Refinamiento a JAVA de casos de prueba abstractos generados por Fastest, un sistema de testing automatizado

Pablo D. Coca
pablodamiancoca@gmail.com

Tesina de grado para la carrera
Licenciatura en Ciencias de la Computación

Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario
Pellegrini 250, Rosario, Santa Fe, Argentina

Director: MSc. Maximiliano Cristiá

Agosto, 2010

Resumen

El testing funcional basado en especificaciones es el proceso de testear en forma estructurada un software a partir de su especificación. Este proceso comúnmente se divide en varias fases, las cuales pueden ser automatizadas casi en su totalidad.

Una fase crítica, es la de ejecutar los casos de prueba abstractos generados en fases anteriores en el proceso de testing. Para esto, antes es necesario refinar los casos de prueba abstractos escritos en el lenguaje de especificación a casos de pruebas concretos escritos en el lenguaje en el que el sistema fue implementado. Una de las técnicas más utilizadas para esta tarea es la de concretización.

En este trabajo se desarrolla TCRL v2.0, un lenguaje que permite reducir la brecha semántica de los casos de prueba abstractos escritos en un lenguaje de especificación (por ejemplo Z) a casos de prueba concretos escritos en algún lenguaje de implementación (por ejemplo JAVA o C).

Además, se desarrolla el parser y el intérprete para TCRL v2.0 y un prototipo del sistema de refinamiento de casos de pruebas abstractos escritos en el lenguaje de especificación Z a casos de pruebas concretos para el lenguaje de programación JAVA.

Dichos desarrollos se piensan integrar a Fastest, un primer prototipo de una herramienta ideada para automatizar el proceso de testing funcional basado en especificaciones antes mencionado.

Agradecimientos

Se agradece:

- a **Maximiliano Cristiá** (director de esta tesis):
  - miembro fundador de **Flowgate Security Consulting**[1] y
  - profesor de las materias "**Análisis de Sistemas**" e "**Ingeniería de Software**" para la carrera "**Licenciatura en Ciencias de la Computación**"[2] de la "**Facultad de Ciencias Exactas Ingeniería y Agrimensura**"[3] de la **Universidad Nacional de Rosario**[4]

  por haber sido una guía muy importante, evacuando cada consulta que surgía durante el desarrollo de esta tesis.

- A **FLOWGATE SECURITY CONSULTING**[5] por haber financiado este proyecto.

- A **Pablo Rodriguez Monetti** y **Diego Ariel Hollmann** por su gran ayuda desinteresada y aportes puntuales que han brindaron útilmente durante el desarrollo del presente trabajo.

- A **Pamela Viale**, **Federico Olmedo**, **Diego Llarrull** y **Pablo Rodriguez Monetti** por su generosidad y compañerismo durante el cursado de toda la carrera.

- Y finalmente, pero no menos importante, al **núcleo familiar** del autor de esta tesis:
  - **Elvira Teresa D'Alessandro**,
  - **Eduardo Luis Coca** y
  - **Noelia Melisa Coca**

  por su incondicional y continuo apoyo no sólo durante el desarrollo de ésta sino a lo largo de toda su vida.

[1] `http://www.flowgate.net/`
[2] `http://www.fceia.unr.edu.ar/lcc/`
[3] `http://www.fceia.unr.edu.ar/`
[4] `http://www.unr.edu.ar/`
[5] `http://www.flowgate.net/`

Índice general

1. Introducción **4**

2. Testing funcional basado en especificaciones Z **6**
2.1. Especificaciones formales 6
2.2. El Lenguaje Z 6
2.3. Especificación sencilla en Z: Agenda Telefónica 7
2.4. Testing funcional 9
2.4.1. Casos de prueba exitosos, VIS, funciones concr y abs 9
2.4.2. Las etapas del testing funcional 11
2.4.3. Ejemplo de Testing 13

3. Fastest **15**
3.1. Utilización de FASTest 16

4. Investigación del Campo de Desarrollo **19**
4.1. Estado del Arte 19
4.2. Enfoques de Concretización 20
4.3. Transformación 21

5. Lenguaje para Refinamiento **23**
5.1. Introducción 23
5.2. TCRL v2.0 - Test Case Refinement Language v2.0 23
5.2.1. BNF 24
5.3. TCRL v2.0 - Gramática - Descripción 25
5.3.1. Ley de Refinamiento 25
5.3.2. Reglas de Refinamiento 27
5.3.3. Regla de Sinónimo 28
5.3.4. Regla de Refinamiento 29
5.3.5. Tipos de Variables 34
5.3.6. Símbolos Terminales 44

6. Implementación del sistema de Refinamiento a JAVA **46**
6.1. Tecnología de la Solución 46
6.1.1. JAVA 46
6.1.2. Eclipse 47
6.1.3. CZT 47
6.1.4. JavaCC 48
6.1.5. JavaCC Plug-in 49

6.2. Diseño . . . 49
6.3. Implementaciones de los módulos del diseño . . . 54
6.3.1. TCRL_Parser . . . 55
6.3.2. TCaseRefClient . . . 58
6.3.3. RefineAST . . . 60
6.3.4. RefineExpr . . . 62
6.3.5. ConstantGenerator . . . 65

7. Casos de Estudio **68**
7.1. Clases de Seguridad . . . 69
7.1.1. Operaciones . . . 69
7.1.2. Casos de Prueba Abstractos . . . 70
7.1.3. Implementación en JAVA . . . 71
7.1.4. Reglas de Refinamiento . . . 72
7.1.5. Leyes de Refinamiento . . . 73
7.1.6. Casos de Prueba Concretos . . . 75
7.2. Entradas de Teatro . . . 77
7.2.1. Operaciones y Estado del sistema . . . 77
7.2.2. Casos de Prueba Abstractos . . . 78
7.2.3. Implementación en JAVA . . . 79
7.2.4. Reglas de Refinamiento . . . 80
7.2.5. Leyes de Refinamiento . . . 82
7.2.6. Casos de Prueba Concretos . . . 83
7.3. Protocolo de Comunicación EOCP para el Microsatélite Científico SACI-1 . . . 85
7.3.1. Operaciones . . . 85
7.3.2. Casos de Prueba Abstractos . . . 86
7.3.3. Implementación en JAVA . . . 87
7.3.4. Reglas de Refinamiento . . . 88
7.3.5. Leyes de Refinamiento . . . 92
7.3.6. Casos de Prueba Concretos . . . 94
7.4. Otros casos de Estudios . . . 95

8. Conclusiones y Trabajo Futuro **97**
8.1. Conclusiones . . . 97
8.2. Trabajo Futuro . . . 98

A. TCRL v2.0 - Gramática en BNF **100**

B. Más implementaciones de módulos de diseño **104**

C. Especificaciones de los Casos de Estudio **112**
C.1. Clases de Seguridad . . . 112
C.2. Protocolo de Comunicación EOCP para el Microsatélite Científico SACI-1 . . . 116

D. Interfaces de los módulos **131**

E. Guía de módulos **137**

Capítulo 1

Introducción

Desde principios de los años setenta, la comunidad de la ingeniería de software era consciente de la necesidad de aplicar técnicas de verificación en el desarrollo de software para asegurar que los productos finales satisfagan sus especificaciones además de los deseos de los clientes interesados en ellos.

Aunque las técnicas de verificación de software estáticas (tales como el análisis de diagramas de diseño y de código fuente) se habían vuelto ampliamente usadas, la técnica de verificación de software predominante por aquellos tiempos era el **testing**.

A fines de la década del '80, ya se había difundido la noción de la utilidad de los métodos formales para especificar y diseñar sistemas de software. Así también, comenzó a destacarse la necesidad de usar métodos formales de especificación en la etapa del testing de software. Esto fue debido a la concienciación de que las especificaciones informales eran útiles pero limitadas, y que los beneficios reales se obtendrían aplicando métodos de especificación formal en todas la etapas de desarrollo del software.

Fueron muchos quienes, a lo largo de estos años, se dedicaron a profundizar y desarrollar el testing de software, en especial, el testing basado en especificaciones formales, viendo la importancia que tiene en la industria de software y sus beneficios.

El **testing de software** es el proceso de evaluar un sistema o componente de un sistema de forma manual o automática para verificar que satisface los requisitos esperados, o para identificar diferencias entre los resultados esperados y los reales (IEEE 1983). Testear un sistema significa ejecutarlo bajo condiciones controladas tales que permitan observar su salida o resultados.

En líneas generales, el proceso de testing de software puede realizarse siguiendo una serie de pasos claramente definidos, siendo la mayoría de ellos automatizables en forma considerable. Esto permite pensar en que es posible el desarrollo de una herramienta que, implementando el marco propuesto en los trabajos de:

- Stocks [1],
- Hörcher y Peleska [2] y,

- Stocks y Carrington [3]

sea capaz de automatizar o semi-automatizar el proceso de testing funcional basado en especificaciones escritas en el lenguaje de especificación formal llamado Z.

Un primer prototipo de una herramienta que sigue estos marcos teóricos mencionados anteriormente, ya ha sido concebido ideológicamente por el MSc. y director de esta tesis **Maximiliano Cristiá** y desarrollado en las tesis de grado de **Pablo Rodríguez Monetti** [10] y **Pablo Albertengo** hasta la etapa en la que los casos de prueba abstractos son generados y al cual se lo denominó **FASTest**.

A partir de este etapa comienza la tesis de grado [11] de **Diego Ariel Hollmann** que desarrolla un lenguaje llamado **TCRL** que permite al usuario de la herramienta (*tester*) describir en forma sencilla cómo ha sido implementada cada variable de la especificación y luego tomar los casos de prueba abstractos generados por **FASTest** y a partir de estos, generar los casos de prueba concretos correspondientes para el lenguaje de implementación C.

En esta tesis de grado se mejorarán algunos aspectos y ampliará el lenguaje TCRL desarrollado por Diego Hollmann, se tomarán los casos de prueba abstractos generados por **Fastest** y a partir de estos se generarán los casos de prueba concretos correspondientes para sistemas implementados en JAVA. De este modo, estos casos de pruebas concretos podrían ser ejecutados, capturadas sus salidas, abstraídos estos resultados y comparados con la especificación formal para determinar si se ha detectado o no alguna falla en sistemas implementados en JAVA.

Este trabajo está motivado por el hecho de que, en la actualidad, no se conoce ningún software (solo algunas ideas) que permitan hacer esto.

Por otra parte, no se pretende que de este trabajo se obtenga una herramienta comercial, sino un prototipo para mostrar los conceptos fundamentales que involucran los temas fundamentales anteriormente citados. Luego, en un futuro, mejorando y completando esta herramienta, podría ser utilizada finalmente en la industria del software.

Finalmente, cabe aclarar que este trabajo ha sido financiado por:

- **FLOWGATE SECURITY CONSULTING**[1]: Empresa que ofrece servicios de desarrollo, consultoría y capacitación en Ingeniería de Software, Seguridad Informática e Infraestructura basada en Software Libre de la ciudad de Rosario, Santa Fe, Argentina.

[1] `http://www.flowgate.net`

Capítulo 2

Testing funcional basado en especificaciones Z

2.1. Especificaciones formales

En ciencias de la computación, **una especificación formal es una descripción matemática de software o hardware que podría ser utilizada para desarrollar una implementación**.

El objetivo de una especificación formal es definir el comportamiento de un sistema en forma precisa y sin ambigüedad.

Los lenguajes formales, o lenguajes de especificación formal, utilizan una semántica y una sintaxis definida formalmente. El factor distintivo entre la especificación y la implementación es el nivel de abstracción.

Con la **especificación**, nos restringimos a definir **qué es lo que hace el sistema**, sin entrar en los detalles de cómo lo hace.

Las especificaciones definen aspectos fundamentales del sistema, mientras que información más detallada es omitida y debería ser una cuestión inherente a la implementación. Las especificaciones formales no tienen que ser ejecutables, ya que se corre el riesgo de escribir especificaciones que no sean concisas y abstractas.

2.2. El Lenguaje Z

Z [6] es un lenguaje de especificación formal. Fue desarrollado por **J. R. Abrial** conjuntamente al Grupo de investigación en Programación de la **Universidad de Oxford** [1]. Z está basado en la teoría de conjuntos, el cálculo de predicados y utiliza un cálculo de esquemas para definir estados y operaciones.

En su forma más compleja Z permite especificar máquinas jerárquicas de estados; en su forma más simple, se usa para especificar máquinas de estados. En Z las máquinas de estados se describen en dos grandes fases:

Definición del conjunto de estados:

[1] `http://www.ox.ac.uk/`

Lo primero que se hace es definir el conjunto de estados de la máquina. Para ello se escribe un esquema de estado el cual contiene las variables de estado de la máquina. Cada estado de la máquina queda definido por una tupla de valores particulares que se asocia a cada variable de estado. Por lo tanto, si una de las variables varía sobre los enteros, entonces la máquina tiene infinitos estados.

Definición de las operaciones o transiciones de estado:

Todas las operaciones de estado juntas definen la relación de transición de estados de la máquina. Existen dos clases de operaciones de estado:

- aquellas que producen una transición de estado y
- aquellas que sólo consultan el estado actual de la máquina.

Cada operación de estado que modifica el estado del sistema describe cómo la máquina transforma uno de sus estados en otro de ellos. Las operaciones de estado se definen dando uno o varios esquemas de operación para cada una de ellas.

Para ejemplificar algunos conceptos y detalles del lenguaje, se presenta una pequeña especificación de una agenda telefónica y una operación para agregar un contacto.

2.3. Especificación sencilla en Z: Agenda Telefónica

Primero se definen dos tipos básicos: *NOMBRE* y *TELEFONO*. Es irrelevante el detalle sobre la estructura interna de estos tipos, eso es una cuestión de la implementación del sistema. Z permite armar conjuntos, secuencias y otras estructuras con los tipos básicos; y además, se puede distinguir entre dos elementos del mismo tipo, saber si está o no en un conjunto, etc.

También se declara el tipo *MESSAGE*, pero en este caso, es *enumerado* y puede asumir dos valores: *ok* y *error*.

$[NOMBRE]$

$[TENEFONO]$

$MESSAGE ::= ok \mid error$

A continuación se tiene una *definición axiomática* que define una *constante* del tipo $\mathbb{Z}$ ($capMax$) y que asume el valor 1000 (que representará la capacidad máxima de almacenamiento de la agenda).

$$\begin{array}{|l} capMax : \mathbb{Z} \\ \hline capMax = 1000 \end{array}$$

Luego se definen tres esquemas. El primero, representa el conjunto de estados del sistema. Este está representado por una función parcial que va del tipo *NOMBRE* al tipo *TELEFONO*.

Los otros dos, *Agregar_Ok* y *Agregar_Error*, representan, cada uno, una operación parcial. La disyunción de estas dos define la operación total *Actualizar* que es la encargada de, dado un nombre *nom?*, y un teléfono *tel?*, agregar este contacto a la agenda.

AgendaTelefonica

$$at : NOMBRE \pfun TELEFONO$$

Agregar_Ok

$$\Delta AgendaTelefonica$$
$$nom? : NOMBRE$$
$$tel? : TELEFONO$$
$$rep! : MESSAGE$$

$$nom? \notin \mathrm{dom}\ at$$
$$\#\,\mathrm{dom}\ at < capMax$$
$$at' = at \cup \{nom? \mapsto tel?\}$$
$$rep! = ok$$

Agregar_Error

$$\Xi AgendaTelefonica$$
$$nom? : NOMBRE$$
$$tel? : TELEFONO$$
$$rep! : MESSAGE$$

$$\#\,\mathrm{dom}\ at \leq capMax \lor nom? \in \mathrm{dom}\ at$$
$$rep! = error$$

$$Agregar \triangleq Agregar\_Ok \lor Agregar\_Error$$

Si:

- *nom?* no pertenece al dominio de *at* (no está agendado) y
- aún hay lugar en la agenda ($\#\,\mathrm{dom}\ at < capMax$)

se agrega el par $nom? \mapsto tel?$ a la función.

Si en cambio:

- está completa ($\#\,\mathrm{dom}\ at = capMax$) ó
- *nom?* pertenece al dominio de *at* (ya está agendado)

devuelve un error y no modifica el estado de *at*.

Por convención, las variables que finalizan con *?* son variables de entrada para la operación y las que finalizan con *!*, son variables de salida. Por lo tanto, *Actualizar* recibe dos valores, uno de tipo *NOMBRE* y otro de tipo *TELEFONO* respectivamente, y retorna uno de tipo *MESSAGE*.

2.4. Testing funcional

Testear un sistema significa ejecutarlo bajo condiciones controladas tales que permitan observar su salida o resultados. El testing de un sistema se estructura en **casos de prueba** o casos de test y los casos de prueba se reúnen en conjuntos de prueba.

Por otro lado, un sistema complejo suele testearse en varias etapas que, por lo general, se ejecutan siguiendo una estrategia *bottom-up*; es decir, se comienza por el testing de unidad (testear por separado cada unidad funcional, individual del sistema), al cual le sigue el testing de componentes y se finaliza con el testing del sistema.

El testing de software es una fase crítica en el ciclo de vida del software y puede ser muy efectivo si se lo realiza rigurosamente. Los métodos formales (por medio de los lenguajes de especificación) ofrecen las bases para realizar testing de forma rigurosa.

Como se mencionó al inicio de este capítulo, las especificaciones definen los aspectos fundamentales del sistema. Es por esto que el tester tiene acceso a la información más relevante acerca de la funcionalidad del producto.

Testear a partir de especificaciones formales ofrece una manera más simple, estructurada y rigurosa para el testing funcional que las técnicas estándar, como puede ser tomar una especificación informal de un sistema y generar, a mano, algunos casos de prueba.

Por otra parte, decimos que **un programa es correcto si verifica su especificación**. Por ende, para poder efectuar un testing significativo es necesario contar con alguna especificación del programa o sistema que se quiere probar. De lo contrario cualquier resultado que arroje el programa ante un caso de prueba no se sabrá si es correcto o no.

2.4.1. Casos de prueba exitosos, VIS, funciones concr y abs

El testing ve a un programa, sistema o parte del mismo como una función que va del producto cartesiano de sus entradas en el producto cartesiano de sus salidas. Es decir:

$$P : ID \rightarrow OD$$

donde *ID* es el *dominio de entrada* (input domain) del programa y OD es el *dominio de salida* (output domain). Normalmente los dominios de entrada y salida son conjuntos de tuplas tipadas cuyas componentes identifican a cada una de las variables de entrada o salida del programa, es decir:

$$ID \triangleq [x_1 : X_1, ..., x_n : X_n]$$
$$OD \triangleq [y_1 : Y_1, ..., y_m : Y_m]$$

De esta manera, un **caso de prueba** es un elemento, x, del dominio de entrada (es decir $x \in ID$) y testear P con x es simplemente calcular $P(x)$.

Un **conjunto de prueba**, por ejemplo T, es un conjunto de casos de prueba definido por extensión y testear P con T es calcular $P(x)$ para cada $x \in T$.

Se dice que un caso de prueba es exitoso si descubre un error en el programa. Pero para saber si es exitoso o no debemos compararlo con la especificación que es una descripción diferente, pero complementaria, del programa. Entonces surge el interrogante de saber cómo expresar los casos de prueba y los resultados del programa en términos de especificación.

De la misma forma que se puede ver a un programa como una función matemática se puede ver a su correspodiente especificación en Z. Es decir, se puede ver a una especificación *Op* como una función parcial de *IS* en *OS*, donde *IS* es el espacio de entrada (input space) y *OS* es el espacio de salida (output space) y se definen de la siguiente forma:

$$IS \triangleq [v_1? : T_1, ..., v_a? : T_a, s_1 : T_{a+1}, ..., s_b : T_{a+b}]$$
$$OS \triangleq [v_1! : U_1, ..., v_c! : U_c, s_1 : T_{a+1}, ..., s_b : T_{a+b}]$$

donde $v_i?$ son las variables de entrada, s_i las variables de estado y $v_i!$ las de salida utilizadas en *Op*.

Notar que si *Op* es la especificación del programa *P* las dimensiones de *ID* e *IS* no tienen por qué ser iguales. Lo mismo vale para *OD* y *OS*. *Op* es una función parcial porque en general no todas las operaciones son totales; es decir, no todas las operaciones Z especifican qué ocurre para todas las combinaciones de los valores de las variables de entrada y estado. En consecuencia no tiene sentido testear un programa con un caso de prueba para el cual la especificación no es útil. Por lo tanto, se define el **espacio válido de entrada (*VIS*)** de la especificación *Op* como el subconjunto de *IS* que satisface la precondición de *Op*, formalmente sería:

$$VIS_{Op} \triangleq [IS \mid \text{pre } Op]$$

lo que nos permite definir a *Op* como una función total:

$$Op : VIS_{Op} \rightarrow OS$$

Para poder formalizar la noción de caso de prueba exitoso haría falta aún una función que transforme elementos del *VIS* en elementos del *ID* y otra que haga lo propio entre *OD* y *OS*. Estas funciones se definen de la siguiente forma:

$$concr_P^{Op} : VIS_{Op} \rightarrow ID_P$$
$$abs_P^{Op} : OD_P \nrightarrow OS_{Op}$$

Los nombres de las funciones se deben a que *concr* refina o concretiza un elemento a nivel de especificación en un elemento a nivel de implementación; y, análogamente, *abs* abstrae un elemento a nivel de implementación en un elemento a nivel de especificación. Por este motivo, se llaman funciones de refinamiento y abstracción, respectivamente.

Con todos estos elementos introductorios se define el concepto de **caso de prueba exitoso** de la siguiente manera, sea:

- *P* un programa tal que $P : ID \rightarrow OD$,
- *Op* la especificación en Z de *P* tal que $Op : VIS \rightarrow OS$,
- $t \in VIS$ y

- $x = concr_P^{Op}(t)$

decimos que x es un caso de prueba exitoso para P sí y sólo sí: $Op(t) \neq abs_P^{Op}(P(x))$.

Conceptualmente, esta definición dice que lo que se esperaba que retornara P al suministrarle x no coincide con lo que su especificación indica.

En el presente trabajo se desarrolla un prototipo de una herramienta para llevar a cabo la etapa *concr* para refinar sistemas implementados en JAVA partiendo de la especificación del sistema en Z.

2.4.2. Las etapas del testing funcional

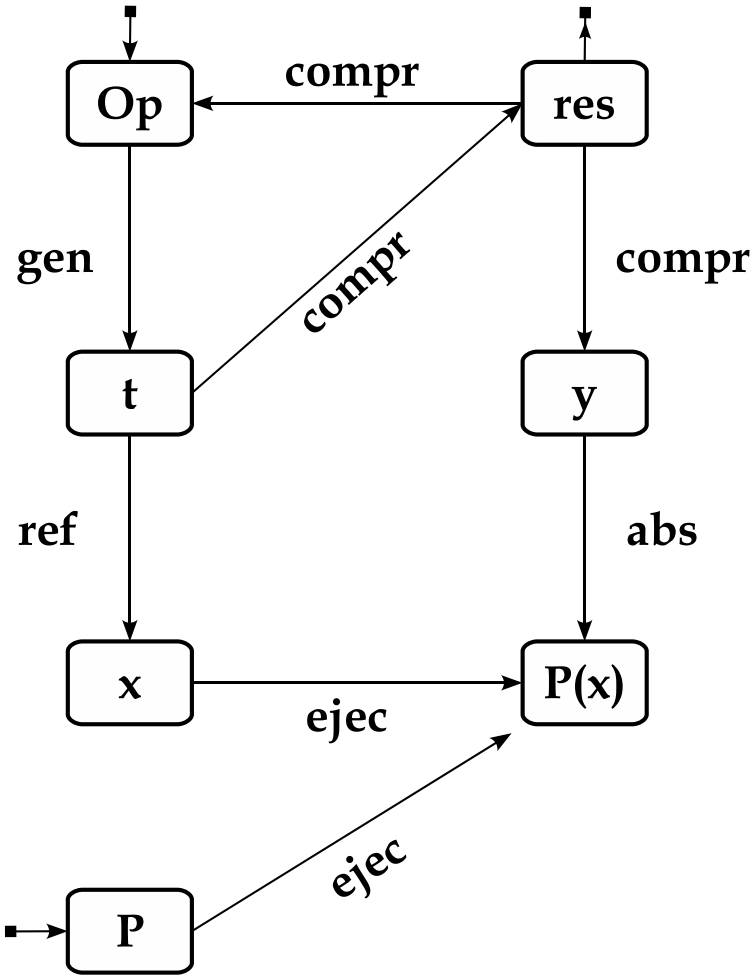

Figura 2.1: El proceso de testing funcional basado en especificaciones formales.

El esquema mostrado en la figura 2.1 muestra las etapas iterativas que hay en la técnica de testing funcional basado en especificaciones formales para poner a prueba un programa P teniendo su especificación Op. Las mismas son:

1. **Generación** de un caso de prueba abstracto a partir de la especificación del sistema. (**gen**)

2. **Refinamiento** o concretización del caso de prueba abstracto para hacerlo ejecutable. (**ref**)

3. **Ejecución** del caso de prueba concretizado en la implementación del sistema. (**ejec**)

4. **Abstración** de la salida. (**abs**)

5. **Comprobación** de los resultados con la especificación. (**compr**)

En el diagrama de la figura 2.1 se puede observar que partiendo de un programa P y de su especificación Op, mediante las etapas antes mencionadas, se puede alcanzar un resultado, *res*, que indica si el proceso en cuestión fue exitoso o no, es decir, si encontró un error en P o no.

La primer etapa, *gen*, es la que genera, a partir de la especificación Op, un caso de prueba a nivel de especificación (llamado *caso de prueba abstracto*), t.

La segunda etapa, *ref*, es la que se encarga de transformar ese caso de prueba abstracto en un caso de prueba a nivel de implementación (llamado *caso de prueba concreto*), x.

La tercer etapa es ejecutar x en P, con lo cual el paso *ejec* representado en la figura por dos flechas, permite obtener la salida $P(x)$ a nivel de implementación, llamada *salida concreta*.

La cuarta etapa, *abs* es la que hace posible abstraer esta salida a nivel de especificación (que como es de esperar, se llama *salida abstracta*), resultando en y.

Por último, la quinta etapa, se realiza a través de *compr* utilizando: la especificación Op, el caso de prueba abstracto t y la salida abstracta y, y se comprueba si y se corresponde con t según Op, obteniendo el resultado *res*. *res* es una respuesta binaria que indica si existe o no tal correspondencia. Si no existe, significa que se ha encontrado un error en el programa P.

