,realWorld_, idStation_);
  solver.add(userToken_.eq(userToken));
  solver.add(doorLatchAlarm_.eq(doorLatchAlarm));
  solver.add(stats_.eq(stats));
  solver.add(screenStats.eq(displayStats(stats)));
  solver.add(screenConfig.eq(displayConfigData(config)));
  solver.add(status.eq("waitingEntry"));
  solver.add(userTokenPresence.eq("present"));
  solver.add(userAllowedEntry(userToken, config, currentTime));
  solver.add(status_.eq("waitingRemoveTokenSuccess"));
  solver.add(currentDisplay_.eq("openDoor"));
  solver.add(tokenRemovalDuration_.eq(
  currentTime.getValue()+tokenRemovalDuration.getValue()));
  solver.add(screenStats_.eq(screenStats));
  solver.add(screenConfig_.eq(screenConfig));
```

```
    return solver.getConstraint();

}
```

Analizzando lo schema *EntryOK* possiamo osservare che i componenti *DoorLatchAlarm, UserToken e Stats* rimangono invariati, quindi il loro stato non cambia. La componente *status* all' inizio dell'operazione assume valore *waitingEntry* mentre allo stato finale cambierà in *waitingRemoveTokenSuccess* perché dopo la validazione sarà chiesto all' utente di rimuovere il proprio token.
Similmente nel codice in Java del metodo `entryrOk`, tramite l'utilizzo di vincoli, viene imposto che `status` sia uguale a `waitingEntry`, e `status_` uguale a `waitingRemoveTokenSuccess`, dato che finita l'autenticazione l'utente dovrà rimuovere il proprio token. Verrà aggiornato anche l'oggetto `display` al fine di comunicare all' utente il successivo sblocco e apertura della porta.

4.8 Sblocco della porta

La porta verrà sbloccata solo dopo che l'utente avrà rimosso il proprio token. Il sistema attenderà a tempo indeterminato la sua rimozione; tuttavia il sistema negherà l'ingresso all' utente che impiega troppo tempo per estrarlo. Quindi, se l'utente attende troppo a lungo per rimuovere il token, il sistema continuerà ad attendere la sua rimozione, ma non consentirà più l'accesso all'enclave.

$$TISUnlockDoor \mathrel{\widehat=} UnlockDoorOK$$
$$\lor [WaitingTokenRemoval \mid status = waitingRemoveTokenSuccess]$$
$$\lor TokenRemovalTimeout$$

```
public static Constraint tisUnlockDoor(LList idStation,LList
    realWorld,LList realWorld_,LList idStation_) {

    Solver solver= new Solver();
    LVar status= new LVar();
    LList internal= LList.empty().ins(status,new LVar(),new
        IntLVar());
    Constraint constraint= (waitingTokenRemoval(idStation,
        realWorld, realWorld_, idStation_)).and
```

```
        (idStation.eq(LList.empty().ins(new LList(),new LList(),new
            LList(),new LList(),new LList(),new LList(),
            new LList(),new LList(),new LList(),new LList(),new
                LList(),internal,new LVar(),new LList()))).and
        (status.eq("waitingRemoveTokenSuccess"));
    solver.add((unlockDoorOK(idStation, realWorld, realWorld_,
        idStation_)).or
        (constraint).or
        ((tokenRemovalTimeout(idStation, realWorld, realWorld_,
            idStation_))));

    return solver.getConstraint();
}
```

L'operazione di validazione dell'impronta digitale rappresentata dallo schema *TISUnlockDoor* è formata da tre diverse operazioni legate tra loro: *UnlockDoorOK*, *WaitingTokenRemoval* e *TokenRemovalTimeout*.
Nel metodo Java si ha analogamente le tre funzioni definite nel codice in Z: `unlockDoorOK, waitingTokenRemoval e tokenRemovalTimeout`, legate tramite disgiunzione(or). All' inizio dell' elaborazione, lo stato del sistema(`status`)avrà valore `waitingRemoveTokenSuccess`, come specificato anche nel codice in Z.
La funzione Java `unlockDoorOK` rappresenta lo scenario in cui la porta dell'enclave viene sbloccata correttamente, `WaitingTokenRemoval` rappresenta lo stato in cui il sistema attende l'estrazione del token dal token reader. Infine `TokenRemovalTimeout` definisce la circostanza in cui l'utente impiega troppo tempo per estrarre il token.

4.9 Accesso Fallito

Se un tentativo di accesso non è riuscito, il sistema attende la rimozione del token prima di iniziare un nuovo processo di autenticazione.

$$TISCompleteFailedAccess \mathrel{\widehat=} FailedAccessTokenRemoved \\ \lor [WaitingTokenRemoval \mid status = waitingRemoveTokenFail]$$