La segunda etapa de la figura 2.1 (**ref**) es justamente donde se centra la presente tesis. Osea donde se *concretizan* los casos de prueba abstractos para obtener los casos de prueba concretos. Más adelante se verá que uno de los métodos para concretizar y ejecutar los casos de prueba abstractos, es el que se denomina transformación o refinamiento.

No es un objetivo de esta tesis de grado entrar en los detalles de cómo se generan los casos de prueba abstractos. Simplemente se quiere mencionar que existen varias *Tácticas de Testing* [8], para esta realizar tarea como por ejemplo:

- ***Forma Normal Disyuntiva*** **(FND)**,
- ***Particiones Estándar*** **(PE)**,
- ***Propagación de Subdominios*** **(PSD)** y
- ***Mutación de Especificaciones*** **(ME)**

que dividen el espacio válido de entrada (VIS) en clases de prueba. Cada una de estas clases, expresa una alternativa funcional descripta en la especificación. Como las clases de prueba suelen formar una partición del *VIS* suelen llamarse clases de equivalencia. Estas clases se relacionan entre sí para formar lo que se denomina *árbol de pruebas* [3].

2.4.3. Ejemplo de Testing

A continuación se presenta un ejemplo de aplicación de esta metodología utilizando la operación *Agregar* definida en la sección 2.3.

El $VIS_{Agregar}$ quedaría:

$$VIS_{Agregar} \triangleq [at : NOMBRE \mapsto TELEFONO;\ nom? : NOMBRE;\ tel? : TELEFONO]$$

Aplicando ahora alguna táctica de testing, por ejemplo *Forma Normal Disyuntiva* [8] se obtiene:

$$Agregar_1^{FND} \triangleq [VIS_{Agregar} \mid nom? \in \text{dom } at]$$
$$Agregar_2^{FND} \triangleq [VIS_{Agregar} \mid nom? \notin \text{dom } at \land \#(\text{dom } at) < capTot]$$
$$Agregar_3^{FND} \triangleq [VIS_{Agregar} \mid nom? \notin \text{dom } at \land \#(\text{dom } at) = capTot]$$

En este último caso es conveniente hacer una modificación en la especificación a los fines prácticos de este ejemplo. El hecho es que para generar un caso de prueba en el que $\#(\text{dom } at) = capTot$ habría que definir 1000 elementos dentro de la función *at*. Por lo tanto, para elaborar un caso de prueba para la clase generada, modificaremos $capTot = 3$. Funcionalmente, es equivalente que *capTot* sea 1000 ó 3, debido a que lo que se intenta testear con esta clase de prueba, es el comportamiento de la operación *Agregar* cuando la agenda se encuentra es su capacidad máxima.

Como se mencionó, existen otras tácticas que se podrían seguir aplicando, posiblemente generando nuevas particiones. Luego, por cada una de estas particiones, se genera un caso de prueba abstracto que representa a cada clase.

$Agregar_1^{FND}\_TCase$

$at : NOMBRE \pfun TELEFONO$
$tel? : TELEFONO$
$nom? : NOMBRE$
$rep! : MESSAGE$

$nom? \in \text{dom } at$
$nom? = nom0$
$tel? = tel0$
$at = \{(nom0, tel1)\}$

$Agregar_2^{FND}\_TCase$

$at : NOMBRE \pfun TELEFONO$
$tel? : TELEFONO$
$nom? : NOMBRE$
$rep! : MESSAGE$

$nom? \notin \text{dom } at$
$\#(\text{dom } at) < capMax$
$nom? = nom2$
$tel? = tel2$
$at = \{(nom0, tel0), (nom1, tel1)\}$

$Agregar_3^{FND}\_TCase$

$at : NOMBRE \pfun TELEFONO$
$tel? : TELEFONO$
$nom? : NOMBRE$
$rep! : MESSAGE$

$nom? \notin \dom at$
$\#(\dom at) = capMax$
$nom? = nom3$
$tel? = tel3$
$at = \{(nom0, tel0), (nom1, tel1), (nom2, tel2)\}$

Capítulo 3

Fastest

Fastest fue gestado por la necesidad de contar con una herramienta (inexistente en la actualidad) capaz de automatizar o semi-automatizar el proceso de testing funcional basado en especificaciones Z [6] según la metodología mostrada en la figura 2.1. Es decir, una herramienta que sea capaz, probablemente con alguna intervención de un tester humano, de generar casos de prueba abstractos a partir de la especificación Z de un sistema, refinarlos, ejecutarlos con el código de la implementación, capturar y abstraer las salidas de estas ejecuciones para finalmente comparar los resultados obtenidos con la especificación.

Esta herramienta fue concebida ideológicamente por el **MSc. Maximiliano Cristiá** y un prototipo ya fue desarrollado en la conjunción de las tesis de grado para la carrera "Licenciatura en Ciencias de la Computación"[1] de la Universidad Nacional de Rosario[2] de los graduados:

- **Pablo Rodríguez Monetti** [10]
- **Diego Ariel Hollmann** [11]
- **Pablo Albertengo** [12] (a presentar)

Fastest se desarrolla como parte del proyecto **Flowx**, el cual es co-financiado por la empresa **Flowgate Security Consulting**[3] y por la **Agencia Nacional de Promoción Científica y Tecnológica**[4] a través del **Fondo Tecnológico Argentino (FONTAR)**[5].

Actualmente, está concluido un prototipo de Fastest hasta la fase de la generación de los casos de prueba abstractos y refinamiento de los mismos en casos de prueba concretos para implementaciones hechas en el lenguaje de programación **C**. Justamente, esta etapa de refinamiento, es la que se debe mejorar y ampliar en esta tesis para que se puedan generar casos de prueba concretos también para implementaciones hechas en el lenguaje de programación **JAVA** y permitir agregar de forma flexible más lenguajes de implementación en un futuro.

El sistema Fastest fue desarrollado sobre una arquitectura mixta combinando, entre otros estilos arquitectónicos, los siguientes:

[1] `http://www.fceia.unr.edu.ar/lcc/`
[2] `http://www.unr.edu.ar/`
[3] `http://www.flowgate.net/`
[4] `http://www.agencia.mincyt.org.ar`
[5] `http://www.agencia.gov.ar/spip.php?article38`

- *cliente/servidor* [4] e
- *invocación implícita* [4]

Estos estilos se combinan jerárquicamente, dado que cada cliente de la arquitectura cliente/servidor es organizado de acuerdo a una arquitectura de invocación implícita.

El uso del estilo cliente/servidor fue motivado, principalmente, para aprovechar de manera eficiente los recursos provistos por una red de computadoras, para así obtener mayor performance ejecutando varias tareas en paralelo; y además, poder correr más de un cliente de Fastest en distintas terminales.

Por otra parte, la elección de combinarlo con invocación implícita se da por el hecho de que en lugar de invocar un procedimiento directamente; un componente puede anunciar uno o más eventos y otros componentes del sistema pueden registrar interés en uno o más eventos, asociando a estos distintos procedimientos. Entonces, cuando el evento es anunciado, el propio sistema invoca a todos los procedimientos que anteriormente los componentes registraron interés en tal evento. El beneficio más importante de este aspecto, y el principal motivo que llevó a elegir este estilo para organizar los clientes del sistema es que facilita enormemente la evolución del sistema: los componentes pueden ser reemplazados o pueden agregarse nuevos sin afectar las interfaces de los ya existentes. Justamente, este es el caso del módulo de refinamiento que se realiza en esta tesis, el cual es añadido al sistema sin tener que hacer modificaciones en las interfaces de los demás módulos.

Fastest fue implementado en el lenguaje JAVA. Se utilizó este lenguaje por varias razones, entre las que se pueden destacar la utilización de un lenguaje orientado a objetos y el hecho de que el framework CZT (Community Z Tools [21]), para construir herramientas de métodos formales para el lenguaje Z, usado por Fastest está escrito en código abierto JAVA.

3.1. Utilización de FASTest

Se mostratrá brevemente como se puede utilizar Fastest para la generación de casos de prueba abstractos aplicándolo al ejemplo de especificación en Z de la sección 2.3:

- **Ejecución de *Fastest*.**

```
> java -jar Fastest.jar

Fastest versión 1.0, (C) 2008, Flowgate Security Consulting
Fastest>
```

- **Carga de la especificación** de la Agenda Telefónica de la sección 2.3 (Fastest recibe la especificación Z en formato LaTeX).

```
Fastest> loadspec agenda.tex
Loading specification..
Specification loaded.
Fastest>
```

NOTA: Se supone que la especificación se encuentra en el mismo directorio que dónde se ejecuta Fastest y se llama `agenda.tex`.

- **Visualización de las operaciones cargadas** contenidas en la especificación.

```
Fastest> showloadedops
* Agregar_Ok
* Agregar_Error
* Agregar
Fastest>
```

- **Selección de la operaciones que se quieren testear**, en este caso sólo *Agregar*; que es la operación total (disyunción de las otras dos operaciones parciales).

```
Fastest> selop Agregar
Fastest>
```

- **Generación del árbol de pruebas** para la operación *Agregar*.

```
Fastest> genalltt
Generating test tree for 'Agregar' operation..
Fastest>
```

- **Generación de los casos de prueba abstractos**. Por defecto, Fastest utiliza la *Forma Normal Disyuntiva* como táctica de testing. Justamente, es la que se quiere usar en este ejemplo.

```
Fastest> genalltca
Agregar_DNF_1 test case generation -> SUCCESS.
Agregar_DNF_3 test case generation -> SUCCESS.
Agregar_DNF_2 test case generation -> SUCCESS.
Fastest>
```

La salida en pantalla de Fastest indica que se pudieron generar con éxito tres casos de prueba, uno por cada una de las clases de prueba obtenidas al aplicar la *Forma Normal Disyuntiva*.

- **Presentar en pantalla los esquemas expandidos de los casos de prueba abstractos generados**:

```
Fastest> showsch -tca -u 2

\begin{schema}{Agregar_1^{FND}\_TCase}
   at    : NOMBRE \pfun TELEFONO       \\
   tel?  : TELEFONO                    \\
```

```
   nom?  : NOMBRE                        \\
   rep!  : MESSAGE
\where
   nom?  \in   \dom at                   \\
   nom?  = nom0                          \\
   tel?  = tel0                          \\
   at    = \{(nom0,tel1)\}
\end{schema}

\begin{schema}{Agregar_2^{FND}\_TCase}
   at    : NOMBRE \pfun TELEFONO         \\
   tel?  : TELEFONO                      \\
   nom?  : NOMBRE                        \\
   rep!  : MESSAGE
   \where
   nom?  \notin   \dom at                \\
   \#    (\dom at) < capMax              \\
   nom?  = nom2                          \\
   tel?  = tel2                          \\
   at    = \{(nom0,tel0), (nom1, tel1)\}
\end{schema}

\begin{schema}{Agregar_3^{FND}\_TCase}
   at    : NOMBRE \pfun TELEFONO         \\
   tel?  : TELEFONO                      \\
   nom?  : NOMBRE                        \\
   rep!  : MESSAGE
   \where
   nom?  \notin   \dom at                \\
   \#    (\dom at) = capMax              \\
   nom?  = nom3                          \\
   tel?  = tel3                          \\
   at    = \{(nom0,tel0), (nom1, tel1), (nom2, tel2)\}
\end{schema}

   Fastest>
```

Analizando cada caso de prueba se aprecia que los casos de prueba que obtuvo Fastest, en este sencillo ejemplo, son iguales a los que se han obtenido aplicando manualmente la táctica de *Forma Normal Disyuntiva* a la operación *Agregar* en la sección 2.4.3.

Lo que se mostrará en los próximos capítulos es como, a partir de estos casos de prueba abstractos generados por Fastest, generar los casos de prueba concretos para sistemas escritos en el lenguaje de implementación JAVA.

Capítulo 4

Investigación del Campo de Desarrollo

4.1. Estado del Arte

Desde la década del '80, se han realizado extensas investigaciones en el área del testing basado en especificaciones.

Según la investigación realizada por **Diego Ariel Hollmann** en la sección 4.2 de su tesis de grado "**TCRL, Refinamiento de casos de pruebas para sistema de testing automatizado**" y la realizada por el autor, existen algunas herramientas ya desarrolladas para automatizar el proceso de testing basado en especificaciones formales. **Pero, ninguna de ellas parte de una especificación formal escrita en Z y genera casos de prueba concretos para el lenguaje de programación JAVA**, lo cual permitiría ejecutarlos directamente con la implementación en JAVA del sistema que se quisiera testear.

A continuación se citan los trabajos más relacionados con esta tesina. Algunas de las herramientas crean una *envoltura* alrededor de la implementación para achicar la brecha semántica que existe entre los niveles de abstracción de la especificación y la implementación. Ejemplos de éstas son los frameworks[1] de generación de casos de prueba Torkx [13] de la Universidad de Twente que soporta especificaciones formales en los lenguajes LOTOS, PROMELA y SDL; y TGV (Test Generation with Verification) [14] desarrollado por el IRISA y los laboratorios VERIMAG que soporta especificaciones formales en LOTOS, SDL e IF.

Otras herramientas en cambio, crean, a partir de los casos de prueba abstractos y de la implementación, scripts que *emulan* la ejecución de los casos de prueba por medio del software implementado, como en [15], que genera scripts para casos de prueba que son creados desde una especificación hecha en B.

Mark Utting y Bruno Legeard en [16] hacen una descripción extensa y detallada de una aproximación a una herramienta de testing basada en especificaciones y muestran algunos casos de estudio. También, en el apéndice C de su trabajo, enumeran muchas de las herramientas comerciales de testing y dan una breve descripción de cada una de ellas.

Otro trabajo importante fue el de Machiel van der Bijl, Arend Rensink y Jan Tretmans en

[1] conjunto estandarizado de conceptos, prácticas y criterios para enfocar una problemática particular

"Atomic Action Refinement in Model Based Testing" [17], quienes proponen un framework que agrega un mayor nivel de detalle a la especificación del sistema o a los casos de prueba abstractos para que pueden ser ejecutados. Además, muestran bajo qué circunstancias el refinamiento de un conjunto completo de casos de prueba abstractos obtiene como resultado un conjunto completo de casos de prueba concretos. Para esto muestran un teorema que han denominado *Completitud del refinamiento de casos de prueba*.

Un trabajo más actual es [18] desarrollado por Sebastien Benz quien define **AbstracT**, un lenguaje orientado a aspectos para la instanciación de casos de prueba, modularizando los diferentes detalles del testing en la forma de *aspectos* para permitir el reuso en diferentes contextos de testing. Esta primera aproximación es implementada e integrada en un framework de testing existente.

Finalmente cabe mencionar la herramienta **CADP** (Construction and Analysis of Distributed Processes) que aúna en la misma interfaz provista por la herramienta muchos desarrollos complementarios relacionados con el testing de software para dar como resultado una herramienta que tiene como principal característica poder concretizar una especificación escrita en LOTOS a código del lenguaje de programación C para ser usado con propósitos de simulación, verificación y testing de software.

Como conclusión, luego de investigar y dar con estos desarrollos, se llega a que aún no se ha desarrollado un sistema que implemente el refinamiento como el que se presenta en esta tesis.

4.2. Enfoques de Concretización

Dentro del proceso de testing de software basado en especificaciones formales, la etapa de *refinamiento* o también llamada de *concretización* de los casos de prueba abstractos en casos de prueba concretos es una etapa importante y puede demandar un gran esfuerzo. En algunas aplicaciones de testing basado en especificaciones, el tiempo empleado en el refinamiento puede ser tanto como el tiempo empleado en la modelización del sistema, en otras aplicaciones, puede ser entre un 25 % y un 45 % en relación al tiempo de la modelización [16]. El hecho de hacer testing de software basado en especificaciones formales permite pensar que se podría automatizar o semi-automatizar en gran medida la fase de refinamiento de casos de pruebas abstractos. El problema radica en que los casos de prueba abstractos generados son altamente abstractos, tanto como el nivel de la especificación formal; esto implica que los mismos no contienen suficientes detalles concretos como para ser ejecutados directamente.

Para ejecutar los casos de prueba generados se debe primero inicializar la implementación del sistema a testear, agregar los detalles faltantes en los casos de prueba abstractos y solventar las diferencias existentes entre la especificación y la implementación. Además se debe resolver qué se debe hacer con los valores abstractos del modelo formal y los valores reales que necesita la implementación. Todo esto no es más que reducir la brecha semántica de abstracción existente entre la especificación y la implementación del sistema a ser testeado. Existen varias alternativas para reducir esta brecha [16], las que se pueden clasificar en 3 grupos o enfoques diferentes (ver figura 4.1) :

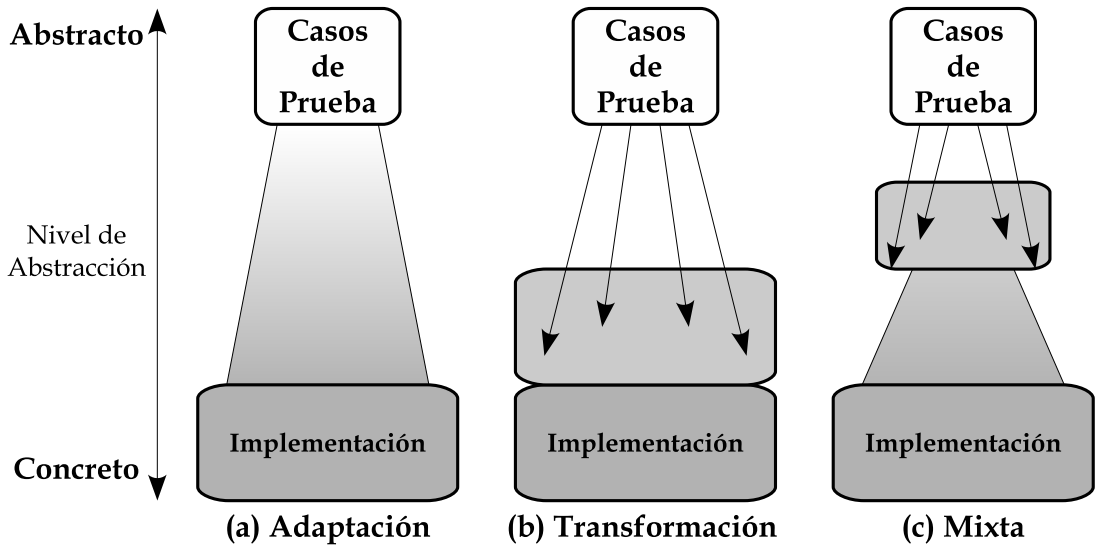

Figura 4.1: Tres enfoques para solventar la brecha semántica entre los casos de prueba abstractos y la implementación del sistema.

- **Adaptación**: consiste en escribir manualmente el código para reducir la brecha semántica. Es necesaria una *envoltura* alrededor de la implementación que provea una interfaz más abstracta para que coincida con el nivel de abstracción de la especificación.
- **Transformación**: consiste en transformar los casos de prueba abstractos en casos de prueba concretos, esto es a nivel del lenguaje de implementación.
- **Mixto**: consiste en una combinación de los otros dos enfoques. A veces, suele ser útil agregar código de adaptación alrededor de la implementación para alcanzar un nivel de abstracción que facilite el testing y luego transformar los casos abstractos en otros más concretos que coincidan con la interfaz de adaptación.

4.3. Transformación

De las diferentes alternativas, mencionadas anteriormente, para la concretización de los casos de prueba abstractos, **la transformación es justamente la que se ha elegido para desarrollar la concretización en esta tesis**.

El enfoque de la transformación requiere convertir cada caso de prueba abstracto en un script ejecutable. Esto se puede realizar por medio de un lenguaje de programación estándar como: C o JAVA, en un lenguaje de sripts como: TCL o JavaScripts o en alguna notación propietaria. Se ha optado por esta última alternativa para la adaptación desarrollándose un lenguaje llamado TCRL v2.o, ver sección 5.

El proceso de transformación puede ser realizado por un programa de traducción que siempre produce una salida en un lenguaje específico, o a través de un motor mas genérico que sea capaz de transformar los casos de prueba abstractos en varios lenguajes de implementación al estar parametrizado por varias interfaces y mapeos para cada uno de estos lenguaje de salida. El

proceso de transformación es realmente complejo, debido a que deben considerarse los siguientes factores:

- El modelo formal de la especificación utiliza constantes y valores abstractos, estos deben ser traducidos a valores concretos significativos para la implementación. Esto suele hacerse por medio de un mapeo entre cada valor de la abstracción y su correspondiente o correspondientes valores concretos, o mediante una expresión concreta que genera el o los valores apropiados.

- Muchas veces es necesario un conjunto de instrucciones de código escritas en el lenguaje de implementación del sistema para una configuración o seteo inicial del entorno del sistema y otro conjunto de instrucciones posterior a la concretización para poder ejecutar el test. Estas instrucciones podrían escribirse en forma manual por el tester y ser insertadas automáticamente por el motor de transformación al inicio y al final de cada test.

  NOTA: Esto en parte se escribe manualmente por el tester y en parte es generado automáticamente por el módulo de refinamiento realizado que más adelante se expondrá en esta tesis.

- La interfaz de cada operación puede ser más complicada debido a que no necesariamente existe un mapeo uno-a-uno entre la **signatura** de la operación abstracta y la operación u operaciones que la implementan. Puede que sea necesario llamar varias operaciones de la implementación para implementar las operaciones abstractas.

- Cuando se testean implementaciones no determinísticas, los test abstractos deberían tener una estructura de árbol en lugar de ser una simple secuencia. Esto requiere que el motor de transformación sea más sofisticado para manejar estas estructuras. Este motor debería generar código fuente ejecutable que incluya sentencias condicionales que chequeen la salida de la implementación y tome la rama adecuada hacia el árbol de la siguiente parte a testear.

  NOTA: Esto no esta soportado hasta la fecha en Fastest.

- Se debe mantener una traza entre el código fuente concreto y el caso de prueba abstracto, y es deseable registrar directamente junto a cada script generado un enlace a la especificación y a los requerimientos informales. Esto puede hacerse insertando estos enlaces a la traza como comentarios en el mismo script o como secuencias ejecutables para que la información de la traza se pueda mostrar cuando el testing falle.

  NOTA: Puede mantenerse un enlace entre el código del script y la especificación mediante la estructuras de datos internas del módulo de refinamiento realizado en esta tesis.

Capítulo 5

Lenguaje para Refinamiento

5.1. Introducción

Como se ha comentado, la implementación de **Fastest** se ha desarrollado en las tesis de grado de **Pablo Rodríguez Monetti** [10] y **Pablo Albertengo** [12] y abarca hasta la generación de los casos de prueba abstractos del sistema que se quiere testear. Posteriormente, **Diego Ariel Hollmann** desarrolla en su tesis de grado "TCRL - Refinamiento de casos de prueba para sistema de testing automatizado"[11] la concretización de estos casos de prueba abstractos para sistemas implementados en el lenguaje de programación C.

La idea central del trabajo de Diego Ariel Hollmann fue desarrollar un lenguaje que denominó **TCRL** (Test Case Refinement Language) que le permite al tester (el usuario de Fastest) describir cómo cada variable de la especificación fue implementada y describir algunos detalles inherentes a la implementación de sistemas escritos en C, como por ejemplo: forma en la que se pide memoria en C, librerías de C incluidas, etc. Al código escrito con este lenguaje llamado TCRL, lo denominó **Ley de refinamiento**.

En esta tesis de grado se corregirán algunas limitaciones que había citado Diego Ariel Hollmann en la sección **6.4. "Limitaciones de TCRL"** de su tesis [11], se agregaran otras mejoras sugeridas por el autor y por el MSc. Maximiliano Cristiá (director de esta tesis), se extenderá y dotará a este lenguaje de mayor flexibilidad para permitir más lenguajes de refinamiento a futuro de una manera más transparente y se desarrollará el módulo de refinamiento para sistemas implementados en JAVA de manera tal de que en un estadío posterior pueda ser integrado a Fastest para la ejecución de los casos de prueba concretos generados y poder realizar la etapa final de **comprobación** de un sistema según el proceso de testing mostrado en la figura 2.1.

5.2. TCRL v2.0 - Test Case Refinement Language v2.0

TCRL v2.0 es una nueva versión del lenguaje ideado por Diego Ariel Hollman en [11] que extiende y mejora algunos aspectos de este lenguaje. TCRL v2.0 es también, así como su predecesor, un **lenguaje declarativo** que permite describir los detalles de la implementación de un sistema a partir de los casos de prueba abstractos de su especificación.

En principio, es diseñado bajo la premisa de soportar un lenguaje de especificación genérico para la descripción de un sistema cuya implementación se realiza en un lenguaje de programación genérico. Dada la premisa de diseño anteriormente mencionada se posibilita que en un futuro se pueda agregar soporte a la implementación del refinamiento para otros tipos de lenguajes de especificación formal fuentes y otros lenguajes de implementación, utilizando también TCRL v2.0. Sin embargo, cabe destacar que toda la implementación de TCRL v2.0 en esta tesis fue realizada teniendo como fuente la descripción de un sistema especificado en el **lenguaje de especificación Z** cuya implementación se realizó en el **lenguaje de programación JAVA**.

Por otra parte, se aclara que la implementación del refinamiento a JAVA realizado es un prototipo y no intenta cubrir, en un principio, todas las posibles implementaciones de un sistema en JAVA (dado que hay infinidad de formas para implementar, por ejemplo, sólamente una función de Z), aunque sí se cubre un amplio espectro dentro de lo que se consideran las formas más usuales de programación en JAVA (utilización de tipos primitivos, arreglos, clases, etc.).

5.2.1. BNF

La BNF (Backus Normal Form) es una meta-sintaxis usada para expresar gramáticas libres de contexto, es decir, una manera formal para describir lenguajes formales. La BNF se utiliza extensamente como notación para las gramáticas, entre otros usos, de:

- lenguajes de programación para computadoras (la mayoría de los libros de textos para la teoría o semántica de lenguajes de programación documentan su lenguaje de programación en BNF),
- sistemas de ingreso de comandos,
- protocolos de comunicación,
- lenguaje natural.

Las reglas de derivación en BNF se realizan por medio de la utilización de símbolos (o tokens) que a su vez utilizan símbolos previamente definidos denominados meta-símbolos. Dichos meta-símbolos son:

- **::=** de definición (la expresión de la izquierda desarrolla a la de la derecha)
- | de alternativa (se puede elegir únicamente uno de los elementos que separa)
- $\{\}^*$ de repetición (las expresiones que incluye se pueden repetir cero o más veces)
- $\{\}^+$ de repetición (las expresiones que incluye se pueden repetir una o más veces)
- [ ] de opción (las expresiones que incluyen pueden utilizarse o no)
- **()** de agrupación (sirven para agrupar las expresiones que incluyen)

5.3. TCRL v2.0 - Gramática - Descripción

A continuación se describirán las partes constitutivas de la gramática de TCRL v2.0 bloque por bloque. La gramática completa puede verse en el apéndice A.

NOTAS:

- Las palabras en MAYÚSCULA son palabras reservadas del lenguaje TCRL v2.0.
- El lenguaje es *case-sensitive*, o sea se distinguen mayúsculas de minúsculas.
- Cada sentencia se separa por medio de un *fin de línea* (eol o Enter).

5.3.1. Ley de Refinamiento

Se denomina *Ley de Refinamiento* a:

$$\begin{aligned}\langle refinementLaw\rangle ::= & \langle lawName\rangle\ eol\\ & \langle preamble\rangle\ eol\\ & \langle rules\rangle\ eol\\ & \langle epilogue\rangle\end{aligned}$$

Los elementos gramaticales referenciados en *refinementLaw* son los siguientes:

- $$\langle lawName\rangle ::= \langle indentifier\rangle$$

  Es el nombre o identificador de la ley de refinamiento. Se debería utilizar para hacer referencia (ver $\langle reference\rangle$) a ella desde otra ley de refinamiento.

  NOTA: Esto será posible cuando se integre la implementación del refinamiento a JAVA en Fastest y pueda soportarse la carga y almacenamiento de varias leyes de refinamiento en el repositorio de Fastest.

  Para ver qué caracteres pueden usarse para el nombre o identificador de la ley de refinamiento, ver sección 5.3.6.

- $$\begin{aligned}\langle preamble\rangle ::= & \textbf{ PREAMBLE}\ eol\\ & \langle PLCode\rangle\\ & \mid \langle reference\rangle\end{aligned}$$

  Código fuente escrito en el lenguaje de la implementación del sistema a ser testeado que necesite ser cargado antes del código de refinamiento generado. Usualmente el código de esta sección refiere a las siguientes cuestiones:

  - importación de librerías necesarias para la implementación del sistema,

- comentarios,
- declaración de variables de la implementación,
- definición de clases y/o paquetes,
- etc.

Se puede ingresar el código fuente por cada ley de refinamiento o hacer referencia al PREAMBLE de otra ley de refinamiento (ver ⟨*reference*⟩).

NOTA: Esto será posible cuando se integre la implementación del refinamiento a JAVA en Fastest y pueda soportarse la carga y almacenamiento de varias leyes de refinamiento en el repositorio de Fastest, además de otros fines dentro de Fastest. Fastest no chequeará la consistencia del código que aquí se escriba. Se asume que el código es correcto.

Para ver qué caracteres pueden usarse en el código fuente, ver ⟨*PLCode*⟩.

- ⟨*epilogue*⟩ ::= **EPILOGUE** *eol*
  ⟨*PLCode*⟩
  | ⟨*reference*⟩

Código fuente escrito en el lenguaje de la implementación del sistema a ser testeado que necesite ser cargado después del código de refinamiento generado. Usualmente el código de esta sección refiere a las siguientes cuestiones:

- liberación de memoria de variables usadas (si corresponde),
- seteo de variables,
- funciones a ejecutar (posteriores al código de refinamiento generado),
- etc.

Se puede ingresar el código fuente por cada ley de refinamiento o hacer referencia al EPILOGUE de otra de otra ley de refinamiento (ver ⟨*reference*⟩).

NOTA: Esto será posible cuando se integre la implementación del refinamiento a JAVA en Fastest y pueda soportarse la carga y almacenamiento de varias leyes de refinamiento en el repositorio de Fastest, además de otros fines dentro de Fastest. Fastest no chequeará la consistencia del código que aquí se escriba. Se asume que el código es correcto.

Para ver qué caracteres pueden usarse en el código fuente, ver ⟨*PLCode*⟩.

- ⟨*PLCode*⟩ ::=⟨*text*⟩

Código fuente escrito en el lenguaje de la implementación del sistema a ser testeado.

NOTA: Fastest no chequeará la consistencia del código que aquí se escriba. Se asume que el código es correcto.

Para ver qué caracteres pueden usarse, ver sección 5.3.6.

- $\langle reference \rangle ::= \langle lawName \rangle \textbf{.} \langle block \rangle$

  Referencia al bloque *block* (ver $\langle block \rangle$) de la ley de refinamiento con nombre o identificador *lawName* (ver $\langle lawName \rangle$).

  NOTA: Esto será posible cuando se integre la implementación del refinamiento a JAVA en Fastest y pueda soportarse la carga y almacenamiento de varias leyes de refinamiento en el repositorio de Fastest, además de otros fines dentro de Fastest. Fastest debería controlar la existencia de las referencias a los bloques antes de la ejecución del módulo RefineAST. Para más información sobre dicho módulo (ver las secciones 6.2 y 6.3.3).

- $\langle block \rangle ::= \textbf{PREAMBLE}$
  $\quad | \textbf{EPILOGUE}$
  $\quad | \textbf{@RULES.} \langle ruleName \rangle \{\textbf{,} \langle ruleName \rangle\}$

  Permite hacer alusión a los bloques: PREAMBLE, EPILOGUE o @RULES de una ley de refinamiento.

  Si se hace alusión al @RULES de una ley de refinamiento se deben indicar cuáles son los nombres de las reglas de refinamiento (ver $\langle rule \rangle$) que se quieren agregar.

  NOTA: Esto será posible cuando se integre la implementación del refinamiento a JAVA en Fastest y pueda soportarse la carga y almacenamiento de varias leyes de refinamiento en el repositorio de Fastest, además de otros fines dentro de Fastest. Fastest debería controlar la existencia de las referencias a los bloques antes de la ejecución del módulo RefineAST. Para más información sobre dicho módulo (ver las secciones 6.2 y 6.3.3).

  Para ver qué caracteres pueden usarse como nombre para una regla de refinamiento, ver $\langle ruleName \rangle$.

- Las reglas de refinamiento $\langle rules \rangle$ se detallarán en profundidad en la sección 5.3.2.

5.3.2. Reglas de Refinamiento

Se denominan *Reglas de Refinamiento* a:

$$\langle rules \rangle ::= \textbf{@RULES}\ eol \quad \langle rule \rangle \{eol\ \langle rule \rangle\}$$

$$| \langle reference \rangle$$

Los elementos gramaticales referenciados en *rules* son los siguientes:

- $$\langle rule \rangle ::= [\langle ruleName \rangle \textbf{:}] \langle RuleSynonym \rangle$$
  $$| [\langle ruleName \rangle \textbf{:}] \langle RuleRefinement \rangle$$

  Es denominada *Regla de Refinamiento*. Es la unidad fundamental para todo el proceso de refinamiento.

  Cada regla de refinamiento puede opcionalmente tener un nombre de regla para usarse como referencia.

  Ver $\langle reference \rangle$ y $\langle ruleName \rangle$ de la sección 5.3.1.

  La reglas de refinamiento pueden ser de 2 tipos:

  - de Sinónimos (ver sección 5.3.3)
  - de Refinamiento propiamente dicha (ver sección 5.3.4)

- $$\langle reference \rangle$$

  Referencia a una regla de refinamiento.

  Ver $\langle reference \rangle$ de la sección 5.3.1.

5.3.3. Regla de Sinónimo

Se denomina *Regla de Sinónimo* a:

$$\langle RuleSynonym \rangle ::= \langle synonymName \rangle == \langle type \rangle$$

donde:

- $$\langle synonymName \rangle ::= \langle identifier \rangle$$

  Permite asociar el nombre *synonymName* al tipo *type* para luego usar *synonymName* en el lugar donde pueda aparecer *type* en cualquier regla de refinamiento, permitiendo escribir de manera sucinta y no repetitiva el script en lenguaje TCRL v2.0.

  Para ver qué caracteres pueden usarse como nombre para una regla de sinónimo, ver $\langle identifier \rangle$ de la sección 5.3.6.

  Para ver la estructura de $\langle type \rangle$, ver 5.3.5.

- $\langle type \rangle$

  Indica un tipo de variable, el cual se utiliza para el proceso de refinamiento. Para conocer que tipos posibles hay ver 5.3.5.

5.3.4. Regla de Refinamiento

Se denomina *Regla de Refinamiento* propiamente dicha a:

$$\begin{aligned} \langle RuleRefinement \rangle ::= & \langle SpecID \rangle \ \{,\langle SpecID \rangle\} \\ & ==> \\ & \langle ImplID \rangle{:}\langle type \rangle \ \{,\langle ImplID \rangle{:}\langle type \rangle\} \\ & [,\langle refinement \rangle] \end{aligned}$$

Sirve para indicar que una o varias variables de la especificación (*SpecID*) se concretizan como una o varias variables de la implementación (*ImlpID*). Cada variable (*ImlpID*) indicada de la implementación se asocia a un tipo de variable (*type*) de las posibles en TCRL v2.0. Para los tipos posibles de variables, ver sección 5.3.5. Los elementos gramaticales referenciados en *RuleRefinement* son los siguientes:

- $\langle SpecID \rangle ::= \langle identifier \rangle$

  Representa al identificador o nombre de una variable en la especificación del sistema.

  NOTA: El sistema de refinamiento a JAVA chequea la existencia de los *SpecID*, de no existir alguno emite un mensaje de error al respecto. Dicho chequeo se hace en el módulo *RefineAST*. Para más información sobre dicho módulo (ver las secciones 6.2 y 6.3.3).

  Para ver qué caracteres pueden usarse para una variable de la especificación del sistema, ver $\langle identifier \rangle$ de la sección 5.3.6.

- $\langle ImplID \rangle ::= \langle identifier \rangle$

  Representa al identificador o nombre de una variable en la implementación del sistema.

  NOTA: El sistema de refinamiento a JAVA chequea la existencia de los *ImplID*, de no existir alguno emite un mensaje de error al respecto. Dicho chequeo se hace en el módulo *RefineAST*. Para más información sobre dicho módulo (ver las secciones 6.2 y 6.3.3).

  Para ver qué caracteres pueden usarse como identificador o nombre de una variable en la implementación del sistema, ver $\langle identifier \rangle$.

- ⟨*refinement*⟩ ::=**@REFINEMENT** *eol*
  { [**@PLCODE** *eol*
  ⟨*PLCode*⟩]
  @SET ⟨*ImplID*⟩ **AS** ⟨*LiveCode*⟩}*

Sirve para indicar opciones complejas para el refinamiento de variables, de ser necesarias. Los elementos gramaticales referenciados en *refinement* son los siguientes:

- Con **@PLCODE** se indica que a continuación se ingresa código fuente escrito en JAVA (⟨*PLCode*⟩). Este código es para ser compilado antes de la etapa de seteo de variables (**@SET**) en la implementación del sistema.

  Debería incluirse todo aquel código que sea necesario para ejecutar el código vivo (ver ⟨*LiveCode*⟩) en tiempo de ejecución del módulo de refinamiento y que no se produzcan errores como:

  - falta de definición de variables que se usan dentro de ⟨*LiveCode*⟩,
  - uso de funciones en ⟨*LiveCode*⟩ que necesitan ser importadas,
  - etc.

  Esta sección es opcional ya que puede no necesitarse ningún código previo.

  NOTA: Fastest no chequeará la consistencia del código que aquí se escriba. Se asume que el código es correcto en caso contrario se producirá el correspondiente error en tiempo de ejecución.
- **@SET** ⟨*ImplID*⟩ **AS** ⟨*LiveCode*⟩
  Sirve para setear la variable o identificador de la implementación ⟨*ImplID*⟩ con el código a compilarse en tiempo de ejecución de ⟨*LiveCode*⟩ el cual debería devolver un valor al evaluarse. Por ejemplo debería devolver el valor de un String de JAVA.

- Se denomina *código vivo* a:

  ⟨*LiveCode*⟩ ::= [⟨*preOp*⟩] ⟨*SpecID*⟩
  |{[**@PLCODE** *eol*
  ⟨*PLCode*⟩]
  @SPECID [⟨*preOp*⟩] ⟨*SpecID*⟩}*

  Sirve para dar instrucciones de cómo realizar el refinamiento para una determinada variable o identificador en la implementación del sistema (*ImplID*). Puede ser de 2 tipos:

  - la primera y más sencilla:

    [⟨*preOp*⟩] ⟨*SpecID*⟩

permite indicar que el refinamiento se hace sobre el valor proveniente en la variable de la especificación *SpecID* aceptando opcionalmente algunos de los siguientes modificadores o pre-operadores:

$$\langle preOp \rangle ::= dom \mid ran \mid first \mid second$$

donde:

- ◇ **dom**: indica que el refinamiento se debe asociar al dominio de la variable o identificador de la especificación *SpecID*, si corresponde aplicar dicho operador a esta variable. Por ejemplo podría aplicarse a una variable que sea una función en la especificación.
- ◇ **ran**: indica que el refinamiento se debe asociar al rango de la variable o identificador de la especificación *SpecID*, si corresponde aplicar dicho operador a esta variable. Por ejemplo podría aplicarse a una variable que sea una función en la especificación.
- ◇ **first**: indica que el refinamiento se debe asociar a la primera componente de la variable o identificador de la especificación *SpecID*, si corresponde aplicar dicho operador a esta variable. Por ejemplo podría aplicarse a una variable que sea un producto cartesiano en la especificación.

  NOTA: Esta funcionalidad todavía no es soportada en la implementación del refinamiento a JAVA realizado en esta tesis.
- ◇ **second**: indica que el refinamiento se debe asociar a la segunda componente de la variable o identificador de la especificación *SpecID*, si corresponde aplicar dicho operador a esta variable. Por ejemplo podría aplicarse a una variable que sea un producto cartesiano en la especificación.

  NOTA: Esta funcionalidad todavía no es soportada en la implementación del refinamiento a JAVA realizada en esta tesis.

EJEMPLO 1:

Supongamos tener el siguiente esquema Z en nuestra especificación del sistema:

Banco

$cajas : NCTA \pfun SALDO$

Supongamos tener las siguientes variables en la implementación de nuestro sistema para representar a *cajas*:

```
public static class Banco
{
//...
static   int[]    ncta  = new int[10];
static   double[] saldo = new double[10];
```

```
//...
}
```

Una posible **regla de refinamiento** para que los valores asignados a la variable de la especificación *cajas* en los casos de prueba abstractos generados por Fastest puedan ser usados en las variables *ncta* y *saldo* de la implementación del sistema, sería:

```
1 regla01: cajas ==> Banco.ncta  : ARRAY[PLTYPE "int"   , 10],
2                    Banco.saldo : ARRAY[PLTYPE "double", 10]
3                    @REFINEMENT
4                    @SET Banco.ncta  AS dom cajas;
5                    @SET Banco.saldo AS ran cajas;
```

Código 5.1: Ejemplo LiveCode N° 1

- la segunda y más compleja:

$$\{[\textbf{@PLCODE } eol \\ \langle PLCode \rangle] \\ \textbf{@SPECID } [\langle preOp \rangle] \ \langle SpecID \rangle\}^*$$

permite indicar que el refinamiento se asocia al valor devuelto por el código JAVA ($\langle PLCode \rangle$) que se ejecuta en tiempo de ejecución y que permite incluir dentro de su cuerpo alusiones a una variable o identificador de la especificación mediante **@SPECID[...]**.
La semántica interna de **@SPECID[...]** es la misma que para el primer tipo más sencillo. Se asume que el valor correspondiente a un *SpecID* siempre se convertiría a la clase String de JAVA (por comodidad) dentro de la implementación del refinamiento.

NOTA: Esta funcionalidad todavía no es soportada en la implementación del refinamiento a JAVA realizada en esta tesis.
Fastest no chequeará la consistencia del código que aquí se escriba. Se asume que el código es correcto en caso contrario se producirá el correspondiente error en tiempo de ejecución.

EJEMPLO 2:

Supongamos tener el siguiente esquema Z en nuestra especificación del sistema:

Date

$dd : DAY$
$mm : MONTH$
$yyyy : YEAR$

Supongamos tener la siguiente variable en la implementación de nuestro sistema:

```
String strFecha;
```

Una posible **regla de refinamiento** para que los valores asignados a las variables de la especificación *dd, mm, yyyy* en los casos de prueba abstractos generados por Fastest puedan ser usados en la variable de la implementación del sistema *strFecha*, sería:

```
1 regla02: dd, mm, yyy ==> strFecha: PLTYPE "String"
2                          @REFINEMENT
3                          @SET strFecha AS @SPECID dd
4                                           @PLCODE
5                                           + "/" +
6                                           @SPECID mm
7                                           @PLCODE
8                                           + "/" +
9                                           @SPECID yyyy
```

Código 5.2: Ejemplo LiveCode N° 2

EJEMPLO 2 - Explicación del refinamiento:

La extensión al sistema de refinamiento desarrollado en esta tesis para integrarse a Fastest debería refinar este ejemplo de la siguiente manera.

Supongamos que se tiene un caso de prueba abstracto generado por Fastest que contiene las siguientes asignaciones:

Date

$dd : DAY$
$mm : MONTH$
$yyyy : YEAR$
$dd = 16$
$mm = 10$
$yyyy = 1982$

Se debería asignar a la variable de la implementación *strFecha* la evaluación de la siguiente expresión en JAVA:

```
"16" + "/" + "10" + "/" + "1982"
```

la cual debería convertirse al **PLTYPE** indicado para *strFecha* en la regla de refinamiento. Como en este caso se ha indicado que *strFecha* se implementa como un String de JAVA no es necesario convertir la expresión anterior debido a que ya es un String de JAVA. Por consiguiente el valor que asumiría la variable de la implementación *strFecha* debería ser:

```
strFecha = "16/10/1982";
```

5.3.5. Tipos de Variables

Cada identificador ⟨*ImplID*⟩ de la implementación del sistema dentro de una regla de refinamiento de TCRL v2.0 debe indicarse que se refina a uno de los siguientes tipos posibles:

⟨*type*⟩ ::=⟨*synonym*⟩
|⟨*plType*⟩
|⟨*enumeration*⟩
|⟨*pointer*⟩
|⟨*array*⟩
|⟨*structure*⟩
|⟨*list*⟩
|⟨*file*⟩
|⟨*db*⟩
|⟨*rfr*⟩

SYNONYM

⟨*synonym*⟩ ::= **SYNONYM** ⟨*synonymName*⟩

Cómo se vio en la sección 5.3.3, una *regla se sinónimo* asocia un nombre de sinónimo ⟨*synonymName*⟩ a un tipo de variable ⟨*type*⟩ dentro de TCRL v2.0.

Ese ⟨*synonymName*⟩ puede utilizarse para indicar, de forma resumida, el mismo ⟨*type*⟩ de la *regla de sinónimo* para una variable o identificador de la implementación del sistema ⟨*ImplID*⟩ dentro de una *regla de refinamiento*.

Para ver qué caracteres pueden usarse como nombre de sinónimo, ver ⟨*indentifier*⟩ de la sección 5.3.6.

PLTYPE

⟨*plType*⟩ ::= **PLTYPE** "⟨*text*⟩"

Indica que se refina al tipo primitivo ⟨*text*⟩ del lenguaje de implementación del sistema.

Si el lenguaje de implementación del sistema es C, podría ser:

- int
- long

- short
- char
- signed char
- unsigned char
- etc.

Si el lenguaje de implementación del sistema es JAVA se soportan todos sus tipos primitivos más la clase String (muy comúnmente usada). Estos son:

- byte
- short
- int
- long
- float
- double
- char
- String (clase java.lang.String)
- boolean

Es el tipo ⟨*type*⟩ más común para indicar cómo es una variable o identificador de la implementación del sistema.

ENUMERATION

⟨*enumeration*⟩ ::= **ENUMERATION**[⟨*plType*⟩**,**{⟨*enumerationElement*⟩ --> ⟨*constant*⟩}*]

Sirve para refinar variables o identificadores de la especificación ⟨*specID*⟩ que son tipos libres enumerados en esa especificación y se refinan como un tipo primitivo del lenguaje de implementación.

Los elementos gramaticales referenciados en *enumeration* son los siguientes:

- ⟨*plType*⟩ : Indica el tipo primitivo del lenguaje de implementación del sistema. Para ver los tipos primitivos posibles, ver 5.3.5.
- ⟨*enumerationElement*⟩ ::= ⟨*identifier*⟩ : Indica uno de los posibles valores que puede asumir el tipo libre enumerado en la especificación.
  Para más detalles sobre los posibles caracteres que pueden utilizarse, ver ⟨*identifier*⟩ de la sección 5.3.6.

- ⟨*constant*⟩ ::= ⟨*text*⟩ : Indica a qué valor constante del tipo primitivo ⟨*plType*⟩ se asocia el valor de la especificación ⟨*enumerationElement*⟩.
  Para más detalles sobre los posibles caracteres que pueden utilizarse, ver ⟨*text*⟩ de la sección 5.3.6.

NOTA: Se recomienda declarar todos los posibles valores que puede asumir la variable o identificador de la especificación ⟨*specID*⟩ con sus respectivas constantes asociadas.

EJEMPLO:

Supongamos tener el siguiente tipo enumerado en nuestra especificación Z del sistema:

$$CTYPE ::= RM \mid SC \mid IDA \mid SDA \mid TD \mid RC \mid MD \mid ML \mid LP$$

y el siguiente esquema de estado:

Status

$$acquiring, waiting, sending, dumping, waitsignal : BIN$$
$$mode : CMODE$$
$$ccmd : CTYPE$$

Supongamos tener la siguiente variable que implementa a la variable de la especificación *ccmd*:

```
int command;
```

La **regla de refinamiento** para que el valor asignado a la variable de especificación *ccmd* en los casos de prueba abstractos generados por Fastest pueda ser usado en la variable de la implementación del sistema *command* sería:

```
1  ccmd    ==> command     : ENUMERATION
2                             [ PLTYPE "int",
3                               RM  -->     "0",
4                               SC  -->     "1",
5                               IDA -->     "2",
6                               SDA -->     "3",
7                               TD  -->     "4",
8                               RC  -->     "5",
9                               MD  -->     "6",
10                              ML  -->     "7",
11                              LP  -->     "8"
12                            ]
```

Código 5.3: Ejemplo ENUMERATION

EJEMPLO - Explicación del refinamiento:

Supongamos que se tiene un caso de prueba abstracto generado por Fastest que contiene la siguiente asignación:

MemoryLoad_SP_2_TCASE

...
ccmd : *CTYPE*
...

---

ccmd = *ML*

El sistema de refinamiento a JAVA desarrollado en esta tesis genera, debido a la regla de refinamiento expuesta anteriormente, la siguiente asignación para la variable de la implementación *command*:

```
command = 7;
```

Esto es debido a que el sistema de refinamiento:

- toma el valor `ML` asignado a la variable de la especificación *ccmd*,
- mediante la indicación `ML -->"7"` de regla de refinamiento se asocia al String de JAVA `"7"` y
- luego se hace el cast correspondiente al tipo primitivo de JAVA *int* indicado en la regla de refinamiento mediante `PLTYPE int "` obteniendo el valor entero `7`.

POINTER

⟨*pointer*⟩ ::= **POINTER**[⟨*type*⟩]

Indica que se refina a un puntero del lenguaje de implementación del sistema que apunta al tipo ⟨*type*⟩. Para más información sobre los posibles tipos de ⟨*type*⟩, ver 5.3.5.

NOTA: Si el sistema se refina al lenguaje de implementación JAVA, no se tiene en cuenta este tipo de refinamiento.

STRUCTURE

⟨*structure*⟩ ::= **STRUCTURE**[⟨*identifier*⟩**,**⟨*element*⟩{**,**⟨*element*⟩}]

Indica que se refina a una estructura del lenguaje de implementación del sistema. Por ejemplo, si se refina a C indicaría un *struc*; si se refina a JAVA indicaría una clase (*class*).
Los elementos gramaticales referenciados en *structure* son los siguientes:

- ⟨*identifier*⟩ : nombre del tipo de la estructura en la implementación del sistema.
- ⟨*element*⟩ ::= ⟨*identifier*⟩**:**⟨*type*⟩ [,"⟨*constant*⟩"] : es una variable miembro de la estructura en la implementación del sistema.

  Los elementos gramaticales referenciados en *element* son los siguientes:

- ⟨*identifier*⟩ : indica el nombre de la variable miembro de la estructura en la implementación del sistema.
- ⟨*type*⟩ : indica el tipo de TCRL v2.0 de la variable al cual se tendría que refinar la variable miembro de la estructura.
  Para más detalles sobre los tipos posibles en TCRL 2.0, ver 5.3.5.

- ⟨*constant*⟩ ::= ⟨*text*⟩ : indica el valor constante que debería asumir el refinamiento sin tener en cuenta el valor provisto por la variable o identificador de la especificación del sistema *specID* asociado en la regla de refinamiento (ver 5.3.4).

  Para los posibles caracteres que pueden utilizarse, ver ⟨*text*⟩ de la sección 5.3.6.

ARRAY

⟨*array*⟩ ::= **ARRAY**[⟨*type*⟩[, ⟨*number*⟩]]

Indica que se refina a un arreglo de tipo ⟨*type*⟩. Opcionalmente se puede indicar con ⟨*number*⟩ la cantidad de sus componentes.

Para más detalles sobre los tipos posibles ⟨*type*⟩ en TCRL 2.0, ver 5.3.5.

LIST

⟨*list*⟩ ::= **LIST**[⟨*identifier*⟩[,⟨*linkType*⟩][,⟨*next*⟩][,⟨*previous*⟩],[⟨*element*⟩{,⟨*element*⟩}][,⟨*memAllocation*⟩]]

Indica que se refina como una lista enlazada.
Los elementos gramaticales referenciados en *list* son los siguientes:

- ⟨*identifier*⟩ : nombre del tipo de la estructura que implementa cada nodo de la lista en el sistema.
- ⟨*linkType*⟩ ::= **SLL** | **DLL** | **CLL** | **DCLL** : tipo de enlace que se utiliza.

  donde:

  - **SLL**: lista simplemente enlazada.
  - **DLL**: lista doblemente enlazada.
  - **CLL**: lista simplemente enlazada circular.
  - **DCLL**: lista doblemente enlazada circular.