```
public static Constraint tisCompleteFailedAccess(LList
    idStation,LList realWorld,LList realWorld_,LList idStation_) {

    Solver solver= new Solver();
    LVar status= new LVar();
    new LList();
    LList internal= LList.empty().ins(status,new LVar(),new
        IntLVar());
    new LList();
    Constraint constraint= ((waitingTokenRemoval(idStation,
        realWorld, realWorld_, idStation_)).and
          (idStation.eq(LList.empty().ins(new LList(),new
              LList(),new LList(),new LList(),new LList(),new
              LList(),
              new LList(),new LList(),new LList(),new LList(),new
                  LList(),internal,new LVar(),new LList()))).and
          (status.eq("waitingRemoveTokenFail")));

  solver.add((failedAccessTokenRemoved(idStation, realWorld,
      realWorld_, idStation_)).or
  (constraint));

    return solver.getConstraint();
  }
```

L'operazione *TISCompleteFailedAccess* appena descritta mostra e implementa lo stato in cui il sistema non riesce a far accedere l'utente per motivi precedentemente descritti. Le operazioni che compongono *TISCompleteFailedAccess* sono due: *FailedAccessTokenRemoved e WaitingTokenRemoval.*
Nel metodo Java si ha analogamente le due funzioni definite nel codice in Z: `failedAccessTokenRemoved e WaitingTokenRemoval`, legate tramite disgiunzione(or). In questo caso `status` , entità che rappresenta lo stato attuale del sistema, dovrà assumere valore `waitingRemoveTokenFail`, come specificato anche nel codice in Z.
La funzione Java `failedAccessTokenRemoved` rappresenta lo scenario in cui il sistema non permette l'ingresso perché si è presentato un errore durante la validazione. `WaitingTokenRemoval` rappresenta la condizione in cui il sistema aspetta per troppo tempo la rimozione del token e quindi tutta l'operazione di autenticazione avrà esito negativo.

Capitolo 5

Inizializzazione e configurazione

All'inizio del processo, TIS dovrà svolgere delle operazioni di inizializzazione e di configurazione per poter compiere il suo normale funzionamento.

5.1 Proprietà iniziali del sistema

- Key Store vuoto, quindi nessun utente potrà autorizzarsi all' ingresso.
- Dati di configurazione predefiniti, che non consentono l'ingresso a nessuno.
- Componente Door in stato bloccato.
- Componente Audit Log vuoto.
- Tempi interni impostati a 0.

5.2 Inizializzazione dei componenti

Di seguito, alcuni schemi e metodi che compongono l'operazione di inizializzazione e configurazione del sistema.

- **InitDoorLatchAlarm**

  La porta è chiusa all'inizializzazione; questo garantisce che l'allarme non suoni prima che i dati vengano ricevuti e controllati.

InitDoorLatchAlarm
DoorLatchAlarm

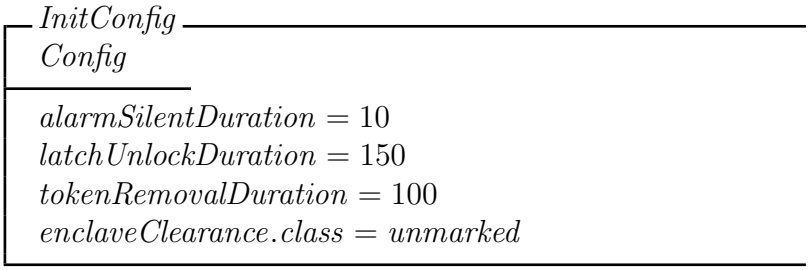
$currentTime = zeroTime$
$currentDoor = closed$
$latchTimeout = zeroTime$
$alarmTimeout = zeroTime$

```
public static Constraint initDoorLatchAlarm(LList
    doorLatchAlarm) {
  IntLVar currentTime= new IntLVar();
  LVar currentDoor= new LVar();
  LVar latchTimeout= new LVar();
  LVar alarmTimeout= new LVar();
  LVar zeroTime = new LVar("zeroTime",0);