- ⟨*next*⟩ ::= ⟨*identifier*⟩ : nombre del miembro de la lista que se utiliza como enlace al siguiente nodo.
  Para los posibles caracteres que pueden utilizarse, ver ⟨*identifier*⟩ de la sección 5.3.6.

- [⟨*previous*⟩ ::= ⟨*identifier*⟩] : nombre del miembro de la lista que se utiliza como enlace al anterior nodo, si corresponde.
  Para los posibles caracteres que pueden utilizarse, ver ⟨*identifier*⟩ de la sección 5.3.6.

- ⟨*element*⟩ : miembros de la lista. Para más detalles, ver 5.3.5.

- ⟨*memAllocation*⟩ ::= ⟨*text*⟩ : función utilizada para alojar memoria. Sino se especifica se utiliza *malloc* como predeterminada. Esta opción es sólo para refinamiento de sistemas en C.
  Para los posibles caracteres que pueden utilizarse, ver ⟨*text*⟩ de la sección 5.3.6.

Para más información, ver [11].

NOTA: Este tipo de refinamiento no es soportado en la implementación del refinamiento a JAVA realizado en esta tesis.

Cómo debería hacerse el refinamiento

Supongamos tener el siguiente esquema de estado en la especificación Z del sistema:

GarbageCollector

$freeMem : \text{seq}\, MEMORY_ADDRESS$

Supongamos tener la siguiente variable que implementa a la variable de la especificación *freeMem*:

```
ArrayList<Double> freeMemory = new ArrayList<Double>();
```

La **regla de refinamiento** para que los valores asignados a la variable de especificación *freeMem* en los casos de prueba abstractos generados por Fastest puedan ser usados en la variable de la implementación del sistema `freeMemory` sería:

```
freeMen ==> freeMemory  : LIST [ ArrayList<Double> ]
```

Código 5.4: Ejemplo LIST

Supongamos que se tiene un caso de prueba abstracto generado por Fastest que contiene la siguiente asignación:

Collect_DNF_1_TCASE

...
$freeMem : \text{seq}\, MEMORY\_ADDRESS$
...

...
$freeMem = \langle addr0, addr1, addr2, addr3 \rangle$
...

La extensión al sistema de refinamiento a JAVA desarrollado en esta tesis debería ser capaz de saber (posiblemente mediante el framework JAVA REFLECTION) cuál es método para agregar elementos de la estructura utilizada para la lista, en este caso `ArrayList.add()`, y generar el código correspondiente que cargue los valores de las componentes de la secuencia asignada a la variable de especificación *freeMem* del caso de prueba abstracto (en este ejemplo: `addr0 , addr1, addr2, addr3`) en la variable de implementación *freeMemory*. Dicho código podría ser para este ejemplo, sólo a modo tentativo y como guía, de la siguiente manera:

```
freeMemory.clear();
freeMemory.add(372156700);
freeMemory.add(372156701);
freeMemory.add(372156702);
freeMemory.add(372156703);
```

en donde los valores que se agregan: `372156700, 372156701, 372156702, 372156703` serían las constantes generadas de tipo `Double` por el sistema de refinamiento a JAVA correspondientes a los valores: `addr0, addr1, addr2, addr3` de la variable de especificación *freeMem*.

FILE

⟨*file*⟩ ::= **FILE**[⟨*name*⟩,⟨*path*⟩,⟨*delimiter*⟩,⟨*text*⟩⟨*fileStructure*⟩
[, **ENDOFLINE**⟨*text*⟩]
[, **ENDOFFILE**⟨*text*⟩]
[,⟨*fieldsRelation*⟩]]

Indica que se refina como una archivo de texto plano.
Los elementos gramaticales referenciados en *file* son los siguientes:

- ⟨*name*⟩ ::= ⟨*text*⟩ : nombre y extensión del archivo de texto plano dentro del sistema.

  Para los posibles caracteres que pueden utilizarse, ver ⟨*text*⟩ de la sección 5.3.6.

- ⟨*path*⟩ ::= ⟨*text*⟩ : ruta del archivo de texto plano dentro del sistema.

  Para los posibles caracteres que pueden utilizarse, ver ⟨*text*⟩ de la sección 5.3.6.

- ⟨*demilimiter*⟩ ::= ⟨*text*⟩ : delimitador entre registros (tabulación, cantidad de espacios en blanco, caracteres especiales, etc.).

  Para los posibles caracteres que pueden utilizarse, ver ⟨*text*⟩ de la sección 5.3.6.

- ⟨*fileStructure*⟩ ::= **LINEAR** | **RPL** | **FPL** : indica cómo se deben ir escribiendo los datos dentro del archivo.

  donde:

  - **LINEAR**: indica que los datos se escriben uno a continuación del otro, separados por el delimitador, todos sin fin de línea.
  - **RPL**: indica que los registros se escriben en líneas individuales separadas con un fin de línea. Cada campo de un registro se separa con el delimitador.
  - **FPL**: indica que los campos de los registros se escriben cada uno en líneas individuales separadas con un fin de línea.

- [, **ENDOFLINE**⟨*text*⟩] : sirve para indicar si el caracter de fin de línea es diferente al estándar.

  Para los posibles caracteres que pueden utilizarse, ver ⟨*text*⟩ de la sección 5.3.6.

- [, **ENDOFFILE**⟨*text*⟩] : sirve para indicar si el caracter de fin de archivo es diferente al estándar.

  Para los posibles caracteres que pueden utilizarse, ver ⟨*text*⟩ de la sección 5.3.6.

- ⟨*fieldsRelations*⟩ ::= ⟨*identifier*⟩**:**⟨*number*⟩{**,**⟨*identifier*⟩**:**⟨*number*⟩} : sirve para indicar la relación que existe entre cada campo escrito en el archivo y cada componente de la variable de la especificación.

Para más información, ver [11].

NOTA: Este tipo de refinamiento no es soportado en la implementación del refinamiento a JAVA realizado en esta tesis.

Cómo debería hacerse el refinamiento

Supongamos tener el siguiente esquema de estado en la especificación Z del sistema:

FilmsDB

$films : \mathbb{P}\, FILMS$

Supongamos que la variable de la especificación *films* se implementa con la siguiente variable que sirve para la escritura de un archivo en el sistema:

```
FileWriter fileFilms;
```

Una **regla de refinamiento** para que los valores asignados a la variable de especificación *films* en los casos de prueba abstractos generados por Fastest puedan ser usados en la variable de implementación `fileFilms`, sería:

```
films   ==> fileFilms   : FILE ["films.db", "home\user\", "\t", RPL]
```

Código 5.5: Ejemplo FILE

Supongamos que se tiene un caso de prueba abstracto generado por Fastest que contiene la siguiente asignación:

Collect_DNF_1_TCASE

$...$
$films : \mathbb{P}\, FILMS$
$...$

$...$
$films = \{film0, film1, film2, film3\}$
$...$

La extensión al sistema de refinamiento a JAVA desarrollado en esta tesis debería ser capaz de saber (posiblemente mediante el framework JAVA REFLECTION) cuál es método para escribir en la variable de la implementación `fileFilms` (en este caso `FileWriter.write()`) y generar el código correspondiente para que los valores de la variable de especificación *films* del caso de prueba abstracto (en este ejemplo: `film0 , film1, film2, film3`) sean escritos en el archivo indicado y de la forma indicada en la regla de refinamiento. Dicho código podría ser para este ejemplo, sólo a modo tentativo y como guía, de la siguiente manera:

```
try{
     fileFilms = new FileWriter("\\home\\usr\\films.db");

     BufferedWriter fileFilmsBuffer = new BufferedWriter(fileFilms);

     fileFilmsBuffer.write("film1");
     fileFilmsBuffer.newLine();       // Write system dependent end of line.
     fileFilmsBuffer.write("film2");
     fileFilmsBuffer.newLine();       // Write system dependent end of line.
     fileFilmsBuffer.write("film3");
     fileFilmsBuffer.newLine();       // Write system dependent end of line.

     fileFilmsBuffer.close();         // Close the output stream
}

catch (Exception e){                  //Catch exception if any
     System.err.println("Error: " + e.getMessage());
}
```

DB

$$\langle db \rangle ::= \mathbf{DB}[\langle dbmsID \rangle,\langle connectionID \rangle,\langle tableName \rangle,\langle columnID \rangle\{,\langle columnID \rangle\}]$$

Indica que se refina como una base de datos.
Los elementos gramaticales referenciados en *db* son los siguientes:

- ⟨*dbmsID*⟩ ::= ⟨*text*⟩ : identificador de la base de datos.

  Para los posibles caracteres que pueden utilizarse, ver ⟨*text*⟩ de la sección 5.3.6.

- ⟨*connectionID*⟩ ::= ⟨*text*⟩ : identificador de la conexión a la base de datos.

  Para los posibles caracteres que pueden utilizarse, ver ⟨*text*⟩ de la sección 5.3.6.

- ⟨*tableName*⟩ ::= ⟨*text*⟩ : nombre de la tabla en la base de datos.

  Para los posibles caracteres que pueden utilizarse, ver ⟨*text*⟩ de la sección 5.3.6.

- ⟨*columnID*⟩ ::= ⟨*identifier*⟩**:**⟨*columnType*⟩ : nombre de una columna con su respectivo tipo de columna ⟨*columnType*⟩ posible, donde:
  ⟨*columnType*⟩ ::= **INT** | **CHAR** | **VARCHAR**

  Se deberían listar todas las columnas de la tabla.

Para más información, ver [11].

NOTA: Este tipo de refinamiento no es soportado en la implementación del refinamiento a JAVA realizado en esta tesis.

Cómo debería hacerse el refinamiento

Supongamos tener el siguiente esquema de estado en la especificación Z del sistema:

FilmsDB

$films : \mathbb{P}\, FILMS$

Supongamos que la variable de la especificación *films* se implementa con la siguiente variable que sirve para conectar a una base de datos llamada `students`:

```
java.sql.Connection filmsDBConnection = java.sql.DriverManager.getConnection
(
      "jdbc:mysql:fceia.unr.edu.ar;" +
      "user     = fceia;"            +
      "password = secret;"           +
      "database = students;"
);
```

Una **regla de refinamiento** para que los valores asignados a la variable de especificación *films* en los casos de prueba abstractos generados por Fastest puedan ser usados en por ejemplo la tabla *films* de la base de datos `students` accedida mediante la variable de implementación `filmsDBConnection`, podría ser:

```
films   ==> filmsDBConnection : DB
                                [ jdbc:mysql,
                                  films,
                                  filmName:CHAR
                                ]
```

Código 5.6: Ejemplo DB

Supongamos que se tiene un caso de prueba abstracto generado por Fastest que contiene la siguiente asignación:

Collect_DNF_1_TCASE

...
$films : \mathbb{P}\, FILMS$
...

...
$films = \{film0, film1, film2, film3\}$
...

La extensión al sistema de refinamiento a JAVA desarrollado en esta tesis debería generar el código correspondiente para que los valores de la variable de especificación *films* del caso de prueba abstracto (en este ejemplo: `film0 , film1, film2, film3`) sean escritos en la tabla *films* de la base de datos indicada en la regla de refinamiento expuesta anteriormente. Dicho código podría ser, sólo a modo tentativo y como guía, de la siguiente manera:

```
java.sql.Statement sqlStatement = filmsDBConnection.createStatement();

try
{
    sqlStatement.executeUpdate( "INSERT INTO films VALUES ( 'films1') ");
    sqlStatement.executeUpdate( "INSERT INTO films VALUES ( 'films2') ");
    sqlStatement.executeUpdate( "INSERT INTO films VALUES ( 'films3') ");

    sqlStatement.close();
}
catch (java.sql.SQLException e)
{
    System.err.println("Error: " + e.getMessage());
}
```

5.3.6. Símbolos Terminales

A continuación de describen los símbolos o *tokens* terminales que se han usado en las reglas de derivación de la BNF de TCRL v2.0:

⟨*identifier*⟩ ::=*letter*{*letter* | *digit* | **?** | **!** | _ | **-**}

⟨*text*⟩ ::=⟨*character*⟩{,⟨*character*⟩}

⟨*number*⟩ ::=*digit*{*digit*}

⟨*digit*⟩ ::=0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

⟨*letter*⟩ ::=*a* | *b* | *c* | *d* | *e* | *f* | *g* | *h* | *i* | *j* | *k* | *l* | *m* | *n* | *o* | *p* | *q* | *r* | *s* | *t* | *u* | *v* | *w* | *x* | *y* | *z* |
A | *B* | *C* | *D* | *E* | *F* | *G* | *H* | *I* | *J* | *K* | *L* | *M* |
N | *O* | *P* | *Q* | *R* | *S* | *T* | *U* | *V* | *W* | *X* | *Y* | *Z*

⟨*character*⟩ ::=⟨*number*⟩ |
⟨*letter*⟩ |
! | # | $ | % | & | ′ | (|) |
∗ | + | , | − | . | / | | |: |; |<|=|>|? |
[|] | _ | ′ | { | }

Capítulo 6

Implementación del sistema de Refinamiento a JAVA

En este capítulo se describirán las herramientas utilizadas, el diseño y la implementación para el sistema de refinamiento de casos de prueba abstractos generados por Fastest en casos de pruebas concretos para sistemas escritos en el lenguaje de programación JAVA.

6.1. Tecnología de la Solución

Se describirá la tecnología (lenguajes, frameworks, herramientas, etc.) utilizada para llegar a la solución del refinamiento de casos de prueba abstractos generados por Fastest en casos de pruebas concretos para sistemas escritos en el lenguaje de programación JAVA.

6.1.1. JAVA

Se optó por desarrollar el sistema de refinamiento a JAVA, además de otras ventajas, con el lenguaje de programación JAVA, dado que Fastest, la aplicación a la cual se piensa integrar en un futuro este sistema de refinamiento, está escrito en JAVA debido principalmente a que muchas de las utilidades relacionadas con el lenguaje de especificación Z (lenguaje de especificación del sistema a testear) están desarrolladas en JAVA.

JAVA [19] es un lenguaje de programación orientado a objetos con el que se puede realizar prácticamente cualquier tipo de programa. Una de las principales características por las que JAVA se ha hecho muy famoso es que es un lenguaje independiente de la plataforma. Esto quiere decir que si se escribe un programa en JAVA podrá funcionar, en principio, en cualquier tipo de ordenador. Ésta es una ventaja significativa para los desarrolladores de software, pues si no se debería hacer un programa para cada combinación de sistema operativo (por ejemplo: Windows, Unix, Linux, BSD, etc) y arquitectura (por ejemplo: x86, RISC, MIPS, PowerPC, etc). Esta independencia se consigue porque se ha creado lo que se denomina *máquina virtual de JAVA* para muchas combinaciones de *sistema operativo/arquitectura* que hacen de puente entre el *sistema operativo/arquitectura* y el programa escrito en JAVA posibilitando que este último se entienda perfectamente independientemente del *sistema operativo/arquitectura* del ordenador.

En la actualidad JAVA es un lenguaje muy extendido y cada vez cobra más importancia tanto en el ámbito de Internet como en la informática en general.

JAVA es desarrollado por la compañía **Sun Microsystems**.

Para más información, visitar: http://www.java.com/es/about/.

6.1.2. Eclipse

Como entorno para el desarrollo del código fuente del sistema de refinamiento a JAVA se ha utilizado Eclipse. **Eclipse** [20] es un entorno de desarrollo integrado (del inglés IDE) multiplataforma de código abierto. Típicamente ha sido usado para desarrollar IDE's, como el Java Development Toolkit (JDT) y el Eclipse Compiler for Java (ECJ) que se entregan como parte de Eclipse (y que son también usados para desarrollar el mismo Eclipse). Eclipse dispone de:

- un editor de texto con resaltado de sintaxis,
- compilación en tiempo real,
- pruebas unitarias con JUnit,
- control de versiones con CVS,
- asistentes (wizards) para la creación de proyectos.

Asimismo, a través de *plugins* es posible añadir diferentes tipos de funcionalidades al entorno.

En este trabajo se ha utilizado el JavaCC Eclipse Plug-in (ver 6.1.5) para el soporte nativo de JavaCC (ver 6.1.4) en Eclipse.
Eclipse es desarrollado por la Fundación Eclipse.

Para más información, visitar: http://www.eclipse.org/.

6.1.3. CZT

Para cargar una especificación Z, Fastest se encarga de parsear y chequear la especificación para transformarla en una estructura de objetos JAVA, a partir de la cual puede extraerse y modificarse más fácilmente la información de sus partes. Para tal fin, Fastest hace uso de las clases provistas por el proyecto CZT.

El proyecto ***Community Z Tools*** **(CZT)** [21] es un framework de código abierto creado con el objetivo de construir un conjunto de herramientas, basadas en métodos formales, para la notación Z y otros dialectos de Z. Estas herramientas incluyen soporte para editar, parsear, hacer chequeo de tipos (*typechecking*) e imprimir especificaciones Z en LaTeX, Unicode o formatos XML. Estas utilidades son desarrolladas bajo el lenguaje de programación JAVA.

A continuación se describirán las facilidades provistas por CZT que son usadas por Fastest.

- **CZT Parser**: permite transformar especificaciones Z escritas en varios formatos (LaTeX, Unicode, XML, etc.) en una representación en forma de *Árbol de sintaxis abstracta* (AST, por Annotated Syntax Tree).

  Un AST permite representar en forma de árbol una especificación Z usando una serie de clases e interfaces JAVA (que conforman el llamado CoreJava de CZT). Esto facilita el acceso desde aplicaciones JAVA a elementos sintácticos del lenguaje Z, como esquemas, predicados y expresiones.

- **CZT Typechecker**: una especificación ya parseada puede ser sometida a un chequeo de tipos usando CZT Typechecker. Las reglas de inferencia de tipo que utiliza esta utilidad están definidas en el estándar ISO de Z.

  CZT Typechecker está implementado como un visitante de una estructura AST. Este visita cada término en el árbol para determinar su tipo (si lo tiene) y detectar errores de tipo. Si no hay errores de tipo, el término es modificado para mantener un registro de su tipo. Si en cambio, hay errores, el término es modificado registrando las posiciones de éstos, y manteniendo una lista de referencias a todos ellos para luego poder fácilmente imprimir mensajes de error.

Para más información, visitar: http://czt.sourceforge.net/.

NOTA:

- Por medio de las clases de CZT, Fastest genera una estructura de objetos AST que representa a una especificación Z. Además luego puede generar los casos de pruebas abstractos para esta especificación Z.

- Para trabajar con estos casos de prueba abstractos generados por Fastest, se necesitó también usar el proyecto CZT para el sistema de refinamiento a JAVA desarrollado en esta tesis para poder inspeccionar estos casos de pruebas abstractos como una estructura de objetos AST y generar los concretos para el lenguaje JAVA.

6.1.4. JavaCC

JavaCC (Java Compiler Compiler) [22] es un generador de analizadores lexicográficos y parsers en lenguaje JAVA para gramáticas presentadas en notación BNF (ver 5.2.1). JavaCC se utilizó para generar el analizador lexicográfico y el parser del lenguaje desarrollado en esta tesis denominado **TCRL v2.o** (ver sección 5).

Los analizadores lexicográficos y parsers son programas que trabajan con entrada de caracteres. Los analizadores lexicográficos pueden dividir una secuencia de caracteres en subsecuencias llamadas *tokens* y clasificarlas. La secuencia de *tokens* es luego pasada al parser para que la analice y determine la estructura de la entrada de caracteres de acuerdo a reglas preestablecidas en la gramática en notación BNF. La salida del parser es un árbol que representa la estructura de la entrada de caracteres. Los analizadores lexicográficos y parsers son también responsables de la generación de mensajes de error si la entrada no supera lexica o sintácticamente las reglas preestablecidas en la gramática en notación BNF.

Para más información, visitar: https://javacc.dev.java.net/.

6.1.5. JavaCC Plug-in

Se agregó al IDE Eclipse (ver sección 6.1.2) el JavaCC Eclipse Plugin para que la edición, generación y ejecución del analizador lexicográfico y parser para TCRL v2.0 (ver sección 5) se haga en el mismo entorno de una forma transparente y sencilla.

JavaCC Eclipse Plugin [23] provee un editor con resaltado sintáctico para JavaCC (ver sección 6.1.4) y un generador del analizador lexicográfico y parser que procesan los archivos `.jj` que contienen las reglas de la gramática en notación BNF para el lenguaje que se desee (en este caso TCRL v2.0) dentro del entorno de Eclipse.

Para más información, visitar: http://sourceforge.net/projects/eclipse-javacc/.

6.2. Diseño

En esta sección se describe el prototipo del sistema de refinamiento a JAVA desarrollado en esta tesis. Primeramente se introducen las siguientes definiciones de conceptos que se van utilizar de aquí en adelante:

- **Diseño**: Descomposición del sistema de software en elementos de software, descripción de la función deseada para cada elemento y las posibles relaciones existentes entre los elementos. [9]
- **Módulo**: Unidad de implementación de software que provee una unidad coherente de funcionalidad. Consta de 2 secciones *interfaz* e *implementación*. [9]
- **Interfaz de un Módulo**: Conjunto de servicios que un módulo exporta para que otros módulos puedan ver, usar o acceder de ese módulo. [9]
- **Implementación de un Módulo**: Forma en la que se logra que la *interfaz* funcione según lo perciben los otros módulos. Cualquier elemento de la *implementación* se dice que es privado. [9]

  NOTA: Se trata de definir interfaces que no permitan descifrar cuestiones de implementación.
- **Estructura de Módulos**: Es una vista del diseño de software que representa la relación binaria entre módulos llamada *ES_SUBMODULO_DE* (ESD). En esta tesis se documenta gráficamente en forma de árbol o mediante 2MIL. [9]
- **Módulos Lógicos**: Son aquellos módulos que no son hojas en la estructura de módulos. [9]
- **Módulos Físicos**: Son aquellos módulos que son hojas en la estructura de módulos. Sólo éstos se implementan. [9]

Para el diseño del sistema de refinamiento a JAVA desarrollado en esta tesis se ha tenido que estudiar la arquitectura de Fastest [10] en profundidad para que una vez desarrollado el sistema de refinamiento a JAVA, éste pueda ser integrado en un futuro con facilidad dentro de

Fastest.

El tipo de diseño elegido es el **diseño basado en ocultación de información** (DBOI), utilizando la metodología de **Parnas** (para más información consultar [9]) y aprovechando las ventajas que ofrece el diseño orientado a objetos como: la herencia, la abstracción, el polimorfismo, etc.

La estructura de módulos del sistema de refinamiento a JAVA nace a partir del módulo de Fastest [10] llamado **TCaseRefinement**. TCaseRefinement es un módulo lógico que representa a uno de los posibles sistemas de refinamiento dentro de Fastest. TCaseRefinement se ha extendido para que incluya a dos módulos lógicos más (ver figura 6.1). Éstos son:

- **DataStructures** y
- **JAVAFunctionalModules**

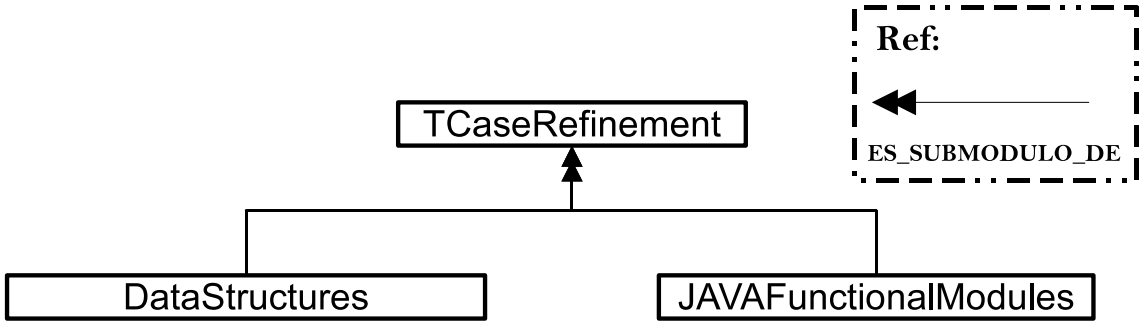

Figura 6.1: Estructura de Módulos - Parte 1

donde:

- **DataStructures**: Es un módulo lógico que agrupa a todos los módulos que representan las estructuras de datos que se utilizan durante el proceso de refinamiento.

  Para más información, consultar:

  - el apéndice D que describe las **interfaces de los módulos** y
  - el apéndice E que describe la **guía de módulos**.

- **JAVAFunctionalModules**: Es un módulo lógico que agrupa a todos los módulos que representan partes funcionales del sistema de refinamiento a JAVA.

  Para más información, consultar:

  - el apéndice D que describe las **interfaces de los módulos** y
  - el apéndice E que describe la **guía de módulos**.

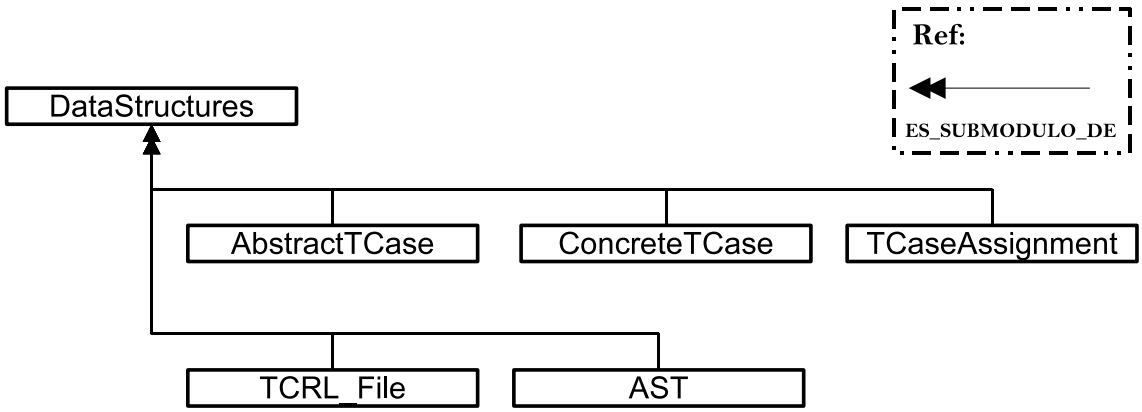

Figura 6.2: Estructura de Módulos - Parte 2

DataStructures a su vez incluye los siguientes módulos (ver figura 6.2):

A continuación se describe que representa cada uno de los módulos incluidos en DataStructure:

- **AbstractTCase**: Es un módulo físico que representa a un caso de prueba abstracto generado por Fastest.
- **ConcreteTCase**: Es un módulo físico que representa a un caso de prueba concreto generado por el sistema de refinamiento.
- **TCaseAssignment**: Es un módulo físico que representa la asignación del código escrito en el lenguaje de programación para el refinamiento (JAVA para esta tesis) para una variable de la implementación del sistema a refinar. (Ejemplo: `maxlevel = 1;`).
- **TCRL_File**: Es un módulo físico que representa al archivo que contiene la ley de refinamiento para el sistema a refinar escrita en el lenguaje TCRL v2.0 (desarrollado en esta tesis, ver sección 5).
- **AST**: Es un módulo lógico que agrupa a los módulos relacionados con el árbol de sintaxis abstracta que representa a una ley de refinamiento escrita en el lenguaje TCRL v2.0 (desarrollado en esta tesis, ver sección 5).

Para más información, consultar:

- el apéndice D que describe las **interfaces de los módulos** y
- el apéndice E que describe la **guía de módulos**.

AST incluye los siguientes módulos (ver figura 6.3):

A continuación se describe que representa cada uno de los módulos incluidos en AST:

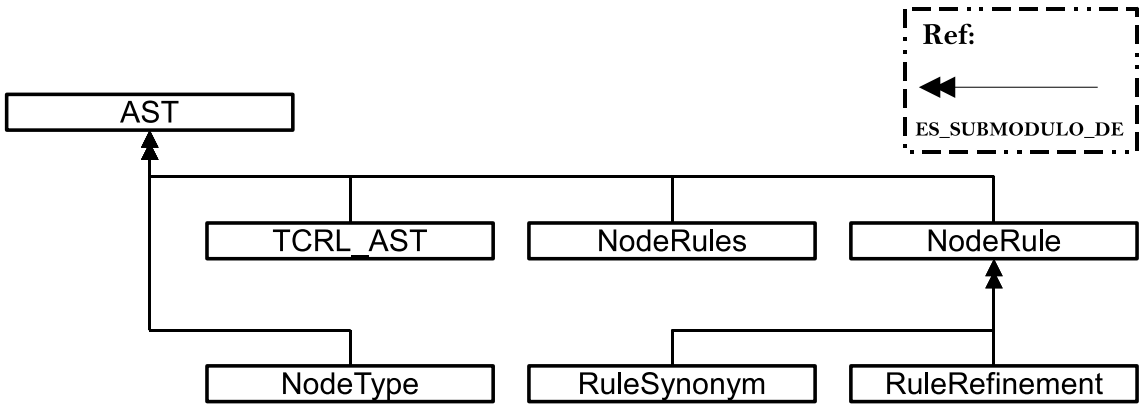

Figura 6.3: Estructura de Módulos - Parte 3

- **TCRL_AST**: Es un módulo físico que representa a la raíz del árbol de sintaxis abstracta de una ley de refinamiento escrita en el lenguaje TCRL v2.o. (desarrollado en esta tesis, ver 5).
- **NodeRules**: Es un módulo físico que representa a todas las reglas de refinamiento que hay dentro de una ley de refinamiento escrita en el lenguaje TCRL v2.o. (desarrollado en esta tesis, ver 5).
- **NodeRule**: Es un módulo físico que representa a una regla de refinamiento (ya sea de sinónimo o de refinamiento propiamente) del lenguaje TCRL v2.o (desarrollado en esta tesis, ver 5). De él heredan los módulos RuleSynonym y RuleRefinement.
- **RuleSynonym**: Es un módulo físico que representa a una regla de sinónimo de una ley de refinamiento escrita en el lenguaje TCRL v2.o (desarrollado en esta tesis, ver 5).
- **RuleRefinement**: Es un módulo físico que representa a una regla de refinamiento de una ley de refinamiento escrita en el lenguaje TCRL v2.o (desarrollado en esta tesis, ver 5).
- **NodeTypes**: Es un módulo lógico que agrupa a todos los tipos de variables a los que se puede refinar una variable de la implementación del sistema a refinar soportados por TCRL v2.0 (desarrollado en esta tesis, ver 5).

Para más información, consultar:

- el apéndice D que describe las **interfaces de los módulos** y
- el apéndice E que describe la **guía de módulos**.

NodeTypes incluye los siguientes módulos (ver figura 6.4):

Para más información, consultar:

- el apéndice D que describe las **interfaces de los módulos** y

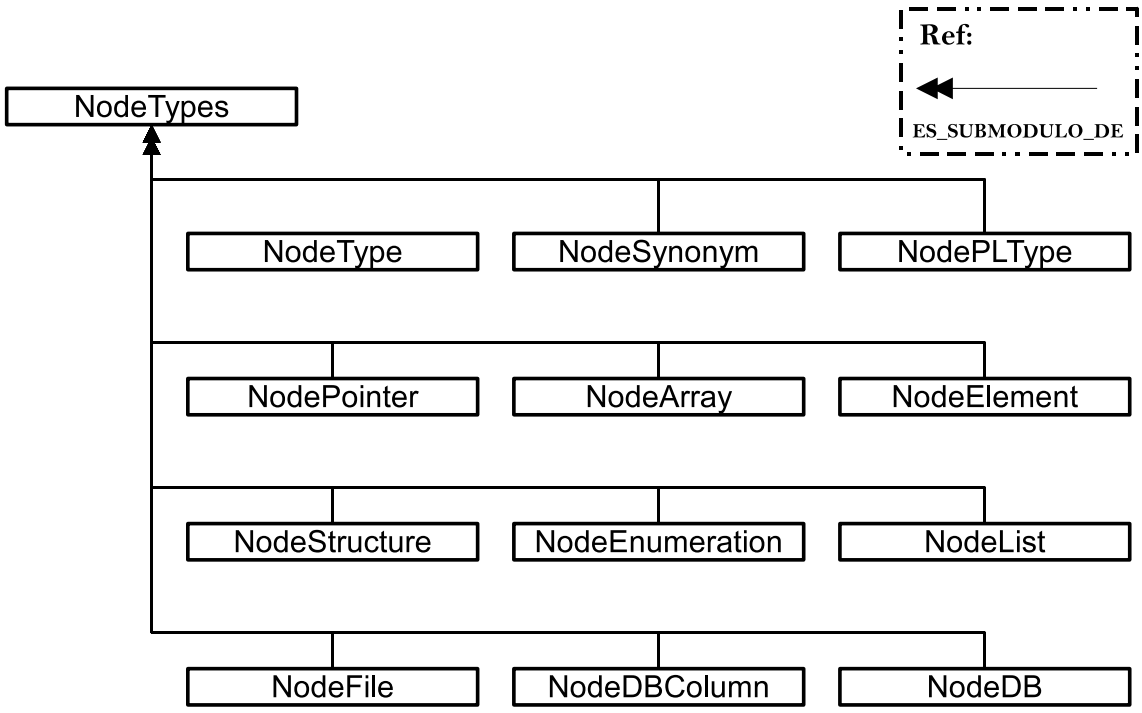

Figura 6.4: Estructura de Módulos - Parte 4

- el apéndice E que describe la **guía de módulos**.

Dentro del módulo lógico **JAVAFunctionalModules** se agrupan los siguientes módulos (ver figura 6.5):

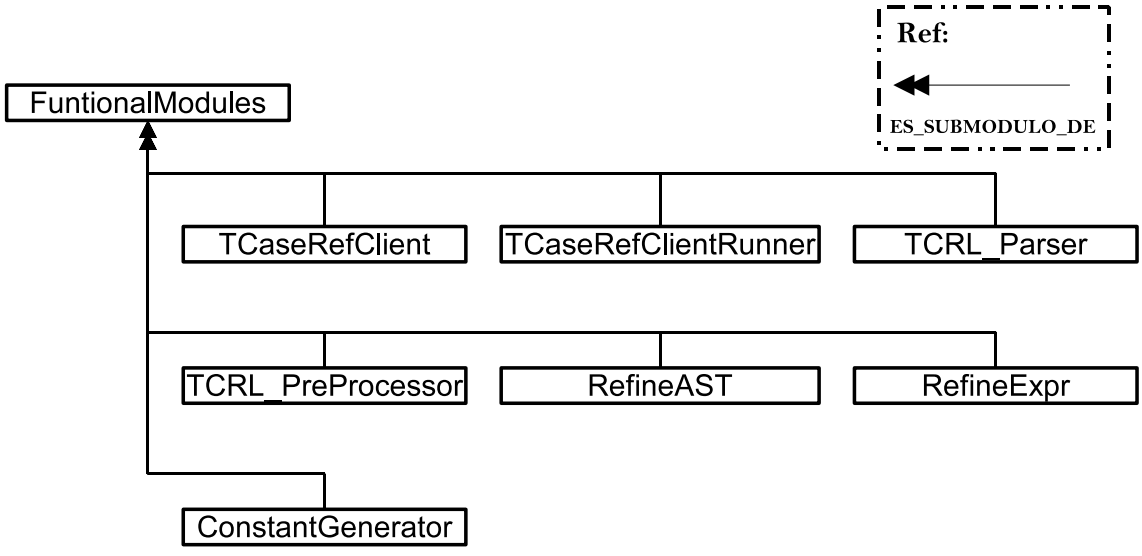

Figura 6.5: Estructura de Módulos - Parte 5

A continuación se describe que representa cada uno de los módulos incluidos en JAVAFunctionalModules:

- **TCaseRefClient**: Es un módulo físico que se encarga de manejar las solicitudes de refinamiento de casos de pruebas abstractos generados por Fastest. Crea una instancia del módulo TCaseRefClientRunner por cada solicitud.
- **TCaseRefClientRunner**: Es un módulo físico que se encarga de iniciar el proceso refinamiento de un caso de prueba abstracto llamando a RefineAST. Cada instancia de TCaseRefClientRunner puede ser ejecutada en clientes distintos, permitiendo aprovechar todos los recursos de la red de computadoras disponibles para el refinamiento.
- **TCRL_Parser**: Es un módulo físico que implementa el analizador lexicográfico y el parser del lenguaje TCRL v2.0 (desarrollado en esta tesis, ver 5).
- **TCRL_PreProcessor**: Es un módulo físico que se encarga del reemplazo de sinónimos (ver sección A) especificados en las posibles reglas de sinónimo contenidas en una ley de refinamiento escrita en el lenguaje TCRL v2.0 (desarrollado en esta tesis, ver 5).
- **RefineAST**: Es un módulo físico que se encarga de la implementación del algoritmo de refinamiento de cada caso de prueba abstracto generado por Fastest con su ley de refinamiento asociada.
- **RefineExpr**: Es un módulo físico que se encarga del refinamiento de las expresiones de igualdad de un caso de prueba abstracto generado por Fastest basado en su correspondiente regla de refinamiento.
- **ConstantGenerator**: Es un módulo físico que se encarga de traducir las constantes incluidas en los casos de prueba abstractos generados por Fastest a sus correspondientes constantes en el lenguaje de implementación del sistema a refinar (en esta tesis JAVA).

Para más información, consultar:

- el apéndice D que describe las **interfaces de los módulos** y
- el apéndice E que describe la **guía de módulos**.

6.3. Implementaciones de los módulos del diseño

Por cada módulo del diseño de la sección 6.2 se ha creado una clase JAVA con el mismo nombre del módulo. Por cada subrutina que exporta un módulo en el apéndice D que describe las **interfaces de los módulos** se definió un método dentro de la clase JAVA correspondiente que implementa la función de la subrutina. Esto se hizo para respetar el diseño lo más fielmente posible y para sea más fácil su comprensión. Como sucede en la mayoría de los proyectos, éstas no fueron las únicas clases creadas ni los únicos métodos definidos sino que se implementaron otras clases y métodos que facilitaron la implementación del diseño. Solamente se agregaron subrutinas en el interior de los módulos para ir resolviendo problemas cada vez más pequeños y más simples así como también algunas clases auxiliares para facilitar la resolución de estos subproblemas. Las interfaces de los módulos, permanecieron sin modificación.

A continuación se muestra el código fuente en JAVA correspondiente a las clases que implementan los módulos que se consideran más importantes y ejemplificativos del sistema de refinamiento a JAVA desarrollado en esta tesis.

El código fuente de otras clases que implementan otros módulos de diseño, que a continuación no se muestran, puede consultarse en el apéndice B.

Para obtener una copia completa del código fuente del sistema de refinamiento a JAVA desarrollado en esta tesis consultar a:

- **Pablo D. Coca** (autor de esta tesis) o
- **Maximiliano Cristiá** (director de esta tesis)

6.3.1. TCRL_Parser

El analizar lexicógrafo y parser para el lenguaje TCRL v2.0 se realizaron con la herramienta JavaCC (ver sección 6.1.4) como se ha comentado. JavaCC genera automáticamente el analizador lexicográfico y el parser escritos en lenguaje JAVA para una gramática dada en una notación especial de JavaCC pasándosela en un archivo **.jj**. A continuación se describirán algunas de las partes más relevantes que componen dicho archivo.

Lo primero que hay setear son las opciones de configuración que ofrece JavaCC que permiten especificar algunos de los comportamientos del analizador lexicográfico y parser a generarse, entre otras cosas, las cuales se asignaron así:

```
1  options {
2    LOOKAHEAD = 1;
3    CHOICE_AMBIGUITY_CHECK = 2;
4    OTHER_AMBIGUITY_CHECK = 1;
5    STATIC = true;
6    DEBUG_PARSER = false;
7    DEBUG_LOOKAHEAD = false;
8    DEBUG_TOKEN_MANAGER = false;
9    ERROR_REPORTING = true;
10   JAVA_UNICODE_ESCAPE = false;
11   UNICODE_INPUT = false;
12   IGNORE_CASE = false;
13   USER_TOKEN_MANAGER = false;
14   USER_CHAR_STREAM = false;
15   BUILD_PARSER = true;
16   BUILD_TOKEN_MANAGER = true;
17   SANITY_CHECK = true;
18   FORCE_LA_CHECK = true;
19 }
```

Código 6.1: Opciones de configuración de JavaCC

Dado que JavaCC permite la inclusión de código JAVA dentro del archivo **.jj** que contiene la gramática del lenguaje para el cual se desea generar el analizador lexicográfico y el parser, se han incluido en este archivo las clases que implementan los submódulos físicos del módulo lógico AST (ver 6.2) que son los que representan el árbol de sintaxis abstracta de una ley de refinamiento escrita en el lenguaje TCRL v2.0. La mayoría de dichas clases en código JAVA fueron expuestas en la sección 6.3.

Se definen los *tokens* propios de la gramática de TCRL v2.0, a continuación se muestran algunos de los principales:

```
1  TOKEN :
2  {
3
```

```
// ...

  <PREAMBLE: "PREAMBLE" <EOL> >                 : PLCodeState
| <EPILOGUE: "EPILOGUE" <EOL> >                 : PLCodeState
| <ABSTRACTION: "@ABSTRACTION" <EOL> >          : Abstraction
| <SYNONYM: "SYNONYM">
| <PLTYPE: "PLTYPE">
| <ENUMERATION: "ENUMERATION">
| <ARRAY: "ARRAY">
| <POINTER: "POINTER">
| <STRUCTURE: "STRUCTURE">
| <NUMBER: (["0"-"9"])+>
| <ID: ("*" ( ["a"-"z","A"-"Z","0"-"9","!","?","_","-"] )*) | (["a"-"z","A"-"Z"] ( ["a"-"z","A
    "-"Z","0"-"9","!","?","_","-","."] )*) >
| <EOL: "\n">
| <IMPLID: ("*" ( ["a"-"z","A"-"Z","0"-"9","!","?","_","-"] )*) | (["a"-"z","A"-"Z"] ( ["a"-"z
    ","A"-"Z","0"-"9","!","?","_","-"] )*) >

// ...

}
```

Código 6.2: Tokens para la gramática de TCRL v2.0 en JavaCC

Luego se continúa con la traducción de las reglas de producción de la gramática en BNF de TCRL v2.0 (ver sección A) a la notación especial que entiende JavaCC (para más información, consultar [22]). Se mostrarán como se tradujeron algunas de la reglas de producción para fines ejemplificativos:

```
//*********************************************************************************
TCRL_AST parse() :
{
  String                lawName;
  String                preamble;
  NodeRules             rules;
  String                epilogue;

}
{
  lawName       = LawName()
  preamble      = Preamble()
  rules         = Rules()
  epilogue      = Epilogue()
  <EOF>

  {
    return new TCRL_AST(        lawName,
                                preamble,
                                rules,
                                epilogue
                        );
  }

}
//*********************************************************************************
NodeRules Rules() :
{
    NodeRule    rule;
    NodeRules   rules = new NodeRules();
    Token       ruleName = new Token();
}
{
    <RULES>

        (
                <EOL>                           // for VISUAL ASPECTS
        )*


    (
```

```
42          {
43
44                  ruleName = new Token(); // RESET the TOKEN
45          }
46
47          [
48                  LOOKAHEAD(2)
49                  ruleName =  <ID>                // <RULENAME>
50                  <COLON>
51          ]
52
53          rule = Rule()
54
55          (
56                  <EOL>                           // for VISUAL ASPECTS
57          )*
58
59          {
60                  if (ruleName.image != null)
61                          rules.addRule(ruleName.image, rule);
62                  else
63                          rules.addRule(rule);
64          }
65
66      )*  // may have one or more REFINEMENT RULES
67
68      {
69          return rules;
70      }
71  }
72  //*****************************************************************************
73  NodeRule Rule():
74  {
75    NodeRule nodeRule;
76  }
77  {
78          LOOKAHEAD(3)
79
80              nodeRule = RuleSynonym()
81              {
82                return nodeRule;
83              }
84
85          |       nodeRule = RuleRefinement()
86              {
87                 return nodeRule;
88              }
89  }
90  //*****************************************************************************
```

Código 6.3: Algunas reglas de la gramática de TCRL v2.0 en JavaCC

Una vez que se tiene el **archivo .jj** que contiene la gramática de TCRL v2.0 junto con las clases que van formando el árbol de sintaxis abstracta se compila el archivo al cual se llamó **tcrl.jj** con JavaCC con la siguiente línea de comando:

```
>javacc tcrl.jj
```

generando las siguientes clases automáticamente que son las que implementan al **módulo físico TCRL_Parser** de la sección 6.2:

- **TCRL_Parser**: es el parser propiamente dicho del lenguaje TCRL v2.0.
- **TCRL_ParserTokenManager**: es el analizador lexicográfico del lenguaje TCRL v2.0.
- **TCRL_ParserConstants**: implementa las constantes utilizadas por el parser (TCRL _Parser).

- **TokenMgrError**: implementa los errores que detecta el analizador lexicográfico (TCRL_ParserTokenManager).
- **ParseException**: implementa excepciones que son arrojadas cuando el parser (TCRL_Parser) encuentra errores.
- **Token**: es una clase que representa tokens de TCRL v2.0. Cada objeto Token tiene asociado un campo de tipo *int* que representa a un token (PREAMBLE, @RULES, EPILOGUE, etc.) y un campo de tipo *String* que indica la secuencia de caracteres de la entrada que representa.
- **SimpleCharStream**: es una clase que envía caracteres al analizador lexicográfico (TCRL_ParserTokenManager).

6.3.2. TCaseRefClient

La clase **TCaseRefClient** inicia el proceso de refinamiento a JAVA y captura los eventos inherentes a este refinamiento. A continuación se expondrá la porción de código fuente más significativa.

NOTA: El código fuente está comentado para poder seguir de una manera más comprensiva y fácil su funcionamiento.

```
1  //--------------------------------------------------------------------------------
2  // VARIABLES
3  //--------------------------------------------------------------------------------
4  TCRL_AST                ast;
5  SectionManager  manager =       new SectionManager();
6  ConcreteTCase   ctc;
7  //--------------------------------------------------------------------------------
8
9
10 try
11 {
12 //--------------------------------------------------------------------------------
13 // Gets the FIRST ARGUMENT, the TCRL FILE (.tcrl)
14 //--------------------------------------------------------------------------------
15 File    tcrlFile        = new File(args[0]);
16 //--------------------------------------------------------------------------------
17
18 //--------------------------------------------------------------------------------
19 // PARSE the TCRL FILE and OBTAIN the ABSTRACT SYNTAX TREE (AST)
20 //--------------------------------------------------------------------------------
21 new TCRL_Parser(new FileInputStream(tcrlFile));
22
23 ast                                     = TCRL_Parser.parse();
24 //--------------------------------------------------------------------------------
25
26 //--------------------------------------------------------------------------------
27 // REPLACES the SYNONYMS in the TCRL AST
28 //--------------------------------------------------------------------------------
29 ast                                     = TCRL_PreProcessor.preProcess(ast);
30 //--------------------------------------------------------------------------------
31
32 //--------------------------------------------------------------------------------
33 // Gets the SECOND ARGUMENT, the TEX FILE (.tex)
34 //--------------------------------------------------------------------------------
35 FileSource      texFile = new FileSource(args[1]);
36 //--------------------------------------------------------------------------------
37
38 //--------------------------------------------------------------------------------
39 // STORES and GETS the TEXFILE in FASTEST
```

```
40 //-------------------------------------------------------------------------------
41 manager.put     (       new Key (       texFile.getName(),
42                                         Source.class
43                                 ),
44                         texFile
45                 );
46
47 Spec spec = (Spec) manager.get  (       new Key (       texFile.getName(),
48                                                         Spec.class
49                                                 )
50                                 );
51 //-------------------------------------------------------------------------------
52
53
54 //-------------------------------------------------------------------------------
55 // Gets the AxParaList of the ABSTRACT TEST CASE (.tex file)
56 //-------------------------------------------------------------------------------
57 List<AxPara> axParaList = Test.getAxParaList(spec);
58 //-------------------------------------------------------------------------------
59
60
61 //-------------------------------------------------------------------------------
62 // MUST BE ONLY 1 ABSTRACT TEST CASE peer .tex file
63 //-------------------------------------------------------------------------------
64 int axParaListSize = axParaList.size();
65
66 if (axParaListSize == 1)
67 {
68         //-------------------------------------------------------------------------------
69         // Gets the ONLY ABSTRACT TEST CASE that MUST BE.
70         //-------------------------------------------------------------------------------
71         AxPara          axPara          =       axParaList.get(0);
72         //-------------------------------------------------------------------------------
73
74         //-------------------------------------------------------------------------------
75         // EXTRACTS the NAME of the ABSTRACT TEST CASE.
76         // EXAMPLE: "Depositar_DNF_1_TCASE"
77         //-------------------------------------------------------------------------------
78         String          testName        =       SpecUtils.getAxParaName(axPara);
79         //-------------------------------------------------------------------------------
80
81         //-------------------------------------------------------------------------------
82         // Creates an instance of an ABSTRAC TEST CASE
83         //-------------------------------------------------------------------------------
84         AbstractTCaseImpl atc           =       new AbstractTCaseImpl(axPara, testName);
85         //-------------------------------------------------------------------------------
86
87         //-------------------------------------------------------------------------------
88         // REFINES the ABSTRACT TEST CASE and OBTAINS its generated CONCRETE TEST CASE
89         //-------------------------------------------------------------------------------
90         ctc                             =       RefineAST.refine(ast, atc);
91         //-------------------------------------------------------------------------------
92
93         //-------------------------------------------------------------------------------
94         // Prints on the "System.out" stream the JAVA CODE TEXT
95         // of the generated CONCRETE TEST CASE
96         //-------------------------------------------------------------------------------
97         System.out.println      (
98                                         Utils.printCTC(testName, ctc)
99                                 );
100        //-------------------------------------------------------------------------------
101 }
```

Código 6.4: TCaseRefClient

6.3.3. RefineAST

RefineAST es la clase encargada del refinamiento propiamente dicho y de enlazar el flujo de datos con las otras clases. El método principal de RefineAST es *refine()*, cuya signatura es:

```
ConcreteTCase refine(TCRL_AST ast, AbstractTCase atc)
```

Este método toma el árbol de sintaxis abstracta *ast* correspondiente a una ley de refinamiento de TCRL v2.0, que parseó la clase TCRL_Parser previamente, y un caso de prueba abstracto *atc* generado por Fastest, correspondiente a la especificación del sistema a refinar, y devuelve un caso de prueba concreto *ConcreteTCase*. Se expondrá la porción de código fuente más significativa de la clase que implementa al módulo físico RefineAST (consultar la sección 6.2).