    Solver solver= new Solver();
    solver.add(doorLatchAlarm.eq(LList.empty().ins(currentTime,currentDoor,new
        LVar(),new LVar(),latchTimeout,alarmTimeout)));
    solver.add(currentTime.eq(zeroTime));
    solver.add(currentDoor.eq("closed"));
    solver.add(latchTimeout.eq(zeroTime));
    solver.add(alarmTimeout.eq(zeroTime));

    return solver.getConstraint();
```

Attraverso l'operazione di inizializzazione dell'entità *DoorLatchAlarm* vengono settati gli oggetti al suo interno.
Osservando lo schema in Z *InitDoorLatchAlarm* è possibile notare che i componenti che rappresentano un tempo: *currentTime*, *latchTimeout* e *alarmTimeout*, vengono settati a tempo zero e *currentDoor* avrà il suo contenuto impostato a *closed* dato che inizialmente la porta dell'enclave sarà chiusa.
La stessa operazione avviene all' interno del metodo Java `initDoorLatchAlarm` che, attraverso lo sfruttamento di vincoli, inizializza le variabili `currentTime, currentDoor, latchTimeout e alarmtimeout` all' interno della lista `doorLatchAlarm`.

- **InitConfig**

  Inizializzazione della componente *Config* che rappresenta lo stato di configurazione del sistema.




```
public static Constraint initConfig(LList config) {
    IntLVar AlarmSilentDuration= new IntLVar();
    IntLVar LatchUnlockDuration=new IntLVar();
    IntLVar TokenRemovalDuration= new IntLVar();
        LSet EnclaveClearance= new LSet();
        LSet EntryPeriod= new LSet();
        LSet authPeriod= new LSet();
        LVar classs= new LVar();
    Solver solver= new Solver();

    solver.add(config.eq(LList.empty().ins(AlarmSilentDuration,
      LatchUnlockDuration,
      TokenRemovalDuration, EnclaveClearance,authPeriod,
      EntryPeriod,new IntLVar(),new IntLVar())));

    solver.add(EnclaveClearance.eq(LSet.empty().ins(classs)));
    solver.add(AlarmSilentDuration.eq(10));
    solver.add(LatchUnlockDuration.eq(150));
    solver.add(TokenRemovalDuration.eq(100));
    solver.add(classs.eq("unmarked"));

      return solver.getConstraint();

  }
```

Attraverso lo schema *InitConfig* vengono inpostate le variabili *alarmSilentDuration* = 10 *latchUnlockDuration* = 150 e *tokenRemovalDuration*=100 con valori predefiniti, lo stesso procedimento avviene all' interno della funzione `initConfig` che però farà l'assegnazione attraverso l'utilizzo di vincoli di uguaglianza.

- **InitStats** La procedura *InitStats* contiene le statistiche delle operazioni terminate con successo e con fallimento. Inizialmente contiene tutti i parametri settati a zero, dato che all' inizio del sistema nessuna operazione è stata fatta e che quindi il sistema non è stato utilizzato.

InitStats

Stats

$successEntry = 0$
$failEntry = 0$
$successBio = 0$
$failBio = 0$

```
public BoolLVar initStats(LList stats) {
    IntLVar SuccessEntry= new IntLVar(),FailEntry= new
       IntLVar(),
    SuccessBio= new IntLVar(),FailBio= new IntLVar();

    try {
      Solver solver= new Solver();
      solver.add(stats.eq(new
          LList().empty().ins(SuccessEntry,FailEntry,SuccessBio,FailBio)));
      solver.add(SuccessEntry.eq(0));
      solver.add(FailEntry.eq(0));
      solver.add(SuccessBio.eq(0));
      solver.add(FailBio.eq(0));

      solver.solve();
      BoolLVar ris= new BoolLVar(solver.test());
      return ris;
    }catch (Failure e) {
      e.printStackTrace();
      return new BoolLVar(false);
    }
  }
```

- **InitIDStation**

  Inizialmente TIS includerà tutte le configurazioni viste precedentemente

$$
\begin{array}{|l}
\textit{InitIDStation} \\
\hline
\textit{IDStation} \\
\\
\textit{InitDoorLatchAlarm} \\
\textit{InitConfig} \\
\textit{InitKeyStore} \\
\textit{InitStats} \\
\textit{InitAuditLog} \\
\textit{InitAdmin} \\
\hline
\textit{currentScreen.screenMsg} = \textit{clear} \\
\\
\textit{currentDisplay} = \textit{blank} \\
\textit{enclaveStatus} = \textit{notEnrolled} \\
\textit{status} = \textit{quiescent} \\
\hline
\end{array}
$$