NOTA: El código fuente está comentado para poder seguir de una manera más comprensiva y fácil su funcionamiento.

```
1  /**
2  * REFINES an ABSTRACT SYNTAX TREE based on its ABSTRACT TEST CASE.
3  * This is the most important method of the class "RefineAST".
4  *
5  * @author       Pablo D. Coca
6  * @since                1.0
7  * @version      2.0
8  *
9  * @param                ast             the ABSTRACT SYNTAX TREE
10 * @param                atc             the ABSTRACT TEST CASE
11 *
12 * @return               the CONCRETE TEST CASE wrote in the IMPLMENTATION LANGUAJE
13 **/
14 public static ConcreteTCase refine(TCRL_AST ast, AbstractTCase atc)
15 {
16 //-------------------------------------------------------------------------------
17 // SET the PREAMBLE and EPILOGUE of CTC EQUALS to ATC
18 //-------------------------------------------------------------------------------
19 ctc.setPreamble(ast.getPreamble());
20 ctc.setEpilogue(ast.getEpilogue());
21 //-------------------------------------------------------------------------------
22
23 //-------------------------------------------------------------------------------
24 // Gets the AST RULES
25 //-------------------------------------------------------------------------------
26 NodeRules rules = ast.getRules();
27 //-------------------------------------------------------------------------------
28
29 //-------------------------------------------------------------------------------
30 // GENERATES the "SpecIDs" TABLE.
31 //-------------------------------------------------------------------------------
32 specIDsToRulenameTable(rules);
33 //-------------------------------------------------------------------------------
34
35 //-------------------------------------------------------------------------------
36 // GENERATES the TABLE that maps "SpecIDs" with its Expr.
37 //-------------------------------------------------------------------------------
38 generateSpecIDsToExprTable(atc);
39 //-------------------------------------------------------------------------------
40
41 //-------------------------------------------------------------------------------
42 // GENERATES THE PARTICULAR REFINEMENT of each PREDICATE
43 //-------------------------------------------------------------------------------
44
45
46 //-------------------------------------------------------------------------------
47 // Gets the AxPara of the ABSTRACT TEST CASE
```

```
48 //-------------------------------------------------------------------------------
49 AxPara axPara = atc.getMyAxPara();
50 //-------------------------------------------------------------------------------
51
52 //-------------------------------------------------------------------------------
53 // Gets the PREDICATE of the ABSTRACT TEST CASE
54 Pred refPred = SpecUtils.getAxParaPred(axPara);
55 //-------------------------------------------------------------------------------
56
57 //-------------------------------------------------------------------------------
58 // ITERATION VARIABLES
59 //-------------------------------------------------------------------------------
60 AbstractRepository<Pred>        predRep = refPred.accept(new AndPredClausesExtractor());
61 AbstractIterator<Pred>          predIt  = predRep.createIterator();
62 //-------------------------------------------------------------------------------
63 while ( predIt.hasNext() )
64 {
65 Pred pred = predIt.next();              // Gets a PREDICATE
66
67 //-------------------------------------------------------------------------------
68 // CHECK is the PREDICATE is an ASSIGNMENT
69 //-------------------------------------------------------------------------------
70 if (    isAnAssignment(pred) )
71 {
72         MemPred         memPred =       ((MemPred) pred);
73
74         if (memPred.getMixfix())        //      getMixfix = true
75         {
76                 specID                  =       getSpecID(memPred);
77                 specIDAssignment        =       getSpecIDAssignment(memPred);
78
79                 if (    existRulename(specID)   )
80                 {
81                         // SEEKS for the RULE that REFINE the "specID" VARIABLE
82                                         ruleName        =       specIDsToRulenameTable.get(
                                                    specID);
83                         NodeRule        rule            =       rules.getRule(ruleName);
84                         RuleRefinement  ruleRefinement  =       (RuleRefinement) rule;
85
86                         Expr            specIDZType     = specIDsToExprTable.get(specID);
87
88                         //-----------------------------------------------------
89                         // SELECT the ZTYPE of the Z VARIABLE "specID"
90                         // in the DECLARATION of the SCHEMA
91                         //-----------------------------------------------------
92                         selectSpecIDZType(ruleRefinement, specIDAssignment, specIDZType);
93                         //-----------------------------------------------------
94                 }
95                 else
96                 {
97                         //-----------------------------------------------------
98                         // ERROR: VARIBLE NOT SPECIFIED
99                         Utils.printErrorVariableNotSpecified(specID);
100                        //-----------------------------------------------------
101                }
102        }
103        else
104        {
105                // memPred.getMixfix() = false
106                // DO NOTHING
107        }
108 }
109 else
110 {
111        // the PREDICATE is not an ASSIGNMENT (HAS NOT the form: VAR = EXP)
112        // DO NOTHING
113 }
114
115 }
116 //-------------------------------------------------------------------------------
117
```

```
118
119 //-------------------------------------------------------------------------
120 //RETURN the CONCRETE TEST CASE
121 //-------------------------------------------------------------------------
122 return ctc;
123 //-------------------------------------------------------------------------
124 }
```

Código 6.5: RefineAST

6.3.4. RefineExpr

RefineExpr es la clase que refina expresiones escritas en el lenguaje de especificación del sistema a refinar (en esta tesis, Z). El método principal de RefineExpr es *refineExpr()*, cuya signatura es:

`TCaseAssignment refineExpr(String implID, NodeType nodeType, RuleRefinement rule, Expr expr)`

Este método refina una expresión particular *expr* escrita en el lenguaje de especificación del sistema a refinar basándose en la regla de refinamiento correspondiente, *rule*, y devolviendo una asignación *TCaseAssignmnet*, escrita en código del lenguaje de implementación del sistema a refinar, basada principalmente en la clase que implementa al módulo de diseño *ConstantGenerator* (ver sección 6.2). Se expondrá la porción de código fuente más significativa de la clase que implementa al módulo físico RefineExpr (consultar la sección 6.2).

NOTA: El código fuente está comentado para poder seguir de una manera más comprensiva y fácil su funcionamiento.

```
1  /**
2  * REFINES a SPECIFICATION EXPRESSION (expr).
3  *
4  * @version      2.0
5  *
6  * @param        implID          the current IMPLEMENTATION ID.
7  * @param        nodeType        the current TYPE of the NODE associated to the REFINEMENT RULE
       .
8  * @param        rule            the current REFINEMENT RULE.
9  * @param        expr            the SPECIFICATION EXPRESSION to REFINE.
10 *
11 * @return       the ASSIGNEMENT of the REFINATION.
12 **/
13 public static TCaseAssignment refineExpr        (       String          implID,
14                                                          NodeType        nodeType,
15                                                          RuleRefinement  rule,
16                                                          Expr            expr
17                                                  )
18 throws IllegalArgumentException
19 {
20         TCaseAssignment assignment;
21
22
23         //-------------------------------------------------------------------------
24         // * PLTYPE
25         //-------------------------------------------------------------------------
26         if (nodeType instanceof NodePLType)
27         {
28                 assignment      =       refineWithPLType(
29                                                                  implID,
30                                                                  nodeType,
31                                                                  rule,
```

```
32                                                              expr
33                                                      );
34          }
35          //-------------------------------------------------------------------------------
36
37
38          //-------------------------------------------------------------------------------
39          // * ENUMERATION
40          //-------------------------------------------------------------------------------
41          else if (nodeType instanceof NodeEnumeration)
42          {
43                  assignment      =       refineWithNodeEnumeration
44                                                      (
45                                                              implID,
46                                                              nodeType,
47                                                              rule,
48                                                              expr
49                                                      );
50          }
51          //-------------------------------------------------------------------------------
52
53
54          //-------------------------------------------------------------------------------
55          // * POINTER
56          //-------------------------------------------------------------------------------
57          else if (nodeType instanceof NodePointer)
58          {
59                  assignment      =       refineWithPointer(
60                                                              implID,
61                                                              nodeType,
62                                                              rule,
63                                                              expr
64                                                       );
65
66          }
67          //-------------------------------------------------------------------------------
68
69
70          //-------------------------------------------------------------------------------
71          // * ARRAY
72          //-------------------------------------------------------------------------------
73          else if (nodeType instanceof NodeArray)
74          {
75                  assignment      =       refineWithArray (
76                                                              implID,
77                                                              nodeType,
78                                                              rule,
79                                                              expr
80                                                      );
81          }
82          //-------------------------------------------------------------------------------
83
84
85          //-------------------------------------------------------------------------------
86          // Para FUTUROS DESARROLLOS
87          //-------------------------------------------------------------------------------
88
89
90          //-------------------------------------------------------------------------------
91          // * STRUCTURE
92          //-------------------------------------------------------------------------------
93          // Quedaría parecido a la IMPLMENTACIÓN de DIEGO HOLLMAN.
94          // Habría que agregar tambien el uso de JAVA REFLEXION para acceder siempre
95          // a los miembros de la CLASE según sugerencia de MAXI.
96          //-------------------------------------------------------------------------------
97          //else if (nodeType instanceof NodeStructure)
98          //{
99          //      assignment      =       refineWithStructure(...);
100         //}
101         //-------------------------------------------------------------------------------
102
```

```
103
104         //-------------------------------------------------------------------------------
105         // * LIST
106         //-------------------------------------------------------------------------------
107         // Quedaría parecido a la IMPLMENTACIÓN de DIEGO HOLLMAN.
108         // Habría que agregar tambien el uso de JAVA REFLEXION para acceder siempre
109         // a los miembros de la CLASE según sugerencia de MAXI.
110         //-------------------------------------------------------------------------------
111         //else if (nodeType instanceof NodeList)
112         //{
113         //      assignment      =       refineWithList(...);
114         //}
115         //-------------------------------------------------------------------------------
116
117
118         //-------------------------------------------------------------------------------
119         // * FILE
120         //-------------------------------------------------------------------------------
121         //else if (nodeType instanceof NodeFile)
122         //{
123         //      assignment      =       refineWithFile(...);
124         //}
125         //-------------------------------------------------------------------------------
126
127
128         //-------------------------------------------------------------------------------
129         // * DATABASE
130         //-------------------------------------------------------------------------------
131         //else if (nodeType instanceof NodeDB)
132         //{
133         //      assignment      =       refineWithDB(...);
134         //}
135         //-------------------------------------------------------------------------------
136
137
138         //-------------------------------------------------------------------------------
139         // EXCEPTIONS
140         //-------------------------------------------------------------------------------
141         else
142         {
143             throw new IllegalArgumentException();
144         }
145         //-------------------------------------------------------------------------------
146
147         return assignment;
148
149 }
150 //-----------------------------------------------------------------------------
151
152
153 /**
154 * REFINES a PROGRAMMING LANGUAGE NODE.
155 *
156 * @version      2.0
157 *
158 * @param        implID          the current IMPLEMENTATION ID.
159 * @param        nodeType        the current TYPE of the NODE associated to the REFINEMENT RULE
    .
160 * @param        rule            the current REFINEMENT RULE.
161 * @param        expr            the SPECIFICATION EXPRESSION to REFINE.
162 *
163 * @return       the ASSIGNEMENT of the REFINATION.
164 **/
165 public static TCaseAssignment refineWithPLType  (       String          implID,
166                                                          NodeType        nodeType,
167                                                          RuleRefinement  rule,
168                                                          Expr            expr
169                                                  )
170 {
171         String  ref                     =       implID + " = ";
172
```

```
173         String  plTypeType              =       ((NodePLType) nodeType).getType();
174
175         String  constantGenerated       =       ConstantGenerator.PLTypeToString
176                                                         (
177                                                                 plTypeType,
178                                                                 expr
179                                                         );
180
181         ref = ref.concat(
182                                         constantGenerated
183                                 +       ";"
184                         );
185
186         return new TCaseAssignment((rule.getSpecIDs()).get(0), ref);
187 }
```

Código 6.6: RefineExpr

6.3.5. ConstantGenerator

ConstantGenerator es la clase que traduce las constantes que se encuentran en el caso de prueba abstracto generado por Fastest (escrito en el lenguaje de especificación del sistema a refinar, en esta tesis Z) en valores para el lenguaje de implementación del sistema a refinar (en esta tesis, JAVA). Se expondrá la porción de código fuente más significativa de la clase que implementa al módulo físico ConstantGenerator (consultar la sección 6.2).

NOTA: El código fuente está comentado para poder seguir de una manera más comprensiva y fácil su funcionamiento.

```
1  /**
2   * Represents the MODULE that GENERATES CONSTANT for the REFINEMENT process
3   *
4   * @author  Pablo D. Coca
5   *
6   * @since   v1.0
7   *
8   * @version 2.0
9   */
10 public class ConstantGenerator
11 {
12     public static String PLTypeToString(String type, Expr expr) throws
           IllegalArgumentException
13     {
14         if (type.equals("byte"))
15         {
16             return toByte(expr);
17         }
18
19         else if (type.equals("short"))
20         {
21             return toShort(expr);
22         }
23
24         else if (type.equals("int"))
25         {
26             return toInt(expr);
27         }
28
29         else if (type.equals("long"))
30         {
31             return toLong(expr);
32         }
33
34         else if (type.equals("float"))
35         {
```

```
36                  return toFloat(expr);
37          }
38
39          else if (type.equals("double"))
40          {
41                  return toDouble(expr);
42          }
43
44          else if (type.equals("char"))
45          {
46              return toChar(expr);
47          }
48
49          else if (type.equals("String"))         // String is supported as PRIMITIVE TYPE
50          {
51              return toString(expr);
52          }
53
54          else if (type.equals("boolean"))
55          {
56              return toBoolean(expr);
57          }
58
59          else
60          {
61                  throw new IllegalArgumentException();
62          }
63
64          //-------------------------------------------------
65          // INFO:
66          //-------------------------------------------------
67          // JAVA - Primitive Data Types
68          //-------------------------------------------------
69          //      Data Type       Default Value (for fields)
70          //      byte    0
71          //      short   0
72          //      int     0
73          //      long    0L
74          //      float   0.0f
75          //      double  0.0d
76          //      char    '\u0000'
77          //      String (or any object)          null
78          //      boolean false
79          //-------------------------------------------------------------------------------
80          //      The String class is not technically a primitive data type,
81          //      but considering the special support given to it by the language,
82          //      you'll probably tend to think of it as such.
83          //      You'll learn more about the String class in Simple Data Objects
84          //-------------------------------------------------------------------------------
85          //      You may have noticed that the new keyword isn't used
86          //      when initializing a variable of a primitive type.
87          //      Primitive types are special data types built into the language;
88          //      they are not objects created from a class.
89          //-------------------------------------------------------------------------------
90          //      use 'single quotes' for char literals and "double quotes" for String literals.
91          //      The Java programming language also supports a few special escape sequences
92          //      for char and String literals:
93          //      \b (backspace), \t (tab), \n (line feed), \f (form feed),
94          //      \r (carriage return), \" (double quote), \' (single quote) and
95          //      \\ (backslash).
96          //-------------------------------------------------------------------------------
97      }
98
99
100     //-------------------------------------------------------------------------------
101     // BYTE
102     //-------------------------------------------------------------------------------
103     public static String toByte(Expr expr)
104     {
105         Byte    ret;
106         String  str             =       exprToString(expr);
```

```
107
108         if      (str.length()   >       1)
109         {
110                 String  str2            =       stringToByte(str);
111                         ret             =       new Byte(str2);
112         }
113         else
114         {
115                         ret             =       new Byte(str);
116         }
117
118                 return ret.toString();
119     }
120     //---------------------------------------------------------------------------
```

Código 6.7: ConstantGenerator

Capítulo 7

Casos de Estudio

En este capítulo se mostrará a través de algunos casos de estudio, cómo funciona el sistema de refinamiento a JAVA realizado en esta tesis de grado.

Por cada caso de estudio se expondrá:

- las **operaciones** en Z de la especificación del caso de estudio para las cuáles se presentan a continuación los casos de prueba abstractos generados por Fastest.
- al menos un **caso de prueba abstracto** generado por Fastest para cada una de las operaciones a refinar.
- al menos una **implementación en JAVA** para las variables de las operaciones Z expuestas anteriormente,
- La descripción paso a paso de las correspondientes **reglas de refinamiento** escritas en TCRL v2.0 basadas en al menos una implementación.
- las **leyes de refinamiento** escritas en TCRL v2.0 para las operaciones Z expuestas anteriormente conformadas por:
  - el **nombre** de la ley de refinamiento,
  - la sección PREAMBLE que incluye la **implementación en JAVA** del sistema a refinar,
  - la sección @RULES que incluye las **reglas de refinamiento** pertinentes, de entre aquellas detalladas paso a paso anteriormente, para la ley de refinamiento en cuestión, y
  - la sección EPILOGUE con el código en JAVA que necesita ser cargado después del código generado por el sistema de refinamiento a JAVA.

  Para más información sobre la gramática de una ley de refinamiento consultar la sección 5.3.
- los **casos de prueba concretos** generados por el sistema de refinamiento a JAVA desarrollado en esta tesis (ver sección 6) correspondientes a los casos de pruebas abstractos y las leyes de refinamiento expuestos anteriormente.

7.1. Clases de Seguridad

Esta especificación formaliza el concepto de clase de seguridad. Una clase de seguridad es un par ordenado cuya primera componente es un número entero, llamado *nivel* (*level*), y la segunda es un conjunto de categorías (*categs*). Para consultar la especificación completa ver **el apéndice C.1**.

A continuación se expondrán sólo las operaciones de las cuáles se presentan posteriormente los casos de prueba abstractos generados por Fastest.

7.1.1. Operaciones

SCSetLevel

SCSetLevel setea el nivel de una clase de acceso siempre y cuando el nivel de entrada se establezca en el intervalo apropiado.

SCSetLevelOk

$$\Delta SecClass$$
$$l? : \mathbb{Z}$$
$$rep! : SCREPORT$$

$$0 \leq l? \leq MAXLEVEL$$
$$level' = l?$$
$$categs' = categs$$
$$rep! = scOk$$

SCSetLevelE

$$\Xi SecClass$$
$$l? : \mathbb{Z}$$
$$rep! : SCREPORT$$

$$l? < 0 \lor MAXLEVEL < l?$$
$$rep! = scError$$

$$SCSetLevel == SCSetLevelOk \lor SCSetLevelE$$

SCSetAddCat

SCSetAddCat agrega una categoría al conjunto de categorías de una clase de acceso siempre y cuando la cantidad actual de categorías no iguale a $MAXNCAT$. La otra precondición ($c? \notin categs$) está para advertir al programador que la categoría por ser agregada no debe estar ya en el conjunto de categorías.

SCAddCatOk

$\Delta SecClass$
$c? : CATEGORY$
$rep! : SCREPORT$

$c? \notin categs$
$\#categs < MAXNCAT$
$categs' = categs \cup \{c?\}$
$level' = level$
$rep! = scOk$

Hay 2 posibles errores: cuando *categs* está lleno y cuando una categoría existente está por ser agregada.

SCAddCatE1

$\Xi SecClass$
$c? : CATEGORY$
$rep! : SCREPORT$

$c? \in categs$
$rep! = scError$

SCAddCatE2

$\Xi SecClass$
$rep! : SCREPORT$

$\#categs = MAXNCAT$
$rep! = scCatFull$

La operación total es resumida a continuación.

$SCAddCatE == SCAddCatE1 \lor SCAddCatE2$
$SCAddCat == SCAddCatOk \lor SCAddCatE$

7.1.2. Casos de Prueba Abstractos

SCSetLevel_DNF_1_TCASE

SCSetLevel_DNF_1_TCASE

$level : \mathbb{Z}$
$categs : \mathbb{P}\, CATEGORY$
$MAXLEVEL : \mathbb{N}$
$l? : \mathbb{Z}$

...
$MAXLEVEL = 1$
$categs = \varnothing$
$l? = 1$
$level = 1$

SCAddCat_SP_14_TCASE

SCAddCat_SP_14_TCASE

$level : \mathbb{Z}$
$categs : \mathbb{P}\, CATEGORY$
$MAXNCAT : \mathbb{N}$
$c? : CATEGORY$

...
$MAXNCAT = 0$
$categs = \{category0, category1\}$
$level = 0$
$c? = category0$

NOTA: El sistema de refinamiento a JAVA sólo tiene en cuenta aquellas expresiones del predicado de un caso de prueba abstracto que son igualdades. Además de las expresiones de igualdad presentadas, los casos de pruebas abstractos tienen otro tipo de expresiones, llamadas precondiciones, que no se exponen porque podrían llegar a confundir al lector.

7.1.3. Implementación en JAVA

A continuación se expone una posible implementación en JAVA para el sistema.

NOTA: Todas las variables de la implementación están incluidas dentro de una clase de JAVA llamada `secClass` y se han declarado con el modificador `static` para que el código generado por el sistema de refinamiento a JAVA pueda acceder a realizar las asignaciones pertinentes a las variables.

```
1  public static class secClass
2  {
3          //-------------------------------------------------------------------------------
4          static  int             level;
5          static  int             maxLevel;
6          static  int             maxCategoryNumber;
7          static  int             assignationLevel;
8          static  short           category;
9          //-------------------------------------------------------------------------------
10         static  short[]         categorias;
11         //-------------------------------------------------------------------------------
12
13         //-------------------------------------------------------------------------------
14         public static void setLevel(int level)
15         {
16                 // ... CÓDIGO que IMPLEMENTA la FUNCIÓN ...
17         }
18         //-------------------------------------------------------------------------------
19
20         //-------------------------------------------------------------------------------
21         public static void addCatergory(short category)
22         {
23                 // ... CÓDIGO que IMPLEMENTA la FUNCIÓN ...
24         }
25         //-------------------------------------------------------------------------------
26 }
```

Código 7.1: Implementación para el caso de estudio SecClass

7.1.4. Reglas de Refinamiento

A continuación se describen paso a paso las reglas de refinamiento en función de las variables de especificación de las operaciones mostradas anteriormente y las correspondientes variables de implementación expuestas:

- **level**:

  ```
  static int level;
  ```

  La correspondiente regla de refinamiento de TCRL v2.0 asociada es:

  ```
  level ==> secClass.level : PLTYPE "int"
  ```

  La misma indica que la variable de especificación *level* se refina con la variable de implementación `secClass.level` (variable miembro `level` de la clase JAVA `secClass`) la cuál es del tipo primitivo de JAVA `int`.

- **categs**:

  ```
  static short[] categorias;
  ```

  La correspondiente regla de refinamiento de TCRL v2.0 asociada es:

  ```
  categs ==> secClass.categorias : ARRAY [PLTYPE "short"]
  ```

  La misma indica que la variable de especificación *categs* se refina con la variable de implementación `secClass.categs` la cuál es un arreglo del tipo primitivo de JAVA `short`.

- **MAXLEVEL**:

  ```
  static int maxLevel;
  ```

  La correspondiente regla de refinamiento de TCRL v2.0 asociada es:

  ```
  MAXLEVEL ==> secClass.maxLevel : PLTYPE "int"
  ```

- **l?**:

  ```
  static int assignationLevel;
  ```

  La correspondiente regla de refinamiento de TCRL v2.0 asociada es:

  ```
  l? ==> secClass.assignationLevel : PLTYPE "int"
  ```

- **MAXNCAT**:

```
static int maxCategoryNumber;
```

La correspondiente regla de refinamiento de TCRL v2.0 asociada es:

```
MAXNCAT ==> secClass. maxCategoryNumber : PLTYPE "int"
```

- **c?**:

```
static int category;
```

La correspondiente regla de refinamiento de TCRL v2.0 asociada es:

```
c? ==> secClass. category : PLTYPE "short"
```

NOTA: las correspondientes reglas de refinamiento recién presentadas son usadas dentro de la sección @RULES de las leyes de refinamiento que se expondrán a continuación para las operaciones presentadas anteriormente.

7.1.5. Leyes de Refinamiento

Notar que se incluyen en la sección @RULE de cada ley de refinamiento para las operaciones expuestas anteriormente, las reglas de refinamiento escritas en TCRL v2.0 detalladas paso a paso de la sección anterior 7.1.4.

Ley de Refinamiento para la operación SCSetLevel

```
1  secClass_SCSetLevel
2  package client.blogic.testing.refinement.java.tests.secClass;
3
4  //-----------------------------------------------------------------------------
5  // SPECIFICATION:       secClass.tex
6  // OPERATION:           SCSetLevel
7  //-----------------------------------------------------------------------------
8  // INFO:
9  //      Esta especificacion formaliza el concepto de clase de seguridad.
10 //      Una clase de seguridad es un par ordenado cuya primera componente
11 //      es un numero entero, llamado nivel, y la segunda es un conjunto de categorias.
12 //      La especificacion presenta un invariante de estado para cualquier clase de
13 //      seguridad y una serie de operaciones que modifican o consultan una clase de
14 //      seguridad.
15 //-----------------------------------------------------------------------------
16
17 //-----------------------------------------------------------------------------
18 public static class secClass
19 {
20         //-------------------------------------------------------------------------------
21         static  int             level;
22         static  int             maxLevel;
23         static  int             maxCategoryNumber;
24         static  int             assignationLevel;
25         static  short           category;
26         //-------------------------------------------------------------------------------
27         static  short[]         categorias;
28         //-------------------------------------------------------------------------------
29
30         //-------------------------------------------------------------------------------
```

```
31          public static void setLevel(int level)
32          {
33                  // ... CÓDIGO que IMPLEMENTA la FUNCIÓN ...
34          }
35          //-------------------------------------------------------------------------------
36
37          //-------------------------------------------------------------------------------
38          public static void addCatergory(short category)
39          {
40                  // ... CÓDIGO que IMPLEMENTA la FUNCIÓN ...
41          }
42          //-------------------------------------------------------------------------------
43
44  }
45  //-------------------------------------------------------------------------------
46  @RULES
47  //-------------------------------------------------------------------------------
48  regla01:        level           ==> secClass.level              : PLTYPE "int"
49  regla02:        categs          ==> secClass.categorias         : ARRAY[PLTYPE "short"]
50  regla03:        MAXLEVEL        ==> secClass.maxLevel           : PLTYPE "int"
51  regla04:        l?              ==> secClass.assignationLevel   : PLTYPE "int"
52  //-------------------------------------------------------------------------------
53  EPILOGUE
54  secClass.setLevel(secClass.assignationLevel);
```

Código 7.2: Ley de Refinamiento el caso de prueba abstracto SCSetLevel_DNF_1_TCASE

Ley de Refinamiento para la operación SCAddCat

```
1   secClass_SCAddCat
2   PREAMBLE
3   package client.blogic.testing.refinement.java.tests.secClass;
4
5   //-------------------------------------------------------------------------------
6   // SPECIFICATION:       secClass.tex
7   // OPERATION:           SCAddCat
8   //-------------------------------------------------------------------------------
9   // INFO:
10  //      Esta especificación formaliza el concepto de clase de seguridad.
11  //      Una clase de seguridad es un par ordenado cuya primera componente
12  //      es un numero entero, llamado nivel, y la segunda es un conjunto de categorías.
13  //      La especificación presenta un invariante de estado para cualquier clase de
14  //      seguridad y una serie de operaciones que modifican o consultan una clase de
15  //      seguridad.
16  //-------------------------------------------------------------------------------
17
18  //-------------------------------------------------------------------------------
19  public static class secClass
20  {
21          //-------------------------------------------------------------------------------
22          static  int             level;
23          static  int             maxLevel;
24          static  int             maxCategoryNumber;
25          static  int             assignationLevel;
26          static  short           category;
27          //-------------------------------------------------------------------------------
28          static  short[]         categorias;
29          //-------------------------------------------------------------------------------
30
31          //-------------------------------------------------------------------------------
32          public static void addCatergory(short category)
33          {
34                  // ... CÓDIGO de la IMPLEMENTACIÓN de la FUNCIÓN ...
35          }
36          //-------------------------------------------------------------------------------
37
38  }
39  //-------------------------------------------------------------------------------
```

```
@RULES
//-------------------------------------------------------------------------------
regla01:        level   ==> secClass.level              : PLTYPE "int"
regla02:        categs  ==> secClass.categorias         : ARRAY[PLTYPE "short"]
regla03:        MAXNCAT ==> secClass.maxCategoryNumber  : PLTYPE "int"
regla04:        c?      ==> secClass.category           : PLTYPE "short"
//-------------------------------------------------------------------------------
EPILOGUE
secClass.addCatergory(secClass.category);
```

Código 7.3: Ley de Refinamiento el caso de prueba abstracto SCAddCat_SP_14_TCASE

7.1.6. Casos de Prueba Concretos

A continuación se presenta el código generado por sistema de refinamiento a JAVA basado en los correspondientes casos de prueba abstractos y leyes de refinamiento expuestos anteriormente.