```
public BoolLVar initIdStation(LList idStation) {

    LList config= new LList(),stats= new LList(),keyStore= new
        LList();
    IntLVar currentTime= new IntLVar();
    LList doorLatchAlarm= LList.empty().ins(currentTime,new
        LVar(),new LVar(),new LVar(),new LVar(),new LVar());
    LVar status= new LVar();
    LVar enclaveStatus= new LVar();
    LList internal= LList.empty().ins(status,enclaveStatus,new
        IntLVar());
    LVar screenStats= new LVar();
    LVar screenConfig= new LVar();
    LVar screenMsg= new LVar();
    LList currentScreen=
        LList.empty().ins(screenStats,screenMsg,screenConfig);
    LList auditLog= new LList();
    LVar currentDisplay= new LVar();
    LList admin= new LList();
    try {
```

```
    Solver solver= new Solver();
    solver.add(idStation.eq(LList.empty().ins(new
        LList(),new LList(),new LList(),doorLatchAlarm,new
        LList(),newLList(),config,stats,
    keyStore,admin,auditLog,internal,currentDisplay,currentScreen)));
    solver.add(initDoorLatchAlarm(doorLatchAlarm));
    solver.add(initConfig(config));
    solver.add(initKeyStore(keyStore));
    solver.add(initStats(stats));
    solver.add(initAuditLog(auditLog));
    solver.add(initAdmin(admin));
    solver.add(screenMsg.eq("clear"));
    solver.add(currentDisplay.eq("blank"));
    solver.add(enclaveStatus.eq("notEnrolled"));
    solver.add(status.eq("quiescent"));
    solver.solve();
    BoolLVar ris= new BoolLVar(solver.test());
    return ris;
  }catch (Failure e) {
    e.printStackTrace();
    return new BoolLVar(false);
  }
}
```

Lo schema *InitIDStation* include *InitDoorLatchAlarm*, *InitConfig*, *InitKeyStore, InitStats, InitAuditLog e InitAdmin* che rappresentano tutte le configurazioni necessarie per l'inizializzazione dell'oggetto *IDStation*. Analogamente vengono incluse le inizializzazioni nel metodo Java `initIDStation` tramite i vincoli che restituiscono i metodi `initDoorLatchAlarm`, `initConfig`, `initKeyStore`, `initStats`, `initAuditLog e initAdmin`.
All'interno di questa operazione vengono settate alcune variabili sempre facenti parte di `IDStation`, tra cui `display` e `screen`. Lo stato del sistema inizialmente è impostato a `quiescent` e da questo stato un utente può inserire un Token.

5.3 Avvio del sistema ID Station

Partendo dal presupposto che una parte dello stato interno di TIS sia persistente, anche attraverso l'arresto, gli elementi persistenti identificati sono *Config*, *KeyStore* e *AuditLog*, invece tutti gli altri componenti dello stato vengono impostati all'avvio.

StartContext

$\Delta IDStation$
$RealWorldChanges$

$\Xi Config$
$\Xi KeyStore$

$InitDoorLatchAlarm'$
$InitStats'$
$InitAdmin'$

$\Xi UserToken$
$\Xi AdminToken$
$\Xi Finger$
$\Xi Floppy$
$\Xi Keyboard$

```
public static Constraint startContext(LList idStation,LList
    realWorld,LList realWorld_,LList idStation_) {

      LVar userTokenPresence= new LVar();
      LList userToken= LList.empty().ins(new
          LList(),userTokenPresence);
      LList config= new LList(),stats= new LList(),keyStore=
          new LList();
      LList finger= new LList();
      LList adminToken= new LList();
      LList floppy= new LList();
      LList keyBoard= new LList();
      LVar status= new LVar();
      LList internal= LList.empty().ins(status,new LVar(),new
          IntLVar());
      LVar screenStats= new LVar();
      LVar screenConfig= new LVar();
```

```
LList currentScreen= LList.empty().ins(screenStats,new
    LVar(),screenConfig);
LList auditLog= new LList();