SCSetLevel_DNF_1_TCASE.java

```
//-------------------------------------------------------------------------------
// SCSetLevel_DNF_1_TCASE
//-------------------------------------------------------------------------------

package client.blogic.testing.refinement.java.tests.secClass;

public class SCSetLevel_DNF_1_TCASE
{

//-------------------------------------------------------------------------------
// SPECIFICATION:       secClass.tex
// OPERATION:           SCSetLevel
//-------------------------------------------------------------------------------
// INFO:
//      Esta especificacion formaliza el concepto de clase de seguridad.
//      Una clase de seguridad es un par ordenado cuya primera componente
//      es un numero entero, llamado nivel, y la segunda es un conjunto de categorias.
//      La especificacion presenta un invariante de estado para cualquier clase de
//      seguridad y una serie de operaciones que modifican o consultan una clase de
//      seguridad.
//-------------------------------------------------------------------------------

//-------------------------------------------------------------------------------
public static class secClass
{
        //-----------------------------------------------------------------------------------
        static  int             level;
        static  int             maxLevel;
        static  int             maxCategoryNumber;
        static  int             assignationLevel;
        static  short           category;
        //-------------------------------------------------------------------------------
        static  short[]         categorias;
        //-----------------------------------------------------------------------------------

        //-----------------------------------------------------------------------------------
        public static void setLevel(int level)
        {
                // ... CÓDIGO que IMPLEMENTA la FUNCIÓN ...
        }
        //-----------------------------------------------------------------------------------

        //-----------------------------------------------------------------------------------
        public static void addCatergory(short category)
        {
                // ... CÓDIGO que IMPLEMENTA la FUNCIÓN ...
        }
```

```
48          //--------------------------------------------------------------------------
49
50  }
51  //--------------------------------------------------------------------------
52
53          public static void runTest_SCSetLevel_DNF_1_TCASE_TCASE()
54          {
55                  secClass.maxLevel = 1;
56
57                  secClass.assignationLevel = 1;
58
59                  secClass.level = 1;
60
61                  //--------------------------------------------------------------------
62                  //                                      EPILOGUE
63                  //--------------------------------------------------------------------
64                  secClass.setLevel(secClass.assignationLevel);
65          }
66  }
```

Código 7.4: Caso de prueba concreto para el caso de prueba abstracto SCSetLevel_DNF_1_TCASE

SCAddCat _SP_14_TCASE.java

```
1   //--------------------------------------------------------------------------
2   // SCAddCat_SP_14_TCASE
3   //--------------------------------------------------------------------------
4
5   package client.blogic.testing.refinement.java.tests.secClass;
6
7   public class SCAddCat_SP_14_TCASE
8   {
9
10  //--------------------------------------------------------------------------
11  // SPECIFICATION:       secClass.tex
12  // OPERATION:           SCSetLevel
13  //--------------------------------------------------------------------------
14  // INFO:
15  //      Esta especificación formaliza el concepto de clase de seguridad.
16  //      Una clase de seguridad es un par ordenado cuya primera componente
17  //      es un numero entero, llamado nivel, y la segunda es un conjunto de categorías.
18  //      La especificación presenta un invariante de estado para cualquier clase de
19  //      seguridad y una serie de operaciones que modifican o consultan una clase de
20  //      seguridad.
21  //--------------------------------------------------------------------------
22
23  //--------------------------------------------------------------------------
24  public static class secClass
25  {
26          //------------------------------------------------------------------------------
27          static  int             level;
28          static  int             maxLevel;
29          static  int             maxCategoryNumber;
30          static  int             assignationLevel;
31          static  short           category;
32          //------------------------------------------------------------------------------
33          static  short[]         categorias;
34          //------------------------------------------------------------------------------
35
36          //------------------------------------------------------------------------------
37          public static void setLevel(int level)
38          {
39                  // ... CÓDIGO que IMPLEMENTA la FUNCIÓN ...
40          }
41          //------------------------------------------------------------------------------
42
43          //------------------------------------------------------------------------------
44          public static void addCatergory(short category)
```

```
45          {
46                  // ... CÓDIGO que IMPLEMENTA la FUNCIÓN ...
47          }
48          //-------------------------------------------------------------------------------
49
50  }
51  //-------------------------------------------------------------------------------
52
53          public static void runTest_SCAddCat_SP_14_TCASE_TCASE()
54          {
55                  secClass.maxCategoryNumber = 0;
56
57                  secClass.categorias[0] = -7438;
58
59                  secClass.categorias[1] = -7437;
60
61                  secClass.level = 0;
62
63                  secClass.category = -7438;
64
65                  //-------------------------------------------------------------------------
66                  //                                      EPILOGUE
67                  //-------------------------------------------------------------------------
68                  secClass.addCatergory(secClass.category);
69          }
70  }
```

Código 7.5: Caso de prueba concreto para el caso de prueba abstracto SCAddCat_SP_14_TCASE

7.2. Entradas de Teatro

Esta especificación formaliza un sistema de venta de entradas para un espectáculo teatral, con la particularidad que la función del primer día está reservada para los amigos de los actores.

7.2.1. Operaciones y Estado del sistema

$[PERSONA, ASIENTO]$

$TipoFuncion ::= estandar \mid primera$

$MENSAJE ::= ok \mid error$

TicketParaFunciones

$$
\begin{array}{l}
amigos : \mathbb{P}\, PERSONA \\
tipoFunc : TipoFuncion \\
vendidos : ASIENTO \pfun PERSONA \\
asientos : \mathbb{P}\, ASIENTO \\
\hline
tipoFunc = primera \Rightarrow \operatorname{ran} vendidos \subseteq amigos
\end{array}
$$

TicketParaFunciones representa el estado del sistema. Las variables que lo compones son:

- un conjunto de personas que representa a los amigos,

- una variable que indica el tipo de función,
- los asientos vendidos hasta el momento y
- el conjunto de los asientos.

VentaExitosa

$\Delta TicketParaFunciones$
$p? : PERSONA$
$a? : ASIENTO$
$r! : MENSAJE$

$a? \in (asientos \setminus \mathrm{dom}\ vendidos)$
$tipoFunc = primera \Rightarrow (p? \in amigos)$
$vendidos' = vendidos \cup \{a? \mapsto p?\}$
$asientos' = asientos$
$tipoFunc' = tipoFunc$
$amigos' = amigos$
$r! = ok$

AsientoNoDisponible

$\Xi TicketParaFunciones$
$a? : ASIENTO$
$p? : PERSONA$
$r! : MENSAJE$

$a? \notin (asientos \setminus \mathrm{dom}\ vendidos) \lor (tipoFunc = primera \land p? \notin amigos)$
$r! = error$

$Venta \triangleq VentaExitosa \lor AsientoNoDisponible$

Venta representa la operación total de la venta de una entrada. Si el asiento requerido no pertenece a los asientos disponibles, o se intenta comprar una entrada para una primera función no siendo un amigo, el resultado de la operación da *error* sin modificar el estado. Caso contrario, se vende la entrada y se hace el correspondiente cambio de estado.

7.2.2. Casos de Prueba Abstractos

EntradasTeatro_Venta_DNF_3_TCASE

Venta_DNF_3_TCASE

$amigos : \mathbb{P}\, PERSONA$
$tipoFunc : TipoFuncion$
$vendidos : ASIENTO \pfun PERSONA$
$asientos : \mathbb{P}\, ASIENTO$
$a? : ASIENTO$
$p? : PERSONA$

$...$
$amigos = \{persona1, persona2\}$
$asientos = \{asiento1, asiento2\}$
$a? = asiento2$
$vendidos = \{(asiento1, persona1), (asiento2, persona2)\}$
$p? = persona2$
$tipoFunc = primera$

NOTA: El sistema de refinamiento a JAVA sólo tiene en cuenta aquellas expresiones del predicado de un caso de prueba abstracto que son igualdades. Además de las expresiones de igualdad presentadas, los casos de pruebas abstractos tienen otro tipo de expresiones, llamadas precondiciones, que no se exponen porque podrían llegar a confundir al lector.

7.2.3. Implementación en JAVA

A continuación se expone una posible implementación en JAVA para el sistema.

NOTA: Todas las variables de la implementación están incluidas dentro de una clase de JAVA y se han declarado con el modificador `static` para que el código generado por el sistema de refinamiento a JAVA pueda acceder a realizar las asignaciones pertinentes a las variables.

```
1  public static class EntradasTeatro
2  {
3          static  String[]        amigos          =       new String[250];
4          static  int[]           asientos        =       new int[250];
5          static  boolean         tipoFuncion;
6          static  boolean         mensaje;
7
8          //-------------------------------------------------------------------------------
9          public static class Vendido
10         {
11                 int             asiento;
12                 String          persona;
13         }
14         //-------------------------------------------------------------------------------
15         static  Vendido[]       vendidos;
16         //-------------------------------------------------------------------------------
17
18         //-------------------------------------------------------------------------------
19         static  int             functionArgAsiento;
20         static  String          functionArgPersona;
21         //-------------------------------------------------------------------------------
22         public static boolean Venta(String persona, int asiento)
23         {
24                 // ... CÓDIGO de la IMPLEMENTACIÓN de la FUNCIÓN ...
25         }
26         //-------------------------------------------------------------------------------
27 }
```

Código 7.6: Implementación para el caso de estudio SecClass

7.2.4. Reglas de Refinamiento

A continuación se describen paso a paso las reglas de refinamiento en función de las variables de especificación de la operación mostrada anteriormente y las correspondientes variables de implementación expuestas:

- **amigos**:

```
static  String[] amigos = new String[250];
```

  La correspondiente regla de refinamiento de TCRL v2.0 asociada es:

```
amigos   ==> EntradasTeatro.amigos      : ARRAY[PLTYPE "String", 250]
```

  La misma indica que la variable de especificación *amigos* se refina con la variable de implementación `EntradasTeatro.amigos` (variable miembro `amigos` de la clase JAVA `EntradasTeatro`) la cuál es un arreglo de la clase de JAVA `String` de 250 componentes.

- **asientos**:

```
static  int[] asientos = new int[250];
```

  La correspondiente regla de refinamiento de TCRL v2.0 asociada es:

```
asientos ==> EntradasTeatro.asientos    : ARRAY[PLTYPE "int", 250]
```

  La misma indica que la variable de especificación *asientos* se refina con la variable de implementación `EntradasTeatro.asientos` (variable miembro `asientos` de la clase JAVA `EntradasTeatro`) la cuál es un arreglo del tipo primitivo de JAVA `int` de 250 componentes.

- **tipoFunc**:

```
static  boolean tipoFuncion;
```

  La correspondiente regla de refinamiento de TCRL v2.0 asociada es:

```
tipoFunc ==> EntradasTeatro.tipoFuncion : ENUMERATION
                                          [ PLTYPE "boolean",
                                           estandar --> "false",
                                           primera  --> "true"
                                          ]
```

  La misma indica que, mediante el tipo ENUMERATION de TCRL v2.0, la variable de especificación *tipoFunc* se refina con la variable de implementación `EntradasTeatro.tipoFunc` la cuál es del tipo primitivo de JAVA `boolean`. Se mapean además los posibles valores que puede asumir la variable de especificación *tipoFunc* con aquellos que puede asumir la variable de implementación `EntradasTeatro.tipoFunc`.

- **vendidos**:

```
public static class Vendido
{
        int      asiento;
        String   persona;
}

static  Vendido[]       vendidos;
```

La correspondiente regla de refinamiento de TCRL v2.0 asociada es:

```
vendidos ==> EntradasTeatro.vendidos : ARRAY
                                       [
                                         STRUCTURE
                                         [ Vendido,
                                           asiento: PLTYPE "int",
                                           persona: PLTYPE "String"
                                         ],
                                         250
                                       ]
             @ABSTRACTION
             @SET dom vendidos AS EntradasTeatro.vendidos.asiento;
             @SET ran vendidos AS EntradasTeatro.vendidos.persona;
             @ENDABSTRACTION
```

La misma indica que la variable de especificación *vendidos* se refina con la variable de implementación `EntradasTeatro.vendidos` la cuál es un arreglo de una clase de JAVA llamada `Vendido` que tiene como miembros a un tipo primitivo de JAVA `int` llamado `asiento` y a una clase de JAVA String llamada persona. Mediante la sección @ABSTRACTION se indica que los valores del dominio de la variable de especificación *vendidos*, la cuál es del tipo $ASIENTO \rightarrow PERSONA$, se asocian a la variable miembro `asiento` de la variable de implementación `EntradasTeatro.vendidos` y los valores del rango de la variable de especificación *vendidos* se asocian a la variable miembro `persona` de la variable de implementación `EntradasTeatro.vendidos`

- **a?**:

```
static  int     functionArgAsiento;
```

La correspondiente regla de refinamiento de TCRL v2.0 asociada es:

```
a?       ==> EntradasTeatro.functionArgAsiento  : PLTYPE "int"
```

La misma indica que la variable de entrada de la especificación *a?* se refina con la variable de implementación `EntradasTeatro.functionArgAsiento` la cuál del tipo primitivo de JAVA `int`.

- **p?**:

```
static  String  functionArgPersona;
```

  La correspondiente regla de refinamiento de TCRL v2.0 asociada es:

```
p?       ==> EntradasTeatro.functionArgPersona  : PLTYPE "String"
```

  La misma indica que la variable de entrada de la especificación *p?* se refina con la variable de implementación `EntradasTeatro.functionArgPersona` la cuál de la clase JAVA `String`.

NOTAS:

- la asignaciones de las variables de implementación `EntradasTeatro.functionArgAsiento` y `EntradasTeatro.functionArgPersona` generadas por el sistema de refinamiento a JAVA en el caso de prueba concreto que se expondrá posteriormente son indicadas como argumentos de la función `EntradasTeatro.Venta()` en la sección EPILOGUE de la ley de refinamiento que se expondrá.
- las reglas de refinamiento recién presentadas son usadas dentro de la sección @RULES de la ley de refinamiento que se expondrán a continuación para la operación presentada anteriormente.

7.2.5. Leyes de Refinamiento

Notar que se incluyen en la sección @RULE de la ley de refinamiento para la operación expuesta anteriormente, las reglas de refinamiento escritas en TCRL v2.0 detalladas paso a paso de la sección anterior 7.2.4.

Ley de Refinamiento para la operación EntradasTeatro_Venta

```
1  EntradasTeatro_Venta
2  PREAMBLE
3  package client.blogic.testing.refinement.java.tests.EntradasTeatro;
4
5  //-----------------------------------------------------------------------------
6  // SPECIFICATION:       EntradasTeatro.tex
7  // OPERATION:           Venta
8  //-----------------------------------------------------------------------------
9  // INFO: Sistema de venta de entradas para un espectáculo teatral,
10 // con la particularidad que la función del primer día está reservada
11 // para los amigos de los actores.
12 //-----------------------------------------------------------------------------
13
14
15
16 //-----------------------------------------------------------------------------
17 public static class EntradasTeatro
18 {
19         static  String[]        amigos          = new String[250];
20         static  int[]           asientos        = new int[250];
21         static  boolean         tipoFuncion;
22         static  boolean         mensaje;
23
```

```
24          //-------------------------------------------------------------------------------
25          public static class Vendido
26          {
27                  int             asiento;
28                  String          persona;
29          }
30          //-------------------------------------------------------------------------------
31          static  Vendido[]       vendidos;
32          //-------------------------------------------------------------------------------
33
34          //-------------------------------------------------------------------------------
35          static  int             functionArgAsiento;
36          static  String          functionArgPersona;
37          //-------------------------------------------------------------------------------
38          public static boolean Venta(String persona, int asiento)
39          {
40                  // ... CÓDIGO que IMPLEMENTA la FUNCIÓN ...
41          }
42          //-------------------------------------------------------------------------------
43  }
44  //-------------------------------------------------------------------------------
45  @RULES
46  //-------------------------------------------------------------------------------
47  regla01: amigos   ==> EntradasTeatro.amigos      : ARRAY[PLTYPE "String", 250]
48  regla02: asientos ==> EntradasTeatro.asientos    : ARRAY[PLTYPE "int", 250]
49  regla03: tipoFunc ==> EntradasTeatro.tipoFuncion : ENUMERATION
50                                                     [    PLTYPE "boolean",
51                                                          estandar --> "false",
52                                                          primera  --> "true"
53                                                     ]
54  //-------------------------------------------------------------------------------
55  regla04: vendidos ==> EntradasTeatro.vendidos    : ARRAY
56                                                     [
57                                                         STRUCTURE
58                                                         [ Vendido,
59                                                           asiento: PLTYPE "int",
60                                                           persona: PLTYPE "String"
61                                                         ],
62                                                         250
63                                                     ]
64                                  @ABSTRACTION
65                                  @PLCODE
66                                  import java.util.*;
67                                  @SET EntradasTeatro.vendidos.asiento AS dom vendidos;
68                                  @SET EntradasTeatro.vendidos.persona AS ran vendidos;
69                                  @ENDABSTRACTION
70  //-------------------------------------------------------------------------------
71  regla05: a?       ==> EntradasTeatro.functionArgAsiento : PLTYPE "int"
72  regla06: p?       ==> EntradasTeatro.functionArgPersona : PLTYPE "String"
73  //-------------------------------------------------------------------------------
74  EPILOGUE
75  EntradasTeatro.mensaje =
76          EntradasTeatro.Venta    (       EntradasTeatro.functionArgPersona,
77                                          EntradasTeatro.functionArgAsiento
78                                  );
```

Código 7.7: Ley de Refinamiento para la operación EntradasTeatro_Venta

7.2.6. Casos de Prueba Concretos

EntradasTeatro_Venta_DNF_3_TCASE.java

```
1  //-------------------------------------------------------------------------------
2  // EntradasTeatro_Venta_DNF_3_TCASE
3  //-------------------------------------------------------------------------------
4
5  package client.blogic.testing.refinement.java.tests.EntradasTeatro;
6
```

```
7  public class Venta_DNF_3_TCASE
8  {
9
10 //----------------------------------------------------------------------------
11 // SPECIFICATION:       EntradasTeatro.tex
12 // OPERATION:           Venta
13 //----------------------------------------------------------------------------
14 // INFO: Sistema de venta de entradas para un espectáculo teatral,
15 // con la particularidad que la función del primer día está reservada
16 // para los amigos de los actores.
17 //----------------------------------------------------------------------------
18
19
20
21 //----------------------------------------------------------------------------
22 public static class EntradasTeatro
23 {
24         static  int[]           amigos;
25         static  int[]           asientos;
26         static  boolean         tipoFuncion;
27         static  boolean         mensaje;
28
29         //------------------------------------------------------------------------------
30         public static class Vendido
31         {
32                 int             asiento;
33                 String          persona;
34         }
35         //------------------------------------------------------------------------------
36         static  Vendido[]       vendidos;
37         //------------------------------------------------------------------------------
38
39         //------------------------------------------------------------------------------
40         static  int             functionArgAsiento;
41         static  String          functionArgPersona;
42         //------------------------------------------------------------------------------
43         public static boolean Venta(String persona, int asiento)
44         {
45                 // ... CÓDIGO de la IMPLEMENTACIÓN de la FUNCIÓN ...
46         }
47         //------------------------------------------------------------------------------
48 }
49 //----------------------------------------------------------------------------
50
51         public static void runTest_Venta_DNF_3_TCASE_TCASE()
52         {
53                 EntradasTeatro.amigos[0] = "persona1";
54
55                 EntradasTeatro.amigos[1] = "persona2";
56
57                 EntradasTeatro.asientos[0] = -659652682;
58
59                 EntradasTeatro.asientos[1] = -659652681;
60
61                 EntradasTeatro.tipoFuncion = true;
62
63                 EntradasTeatro.vendidos[0].asiento = -659652682;
64
65                 EntradasTeatro.vendidos[0].persona =  "persona1";
66
67                 EntradasTeatro.vendidos[1].asiento = -659652681;
68
69                 EntradasTeatro.vendidos[1].persona = "persona2";
70
71                 EntradasTeatro.functionArgAsiento = -659652681;
72
73                 EntradasTeatro.functionArgPersona = "persona2";
74
75                 //------------------------------------------------------------------------
76                 //                                  EPILOGUE
77                 //------------------------------------------------------------------------
```

```
78                  EntradasTeatro.mensaje =
79                          EntradasTeatro.Venta    (       EntradasTeatro.functionArgPersona,
80                                                          EntradasTeatro.functionArgAsiento
81                                                  );
82          }
83  }
```

Código 7.8: Caso de prueba concreto para el caso de prueba abstracto EntradasTeatro_Venta_DNF_3_TCASE

7.3. Protocolo de Comunicación EOCP para el Microsatélite Científico SACI-1[12]

El SACI-1 es un satélite para aplicaciones científicas actualmente en desarrollo en el I.N.P.E. (Instituto Nacional de Pesquisas Espaciais, Brasil). El SACI-1 es el primero de una serie de microsatélites para órbitas bajas sobre el planeta Tierra y se compone de cuatro experimentos científicos, a saber:

- ORCAS: una investigación de los flujos de radiación cósmica anómalos.
- FOTSAT: un fotómetro de luminiscencia atmosférica para medir las emisiones de la atmosférica terrestre,
- PLASMEX: un estudio de la evolución de burbujas de plasma
- MAGNEX: una investigación del efecto del campo geomagnético sobre partículas cargadas.

La especificación de este caso de estudio es el modelo formal del protocolo de comunicación EOCP para el microsatélite científico SACI-1. Para consultar la especificación completa ver el **apéndice C.2**.

7.3.1. Operaciones

Memory_Load

Memory_Load carga la memoria con datos provenientes de OBDH. Es una de las operaciones más complejas , porque se debe formalizar cómo los 43 bytes dedicados a registros de experimentación pueden ser modificados evitando desbordamientos, dado que ellos pueden dar lugar a errores fatales de programa.

[1] `http://www.globalsecurity.org/space/world/brazil/saci.htm`
[2] `http://www2.dem.inpe.br/ijar/4spaperF.html`

*MemoryLoadOk*1

$\Delta ExpState;\ \Xi Time;\ \Xi Status$
$addr? : \mathbb{N}$
$data? : \text{seq}\ MDATA$
$rsp! : RTYPE$

$waiting = no$
$ccmd = ML$
$0 < addr?$
$data? \neq \langle\rangle$
$addr? + \#data? \leq 43$
$addr? + \#data? \leq mep$
$memd' = memd \oplus \{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\}$
$rsp! = MR$
$mdp' = mdp$
$mep' = mep$
$ped' = ped$
$memp' = memp$

*MemoryLoadOk*2

$\Delta ExpState;\ \Xi Time;\ \Xi Status$
$addr? : \mathbb{N}$
$data? : \text{seq}\ MDATA$
$rsp! : RTYPE$

$waiting = no$
$ccmd = ML$
$0 < addr?$
$data? \neq \langle\rangle$
$addr? + \#data? \leq 43$
$mep < addr? + \#data?$
$memd' = memd \oplus \{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\}$
$mep' = addr? + \#data?$
$rsp! = MR$
$mdp' = mdp$
$ped' = ped$
$memp' = memp$

$MemoryLoadE1 == [\Xi ExpState;\ addr? : \mathbb{N};\ data? : \text{seq}\ MDATA \mid waiting \neq no \lor ccmd \neq ML]$

$MemoryLoadE2 == [\Xi ExpState;\ addr? : \mathbb{N};\ data? : \text{seq}\ MDATA;\ low? : BIN \mid$
$addr? = 0 \lor data? = \langle\rangle \lor 43 < addr? + \#data?]$

$MemoryLoad == MemoryLoadOk1 \lor MemoryLoadOk2 \lor MemoryLoadE1 \lor MemoryLoadE2$

7.3.2. Casos de Prueba Abstractos

MemoryLoad_SP_2_TCASE

MemoryLoad_SP_2_TCASE

$memp, memd : \text{seq}\, MDATA$
$mdp, mep, ped : \mathbb{N}$
$ctime : TIME$
$acquiring, waiting, sending,$
$dumping, waitsignal : BIN$
$mode : CMODE$
$ccmd : CTYPE$
$addr? : \mathbb{N}$
$data? : \text{seq}\, MDATA$
$low? : BIN$

$...$
$mode = COF$
$ccmd = ML$
$memp = \varnothing$
$memd = \langle mdata0, mdata1 \rangle$
$mdp = 0$
$mep = 2$
$ped = 0$
$ctime = 0$
$acquiring = no$
$waiting = no$
$sending = no$
$dumping = no$
$waitsignal = no$
$low? = no$
$addr? = 1$
$data? = \langle mdata0 \rangle$

Para ver todos los casos de pruebas generados por Fastest (33 en total), además de este en particular, consultar el "**Apéndice C**"de la tesis de grado de Pablo Rodríguez Monetti [10].

NOTA: El sistema de refinamiento a JAVA sólo tiene en cuenta aquellas expresiones del predicado de un caso de prueba abstracto que son igualdades. Además de las expresiones de igualdad presentadas, los casos de pruebas abstractos tienen otro tipo de expresiones, llamadas precondiciones, que no se exponen porque podrían llegar a confundir al lector.

7.3.3. Implementación en JAVA

A continuación se expone una posible implementación en JAVA para el sistema.