LList config_= new LList(),keyStore_= new LList();
LList userToken_= new LList();
LList stats_= new LList();
LList finger_= new LList();
LList adminToken_= new LList();
LList floppy_= new LList();
LList keyBoard_= new LList();
LList auditLog_= new LList();
LVar status_= new LVar();
LVar screenStats_= new LVar();
LVar screenConfig_= new LVar();
LVar currentDisplay_= new LVar();
LList currentScreen_= LList.empty().ins(screenStats_,new
    LVar(),screenConfig_);
IntLVar tokenRemovalTimeout= new IntLVar(5,10);
LVar currentLatch_= new LVar("currentLatch");
LSet availableOps_= new LSet();
LVar doorAlarm_= new LVar();

LList doorLatchAlarm_= LList.empty().ins(new
    IntLVar(),new LVar(),currentLatch_,doorAlarm_,new
    IntLVar(),new IntLVar());

LList admin_= LList.empty().ins(new
    LSet(),availableOps_,new LSet());

LList internal_= LList.empty().ins(status_,new
    LVar(),tokenRemovalTimeout);

  Solver solver= new Solver();

  solver.add(idStation.eq(LList.empty().ins(userToken,adminToken,
  finger,new LList(),floppy,keyBoard,
      config,new LList(),keyStore,new LList(),new
          LList(),
      new LList(),new LVar(),currentScreen)));

  solver.add(idStation_.eq(LList.empty().ins(userToken_,adminToken_,
      finger_,
```

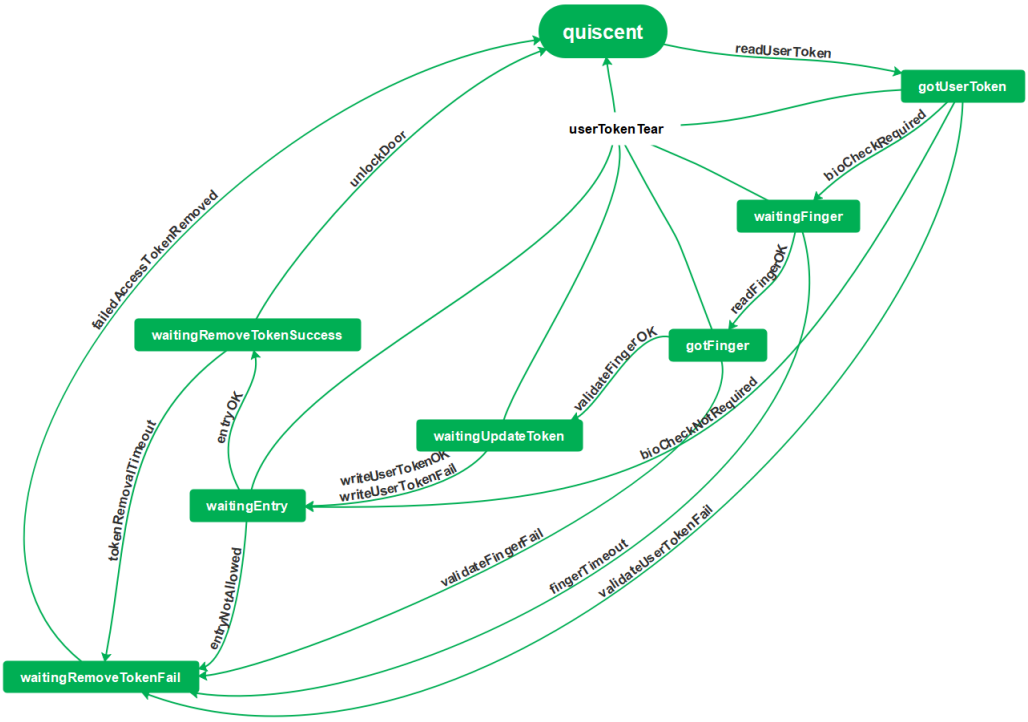
```
        doorLatchAlarm_,floppy_,keyBoard_,
        config_,stats_,keyStore_,admin_,auditLog_,internal_,currentDisplay_,
        currentScreen_)));

        solver.add(realWorldChanges(realWorld, realWorld_));
        solver.add(config_.eq(config));
        solver.add(keyStore_.eq(keyStore));
        solver.add(initDoorLatchAlarm(doorLatchAlarm_));
        solver.add(currentLatch_.eq("locked"));
        solver.add(doorAlarm_.eq("silent"));
        solver.add(initStats(stats_));
        solver.add(initAdmin(admin_));
        solver.add(availableOps_.eq(LSet.empty()));
        solver.add(userToken_.eq(userToken));
        solver.add(adminToken_.eq(adminToken));
        solver.add(finger_.eq(finger));
        solver.add(floppy_.eq(floppy));
        solver.add(keyBoard_.eq(keyBoard));
        solver.add(screenStats_.eq(displayStats(stats)));
        solver.add(screenConfig_.eq(screenConfig));
        solver.add(tokenRemovalTimeout.ge(0));

        return solver.getConstraint();


    }
```

Attraverso lo schema *StartContext* viene inizializzato il contesto in cui avvengono tutte le operazioni che TIS permette di fare. Come si può notare sia dallo schema che nel metodo Java viene settato il componente *currentLatch* a *locked* e l'allarme(*doorAlarm*) della porta a *silent*. Molte componenti rimangono invariate tra cui *usertoken, finger, keyboard* ecc.

5.4 Flusso di esecuzione del prototipo finale

L'implementazione in Java+JSetL è stata fatta al fine di descrive uno scenario che simuli un'operazione di autenticazione e accesso completa, attivata e conclusa dall'utente.
L'attivazione avviene tramite la ricezione di un input dall'esterno, in questo caso dall'inserimento del token, e la conclusione viene effettuata dalla sua estrazione ottenendo come output l'accesso nell'enclave.
L'oggetto finale ottenuto dalla procedura fin'ora descritta è un prototipo funzionale, ovvero un modello parziale del sistema, realizzato con lo scopo di valutarne determinate caratteristiche funzionali.

Per sintetizzare e mostrare nel complesso il flusso di esecuzione del sistema sviluppato è stato utilizzato un diagramma.



Il diagramma di stato sopra mostrato è un tipo specifico di diagramma di flusso, utilizzato per rappresentare gli stati che assume il sistema durante il processo di autenticazione e accesso dell'utente.

Nei riquadri verdi sono presenti gli stati che può assumere il sistema durante l'esecuzione del programma, mentre le frecce rappresentano le transazioni per arrivare ad un determinato stato. Le transizioni sono condizioni che determinano il passaggio del programma allo stato successivo.
In questo specifico caso, gli stati corrisponderanno ai possibili valori che può assumere la variabile status vista nelle precedenti funzioni Java, e le transazioni equivalgono ai metodi Java visti in precedenza.

Il processo inizia e termina nella stessa condizione, infatti sia nel momento dell'inserimento del token che alla fine della sua estrazione il sistema sarà nello stato chiamato *quiescent*.
Si può notare che tutti gli stati all' interno del diagramma posso passare ad uno stato successivo *quiscent*. Infatti *gotUserToken, gotFinger, waitingEntry, waitingFinger e waitingUpdateToken* possono passare allo stato *queiscent* tramite la transazione **userTokenTear** che verifica la condizione in cui l'utente estrae prematuramente il token dal token reader.
Invece *waitingRemoveTokenSuccess e waitingRemoveTokenFail* raggiungeranno lo stato *quiescent* tramite la transazione **unlockDoor** nel caso in cui l'operazione di autenticazione ha avuto esito positivo oppure **failedAccessTokenRemoved** se invece l'utente non è stato validato.

All'interno delle attività del processo di autenticazione si possono presentare diversi problemi per cui l'intera operazione avrà esito negativo, in questo caso il sistema arriverà allo stato *waitingRemoveTokenFail*.
Ci possono essere diverse cause per cui il sistema assume lo stato *waitingRemoveTokenFail*, tra cui errori dovuti all' attesa troppo prolungata per l'inserimento del impronta digitale (**FingerTimeout**) oppure nel caso in cui l'operazione di validazione del token sia correttamente terminata ma l'utente non prosegue con l'estrazione del token(**tokenRemovalTimeout**).
Si possono presentare anche problemi dovuti alla non validazione dei dati nel token; ad esempio dovuti alla mancata corrispondenza tra l'impronta memorizzata nel token e quella inserita dal utente (**validateFingerFail**) oppure i certificati all' interno non sono corretti (**validateUserTokenFail**).

Se invece non si presenteranno difficoltà, dopo le varie attività di controllo e verifica, il sistema arriverà allo stato *waitingRemoveTokenSuccess* concludendo l'operazione di accesso all'enclave con la transazione **unlockDoor**.

Capitolo 6

Conclusioni e lavori futuri

Il lavoro svolto si è focalizzato principalmente sulla codifica della specifica Z del progetto "Tokeneer ID Station" in linguaggio Java, utilizzando la libreria JSetL, ottenendo da quest'ultima un programma, nello specifico un prototipo funzionale.

L'attività è stata svolta in 2 fasi: una fase iniziale basata sull'analisi e comprensione del progetto e della sua specifica, soffermandosi sul suo funzionamento e sulla sua struttura, e una seconda fase che ha riguardato l'implementazione della specifica.

L'utilizzo di vincoli per definire stati e operazioni, permette di rappresentare il comportamento del sistema, focalizzando l'attenzione sul suo funzionamento e definendo scenari funzionali con lo scopo di porre l'attenzione sull'utente e le possibili interazioni con il sistema. Questa tecnica può essere attuata in una fase di analisi e progettazione di un software complesso, permettendo di definire in modo approfondito i requisiti utente. Inoltre, un prototipo funzionale può essere utile a rilevare le fasi più critiche prima dell'implementazione vera e propria. Quindi potrebbe essere molto utile a scopo valutativo per determinare il livello di correttezza del sistema.

Un altro lato interessante che si può notare è che il progetto TIS consiste nello sviluppo di una parte di sistema già esistente (Tokeneer System), questo dimostra il grado di integrazione, non indifferente, che possiede l'intero sistema.
Il lavoro che è stato svolto comprende solamente una parte dell'intero sistema TIS, nello specifico si è analizzato lo scenario di autenticazione dell'utente, questo è stato fatto per focalizzare l'attenzione sulla tecnica per la codifica.
Le funzioni che sono state analizzate e definite in questo elaborato sono sol-

tanto una parte dell'intera implementazione che è stata svolta. Sono state prese in considerazione le operazioni generiche, che includono altre operazioni più specifiche, ma allo stesso tempo è stato spiegato il loro meccanismo. Questa scelta è stata fatta per far comprendere meglio l'implementazione in modo generico e allo stesso tempo complessivo, non focalizzandosi sui particolari.
In conclusione con questo lavoro di tesi è stato possibile dimostrare che partendo da una specifica formale in notazione Z di un sistema complesso si può ottenere un prototipo funzionale scritto in Java+JSetL che conservi la struttura e semantica dei costrutti Z.

6.1 Sviluppi futuri

Tra i possibili sviluppi futuri del progetto ci possono essere delle ulteriori implementazioni finalizzate a rendere il sistema più articolato, ampliando le funzionalità offerte all' utente finale.

- **Gestione del log**: memorizzare tutte le operazioni che si verificano all' interno del sistema, in modo da tener traccia dei processi che TIS svolge e tutte le operazioni che un utente avvia all' interno della struttura.
- **Gestione ruolo dell'utente**: un altro possibile sviluppo è quello di far accedere l'utente con un determinato ruolo, in modo che ad esso siano associati determinate operazione; ad esempio, un utente che accede all' ID Station con ruolo amministrativo potrà modificare configurazioni del sistema e aprire o chiudere la porta dell'enclave. Invece, un utente che accede con un ruolo basico potrà solamente accedere alle postazioni interne all'enclave.
- **Definizione operazione di uscita dell'utente**: implementazione dello scenario in cui l'utente esce dall'enclave.
- **Ottimizzazione comunicazione tra TIS e Token**: migliorare il protocollo di comunicazione tra TIS e il token, aumentando il grado di sicurezza.

Ringraziamenti

Ringrazio il Professor Gianfranco Rossi, relatore di questa tesi di laurea, oltre che per l'aiuto fornitomi durante il periodo di tirocinio, per la disponibilità e precisione dimostratomi durante tutto il periodo di stesura.

Un ringraziamento di cuore alla mia famiglia e a tutti i miei cari, che con il loro sostegno, mi hanno permesso di arrivare fin qui e terminare questo percorso di studio.

Infine un grande ringraziamento alla mia collega e amica Alessandra, con cui ho condiviso l'intero percorso universitario.

Riferimenti bibliografici

[1] Maximiliano Cristiá, Gianfranco Rossi
An Automatically Verified Prototype of the Tokeneer ID Station Specification

[2] Praxis High Integrity Systemsi
Tokeneer ID Station, Formal Specification
14th August 2008

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

[4] JSetL Home Page
http://www.clpset.unipr.it/jsetl/

[5] Gianfranco Rossi, Elio Panegai, Elisabetta Poleo
JSetL: a Java library for supporting declarative programming in Java
Software Practice & Experience 2007; 37:115-149.