NOTA: Todas las variables de la implementación están incluidas dentro de una clase de JAVA llamada `MemoryLoad` y se han declarado con el modificador `static` para que el código generado por el sistema de refinamiento a JAVA pueda acceder a realizar las asignaciones pertinentes a las variables.

```
1 public static class MemoryLoad
```

```
{
        static  int             memoryDumpPointer;
        static  int             lastMemoryUsedPointer;
        static  int             experimentationDataPointer;
        //-------------------------------------------------------------------------------
        static  double          currentTime;
        //-------------------------------------------------------------------------------
        static  boolean         acquiring;
        static  boolean         waiting;
        static  boolean         sending;
        static  boolean         dumping;
        static  boolean         waitsignal;
        //-------------------------------------------------------------------------------
        static  byte[]          memoryForProgram                = new byte[215];
        static  byte[]          memoryForExperimentation        = new byte[43];
        //-------------------------------------------------------------------------------
        static  boolean         mode;
        static  int             command;
        //-------------------------------------------------------------------------------
        static  int             address;
        static  byte[]          data                            = new byte[43];
        static  boolean         low;

        //-------------------------------------------------
        public static void MemoryLoad(int addr, byte[] data, boolean low)
        {
                // ... CÓDIGO que IMPLEMENTA la FUNCIÓN ...
        }
        //-------------------------------------------------
}
```

Código 7.9: Implementación para el caso de estudio Protocolo de Comunicación EOCP

7.3.4. Reglas de Refinamiento

A continuación se describen paso a paso las reglas de refinamiento en función de las variables de especificación de la operación mostrada anteriormente y las correspondientes variables de implementación expuestas:

- **mdp**:

  ```
  static  int     memoryDumpPointer;
  ```

  La correspondiente regla de refinamiento de TCRL v2.0 asociada es:

  ```
  mdp     ==> MemoryLoad.memoryDumpPointer : PLTYPE "int"
  ```

  La misma indica que la variable de especificación *mdp* se refina con la variable de implementación `MemoryLoad.memoryDumpPointer` (variable miembro `memoryDumpPointer` de la clase JAVA `MemoryLoad`) la cuál es del tipo primitivo de JAVA `int`.

- **mep**:

  ```
  static  int     lastMemoryUsedPointer;
  ```

  La correspondiente regla de refinamiento de TCRL v2.0 asociada es:

```
mep     ==> MemoryLoad.lastMemoryUsedPointer : PLTYPE "int"
```

- **ped**:

```
static  int     experimentationDataPointer;
```

  La correspondiente regla de refinamiento de TCRL v2.0 asociada es:

```
ped     ==> MemoryLoad.experimentationDataPointer : PLTYPE "int"
```

- **ctime**:

```
static  double  currentTime;
```

  La correspondiente regla de refinamiento de TCRL v2.0 asociada es:

```
ctime   ==> MemoryLoad.currentTime : PLTYPE "double"
```

- **acquiring**:

```
static  boolean acquiring;
```

  La correspondiente regla de refinamiento de TCRL v2.0 asociada es:

```
acquiring       ==> MemoryLoad.acquiring : PLTYPE "boolean"
```

- **waiting**:

```
static  boolean waiting;
```

  La correspondiente regla de refinamiento de TCRL v2.0 asociada es:

```
waiting ==> MemoryLoad.waiting : PLTYPE "boolean"
```

- **sending**:

```
static  boolean sending;
```

  La correspondiente regla de refinamiento de TCRL v2.0 asociada es:

```
sending ==> MemoryLoad.sending : PLTYPE "boolean"
```

- **dumping**:

```
static  boolean dumping;
```

La correspondiente regla de refinamiento de TCRL v2.0 asociada es:

```
dumping ==> MemoryLoad.dumping : PLTYPE "boolean"
```

- **waitsignal**:

```
static  boolean waitsignal;
```

La correspondiente regla de refinamiento de TCRL v2.0 asociada es:

```
waitsignal      ==> MemoryLoad.waitsignal : PLTYPE "boolean"
```

- **memp**:

```
static byte[] memoryForProgram  = new byte[215];
```

La correspondiente regla de refinamiento de TCRL v2.0 asociada es:

```
memp ==> MemoryLoad.memoryForProgram : ARRAY[PLTYPE "byte", 215]
```

La misma indica que la variable de especificación *memp* se refina con la variable de implementación `MemoryLoad.memoryForProgram` la cuál es un arreglo del tipo primitivo de JAVA `byte` de 215 componentes.

- **memd**:

```
static byte[] memoryForExperimentation  = new byte[43];
```

La correspondiente regla de refinamiento de TCRL v2.0 asociada es:

```
memd ==> MemoryLoad.memoryForExperimentation : ARRAY[PLTYPE "byte", 43]
```

La misma indica que la variable de especificación *memd* se refina con la variable de implementación `MemoryLoad.memoryForExperimentation` la cuál es un arreglo del tipo primitivo de JAVA `byte` de 43 componentes.

- **mode**:

```
static  boolean mode;
```

La correspondiente regla de refinamiento de TCRL v2.0 asociada es:

```
mode ==> MemoryLoad.mode : ENUMERATION
                           [ PLTYPE "boolean",
                             CON        -->     "true",
                             COF        -->     "false"
                           ]
```

La misma indica que, mediante el tipo ENUMERATION de TCRL v2.0, la variable de especificación *mode* se refina con la variable de implementación `MemoryLoad.mode` la cuál es del tipo primitivo de JAVA `boolean`. Se mapean además los posibles valores que puede asumir la variable de especificación *mode* con aquellos que debería asumir la variable de implementación `MemoryLoad.mode` para cada uno de esos valores.

- **ccmd**:

```
static  int command;
```

La correspondiente regla de refinamiento de TCRL v2.0 asociada es:

```
ccmd ==> MemoryLoad.command : ENUMERATION
                              [ PLTYPE "int",
                                RM      -->     "0",
                                SC      -->     "1",
                                IDA     -->     "2",
                                SDA     -->     "3",
                                TD      -->     "4",
                                RC      -->     "5",
                                MD      -->     "6",
                                ML      -->     "7",
                                LP      -->     "8"
                              ]
```

La misma indica que, mediante el tipo ENUMERATION de TCRL v2.0, la variable de especificación *ccmd* se refina con la variable de implementación `MemoryLoad.command` la cuál es del tipo primitivo de JAVA `int`. Se mapean además los posibles valores que puede asumir la variable de especificación *ccmd* con aquellos que debería asumir la variable de implementación `MemoryLoad.command` para cada uno de esos valores.

- **addr?**:

```
static  int     address;
```

La correspondiente regla de refinamiento de TCRL v2.0 asociada es:

```
addr?   ==> MemoryLoad.address : PLTYPE "int"
```

- **data?**:

```
static byte[] data      = new byte[43];
```

La correspondiente regla de refinamiento de TCRL v2.0 asociada es:

```
data? ==> MemoryLoad.data : ARRAY[PLTYPE "byte", 43]
```

- **low?**:

```
static  boolean low;
```

La correspondiente regla de refinamiento de TCRL v2.0 asociada es:

```
low?    ==> MemoryLoad.low : PLTYPE "boolean"
```

NOTA: las reglas de refinamiento recién presentadas son usadas dentro de la sección @RULES de la ley de refinamiento que se expondrá a continuación para la operación *MemoryLoad* presentada anteriormente.

7.3.5. Leyes de Refinamiento

Notar que se incluyen en la sección @RULE de la ley de refinamiento para la operación *MemoryLoad* expuesta anteriormente, las reglas de refinamiento escritas en TCRL v2.0 detalladas paso a paso de la sección anterior 7.3.4.

Ley de Refinamiento para MemoryLoad

```
1  Model_Test_Protocol_MemoryLoad
2  PREAMBLE
3  package client.blogic.testing.refinement.java.tests.EXP_Emulator;
4
5  //-----------------------------------------------------------------------------
6  // SPECIFICATION:       ModelTestProtocol.tex
7  // OPERATION:           MemoryLoad
8  //-----------------------------------------------------------------------------
9  // INFO: EXP-OBDH Communication Protocol (EOCP) for Scientific Microsatellite SACI-1
10 //
11 //-----------------------------------------------------------------------------
12
13 import client.blogic.testing.refinement.java.tests.EXP_Emulator.eop.ORCAS;
14 import client.blogic.testing.refinement.java.tests.EXP_Emulator.eop.EXP;
15 import client.blogic.testing.refinement.java.tests.EXP_Emulator.eop.Message;
16
17
18 //-----------------------------------------------------------------------------
19 public static class MemoryLoad
20 {
21         static  int             memoryDumpPointer;
22         static  int             lastMemoryUsedPointer;
23         static  int             experimentationDataPointer;
24         //-------------------------------------------------------------------------
25         static  double          currentTime;
26         //-------------------------------------------------------------------------
27         static  boolean         acquiring;
28         static  boolean         waiting;
29         static  boolean         sending;
30         static  boolean         dumping;
```

```
31          static  boolean         waitsignal;
32          //-------------------------------------------------------------------------------
33          static  byte[]          memoryForProgram                = new byte[215];
34          static  byte[]          memoryForExperimentation        = new byte[43];
35          //-------------------------------------------------------------------------------
36          static  boolean         mode;
37          static  int             command;
38          //-------------------------------------------------------------------------------
39          static  int             address;
40          static  byte[]          data                            = new byte[43];
41          static  boolean         low;
42
43          //-------------------------------------------------
44          public static void MemoryLoad(int addr, byte[] data, boolean low)
45          {
46                  // ... CÓDIGO que IMPLEMENTA la FUNCIÓN ...
47          }
48          //-------------------------------------------------
49
50
51  }
52  //-------------------------------------------------------------------------------
53  @RULES
54  //-------------------------------------------------------------------------------
55  regla01: mdp            ==> MemoryLoad.memoryDumpPointer                : PLTYPE "int"
56  regla02: mep            ==> MemoryLoad.lastMemoryUsedPointer            : PLTYPE "int"
57  regla03: ped            ==> MemoryLoad.experimentationDataPointer       : PLTYPE "int"
58  //-------------------------------------------------------------------------------
59  regla04: ctime          ==> MemoryLoad.currentTime                      : PLTYPE "double"
60  //-------------------------------------------------------------------------------
61  regla05: acquiring      ==> MemoryLoad.acquiring                        : PLTYPE "boolean"
62  regla06: waiting        ==> MemoryLoad.waiting                          : PLTYPE "boolean"
63  regla07: sending        ==> MemoryLoad.sending                          : PLTYPE "boolean"
64  regla08: dumping        ==> MemoryLoad.dumping                          : PLTYPE "boolean"
65  regla09: waitsignal     ==> MemoryLoad.waitsignal                       : PLTYPE "boolean"
66  //-------------------------------------------------------------------------------
67  // 5 BUFFERS de 43 ELEMENTOS de tipo MDATA (215 ELEMENTOS)
68  regla10: memp           ==> MemoryLoad.memoryForProgram         : ARRAY[PLTYPE "byte",  215]
69  //-------------------------------------------------------------------------------
70  // 1 BUFFER de 43 ELEMENTOS de tipo MDATA
71  regla11: memd           ==> MemoryLoad.memoryForExperimentation : ARRAY[PLTYPE "byte",  43]
72  //-------------------------------------------------------------------------------
73  regla12: mode           ==> MemoryLoad.mode             : ENUMERATION
74                                                              [ PLTYPE"boolean",
75                                                                CON -->     "true",
76                                                                COF -->     "false"
77                                                              ]
78  regla13: ccmd           ==> MemoryLoad.command          : ENUMERATION
79                                                              [ PLTYPE "int",
80                                                                RM  -->     "0",
81                                                                SC  -->     "1",
82                                                                IDA -->     "2",
83                                                                SDA -->     "3",
84                                                                TD  -->     "4",
85                                                                RC  -->     "5",
86                                                                MD  -->     "6",
87                                                                ML  -->     "7",
88                                                                LP  -->     "8"
89                                                              ]
90  //.............................................................................
91  //regla13: ccmd         ==> MemoryLoad.command          : ENUMERATION
92  //                                                          [ PLTYPE "String",
93  //                                                            RM  -->     "Reset Machine",
94  //                                                            SC  -->     "Send Clock",
95  //                                                            IDA -->     "Init Data Acq",
96  //                                                            SDA -->     "Stop Data Acq",
97  //                                                            TD  -->     "Transmit Data",
98  //                                                            RC  -->     "Reconfig",
99  //                                                            MD  -->     "Memory Dump",
100 //                                                            ML  -->     "Memory Load",
101 //                                                            LP  -->     "Load Param"
```

```
//                                                                    ]
//-------------------------------------------------------------------------------
regla14: addr?          ==> MemoryLoad.address          : PLTYPE "int"
regla15: data?          ==> MemoryLoad.data             : ARRAY[PLTYPE "byte",  43]
regla16: low?           ==> MemoryLoad.low              : PLTYPE "boolean"
//-------------------------------------------------------------------------------
EPILOGUE
MemoryLoad.MemoryLoad   (
                                MemoryLoad.address,
                                MemoryLoad.data,
                                MemoryLoad.low,
                        );
```

Código 7.10: Ley de Refinamiento para la operación MemoryLoad

7.3.6. Casos de Prueba Concretos

MemoryLoad_SP_2_TCASE.java

```
//-------------------------------------------------------------------------------
// MemoryLoad_SP_2_TCASE
//-------------------------------------------------------------------------------

import client.blogic.testing.refinement.java.tests.EXP_Emulator.eop.ORCAS;
import client.blogic.testing.refinement.java.tests.EXP_Emulator.eop.EXP;
import client.blogic.testing.refinement.java.tests.EXP_Emulator.eop.Message;

public class MemoryLoad_SP_2_TCASE
{

//-------------------------------------------------------------------------------
// SPECIFICATION:       ModelTestProtocol.tex
// OPERATION:           MemoryLoad
//-------------------------------------------------------------------------------
// INFO: EXP-OBDH Communication Protocol (EOCP) for Scientific Microsatellite SACI-1
//
//-------------------------------------------------------------------------------



//-------------------------------------------------------------------------------
public static class MemoryLoad
{
        static  int             memoryDumpPointer;
        static  int             lastMemoryUsedPointer;
        static  int             experimentationDataPointer;
        //-------------------------------------------------------------------------------
        static  double          currentTime;
        //-------------------------------------------------------------------------------
        static  boolean         acquiring;
        static  boolean         waiting;
        static  boolean         sending;
        static  boolean         dumping;
        static  boolean         waitsignal;
        //-------------------------------------------------------------------------------
        static  byte[]          memoryForProgram                = new byte[215];
        static  byte[]          memoryForExperimentation        = new byte[43];
        //-------------------------------------------------------------------------------
        static  boolean         mode;
        static  int             command;
        //-------------------------------------------------------------------------------
        static  int             address;
        static  byte[]          data                            = new byte[43],
        static  boolean         low;

        //-------------------------------------------------
        public static void MemoryLoad(int addr, byte[] data, boolean low)
        {
```

```
50                  // ... CÓDIGO que IMPLEMENTA la FUNCIÓN ...
51          }
52          //-------------------------------------------------
53  }
54  //-----------------------------------------------------------------------------
55
56          public static void runTest_MemoryLoad_SP_2_TCASE_TCASE()
57          {
58                  MemoryLoad.low = false;
59
60                  MemoryLoad.address = 1;
61
62                  MemoryLoad.data[0] = 121;
63
64                  MemoryLoad.mode = false;
65
66                  MemoryLoad.command = 7;
67
68                  MemoryLoad.memoryForExperimentation[0] = 121;
69
70                  MemoryLoad.memoryForExperimentation[1] = 122;
71
72                  MemoryLoad.memoryDumpPointer = 0;
73
74                  MemoryLoad.lastMemoryUsedPointer = 2;
75
76                  MemoryLoad.experimentationDataPointer = 0;
77
78                  MemoryLoad.currentTime = 0.0;
79
80                  MemoryLoad.acquiring = false;
81
82                  MemoryLoad.waiting = false;
83
84                  MemoryLoad.sending = false;
85
86                  MemoryLoad.dumping = false;
87
88                  MemoryLoad.waitsignal = false;
89
90                  //-------------------------------------------------------------------------
91                  //                                      EPILOGUE
92                  //-------------------------------------------------------------------------
93                  MemoryLoad.MemoryLoad   (
94                                                  MemoryLoad.address,
95                                                  MemoryLoad.data,
96                                                  MemoryLoad.low
97                                          );
98          }
99  }
```

Código 7.11: Caso de prueba concreto para el caso de prueba abstracto MemoryLoad_SP_2_TCASE

7.4. Otros casos de Estudios

Además de los casos de estudios expuestos se ha probado el sistema de refinamiento a JAVA con otros que a continuación comentan brevemente:

- **Agenda Telefónica**: sistema que modela el estado y el alta, baja, modificación y búsqueda de teléfonos de personas en una agenda telefónica.
- **Banco**: sistema que modela el estado y las operaciones básicas de depósito, extracción, etc. para un banco. Para más información consultar [8].

- **CVS**: sistema que modela el control de versiones de programas para diferentes tipos de usuarios (lector, editor, autor).

Capítulo 8

Conclusiones y Trabajo Futuro

8.1. Conclusiones

A lo largo de más de un año de trabajo se desarrolló el **prototipo** de un **sistema de refinamiento** de casos de prueba abstractos escritos en el **lenguaje Z** (ver sección 2.2) para **sistemas implementados** en el **lenguaje de programación JAVA** (ver sección 6.1.1) usando las construcciones básicas que ofrece este lenguaje.

Este sistema de refinamiento a JAVA está pensado para integrarse a la herramienta de software llamada **Fastest** (ver sección 3 y la bibligrafía [10]) la cual está ideada para **automatizar** en gran medida el proceso del **testing funcional basado en especificaciones formales** escritas en el lenguaje **Z** (ver sección 2.2) según el ciclo del testing funcional mostrado en la figura 2.1.

Además se ha desarrollado, para facilitar el proceso de refinamiento, el lenguaje denominado **TCRL v2.0** (ver sección 5) que permite describir cómo se realiza el proceso concretización o refinamiento de las variables en juego de la especificación e implementación. Se implementó también el analizador lexicográfico y el parser para el mismo en el lenguaje JAVA, los cuales forman parte del sistema de refinamiento.

De esta forma se ha cumplimentado otra porción más de la esperada herramienta final para el proceso de testing de software basado en especificaciones formales que sería de suma utilidad para cualquier proyecto de desarrollo de software, ya que permitiría la dedicación de los recursos humanos disponibles a tareas que requieran de las cualidades humanas de inteligencia y creatividad sin perder tiempo en actividades rutinarias y metódicas, como es en gran medida el proceso de testing de software, las que son ideales para asignar al poder de proceso de las computadoras.

Se ha podido desarrollar un prototipo de refinamiento a JAVA que concretiza casos de pruebas abstractos para sistemas con implementaciones que hacen uso de las construcciones básicas del lenguaje de programación JAVA (tipos primitivos de datos, arreglos, determinadas clases, etc.) de un número importante de especificaciones de sistemas, algunos de los cuales se exponen en la sección 7.

Se puede concluir entonces que el desarrollo del sistema de refinamiento a JAVA es satisfactorio, cumpliendo las expectativas que se tenían antes de la consumación del mismo y

mostrando que es viable seguir desarrollando contribuciones en esta línea de trabajo.

Dado que ningún sistema es la solución para todos los problemas del área en cuestión y cómo todo prototipo, éste no ofrece la funcionalidad completa del posible e imaginado sistema final. En la sección siguiente 8.2 se describen algunas de las posibles contribuciones a futuro para la evolución del sistema de refinamiento a JAVA desarrollado en este trabajo.

8.2. Trabajo Futuro

La siguiente lista presenta algunas de las funcionalidades o contribuciones que pueden agregarse al prototipo del sistema de refinamiento a JAVA desarrollado en esta tesis en futuros trabajos:

- Implementar el refinamiento para los tipos de variables (*type*) que son soportados por la gramática de TCRL v2.0 para una regla de refinamiento (ver sección 5.3.4) pero que no fueron implementados en esta tesis de grado, como son los tipos de variable: LIST, FILE, DB.

  El inconveniente que se tuvo para la implementación de este tipo de variables que soporta la gramática de TCRL v2.0 es que se necesita saber cómo es la estructura interna de las variables de la implementación (*ImplID* en TCRL 2.0) para saber en qué miembros o métodos asignar la o las correspondientes concretizaciones de los valores abstractos entregados en los casos de pruebas abstractos generados por FASTest.

  A priori se detallan 2 alternativas que podrían solucionar este inconveniente:

  - incluir campos en la sintaxis para cada tipo de variable de TCRL v2.0 para describir este tipo de información faltante, como son: la clase de JAVA de la cual es instancia una variable de la implementación (*ImplID* en TCRL 2.0), los métodos de *set*...() y/o *get*...() que permiten acceder a miembros privados u ocultos de la variable, etc.
  - investigar la posibilidad de la aplicación del framework **JAVA REFLECTION**[1] para dada una variable de la implementación del sistema (*ImplID* en TCRL 2.0), obtener por ejemplo información como:
    - instancia de qué clase es la variable,
    - cuáles son sus miembros (variables o métodos) privados y/o públicos que contiene,
    - etc.

- Implementar las funcionalidades que ofrece la gramática de TCRL v2.0 que aún no fueron implementadas en esta tesis de grado y que son citadas como una **NOTA** en las subsecciones de la sección 5.3.

- Desarrollar el **sistema de ejecución** de los casos de pruebas concretos generados por el sistema de refinamiento a JAVA desarrollado en esta tesis y desarrollar el **sistema de**

[1] `http://java.sun.com/developer/technicalArticles/ALT/Reflection/`

abstracción de la salida de la ejecución de esos casos de pruebas concretos para finalmente poder realizar la etapa de comprobación del ciclo de testing funcional mostrado en la figura 2.1 y determinar si existe un error o no en el sistema o programa testeado.

Apéndice A

TCRL v2.0 - Gramática en BNF

Para la formalización de la gramática de TCRL v2.0 se usó la notación BNF. A continuación se detalla su gramática en este formalismo mediante sus reglas de derivación:

⟨*refinementLaw*⟩ ::=⟨*lawName*⟩ *eol*
⟨*preamble*⟩ *eol*
⟨*rules*⟩ *eol*
⟨*epilogue*⟩

⟨*lawName*⟩ ::=⟨*indentifier*⟩

⟨*preamble*⟩ ::= **PREAMBLE** *eol*
⟨*PLCode*⟩
| ⟨*reference*⟩

⟨*epilogue*⟩ ::= **EPILOGUE** *eol*
⟨*PLCode*⟩
| ⟨*reference*⟩

⟨*reference*⟩ ::= ⟨*lawName*⟩**.**⟨*block*⟩

⟨*block*⟩ ::= **PREAMBLE**
| **EPILOGUE**
| **@RULES.**⟨*ruleName*⟩{**,**⟨*ruleName*⟩}

⟨*rules*⟩ ::= **@RULES** *eol*
⟨*rule*⟩{*eol* ⟨*rule*⟩}
| ⟨*reference*⟩

⟨*rule*⟩ ::= [⟨*ruleName*⟩**:**] ⟨*RuleSynonym*⟩
| [⟨*ruleName*⟩**:**] ⟨*RuleRefinement*⟩

⟨*ruleName*⟩ ::=⟨*identifier*⟩

⟨*RuleSynonym*⟩ ::=⟨*synonymName*⟩ == ⟨*type*⟩

⟨*synonymName*⟩ ::=⟨*identifier*⟩

⟨*RuleRefinement*⟩ ::= ⟨*SpecID*⟩ {**,**⟨*SpecID*⟩}
==>
⟨*ImplID*⟩**:**⟨*type*⟩ {**,**⟨*ImplID*⟩**:**⟨*type*⟩}
[**,**⟨*refinement*⟩]

⟨*SpecID*⟩ ::=⟨*identifier*⟩

⟨*ImplID*⟩ ::=⟨*identifier*⟩

⟨*refinement*⟩ ::=**@REFINEMENT** *eol*
{ [**@PLCODE** *eol*
⟨*PLCode*⟩]
@SET ⟨*ImplID*⟩ **AS** ⟨*LiveCode*⟩}*

⟨*LiveCode*⟩ ::= [⟨*preOp*⟩] ⟨*SpecID*⟩
|{[**@PLCODE** *eol*
⟨*PLCode*⟩]
@SPECID [⟨*preOp*⟩] ⟨*SpecID*⟩}*

⟨*PLCode*⟩ ::=⟨*text*⟩

⟨*preOp*⟩ ::= *dom* | *ran* | *first* | *second*

⟨*type*⟩ ::=⟨*synonym*⟩
|⟨*plType*⟩
|⟨*enumeration*⟩
|⟨*pointer*⟩
|⟨*array*⟩
|⟨*structure*⟩
|⟨*list*⟩
|⟨*file*⟩
|⟨*db*⟩

⟨*synonym*⟩ ::= **SYNONYM** ⟨*synonymName*⟩

⟨*plType*⟩ ::= **PLTYPE** "⟨*text*⟩"

⟨*enumeration*⟩ ::= **ENUMERATION[** ⟨*plType*⟩**,**{⟨*enumerationElement*⟩ --> ⟨*constant*⟩}*]

⟨*enumerationElement*⟩ ::=⟨*identifier*⟩

⟨*constant*⟩ ::=⟨*text*⟩

⟨*pointer*⟩ ::= **POINTER**[⟨*type*⟩]

⟨*structure*⟩ ::= **STRUCTURE**[⟨*identifier*⟩,⟨*element*⟩{,⟨*element*⟩}]

⟨*element*⟩ ::=⟨*identifier*⟩**:**⟨*type*⟩ [,"⟨*constant*⟩"]

⟨*array*⟩ ::= **ARRAY**[⟨*type*⟩[, ⟨*number*⟩]]

⟨*list*⟩ ::= **LIST**[⟨*identifier*⟩[,⟨*linkType*⟩][,⟨*next*⟩][,⟨*previous*⟩][,⟨*element*⟩{,⟨*element*⟩}][,⟨*memAllocation*⟩]]

⟨*linkType*⟩ ::=**SLL** | **DLL** | **CLL** | **DCLL**

⟨*next*⟩ ::=⟨*identifier*⟩

⟨*previous*⟩ ::=⟨*identifier*⟩

⟨*memAllocation*⟩ ::=⟨*text*⟩

⟨*file*⟩ ::= **FILE**[⟨*name*⟩,⟨*path*⟩,⟨*delimiter*⟩,⟨*text*⟩⟨*fileStructure*⟩[, **ENDOFLINE**⟨*text*⟩][, **ENDOFFILE**⟨*text*⟩][,⟨*fieldsRelation*⟩]]

⟨*name*⟩ ::=⟨*text*⟩

⟨*path*⟩ ::=⟨*text*⟩

⟨*demilimiter*⟩ ::=⟨*text*⟩

⟨*fileStructure*⟩ ::=**LINEAR** | **RPL** | **FPL**

⟨*fieldsRelations*⟩ ::=⟨*identifier*⟩**:**⟨*number*⟩{,⟨*identifier*⟩**:**⟨*number*⟩}

⟨*db*⟩ ::= **DB**[⟨*dbmsID*⟩,⟨*connectionID*⟩,⟨*tableName*⟩,⟨*columnID*⟩{,⟨*columnID*⟩}]

⟨*dbmsID*⟩ ::=⟨*text*⟩

⟨*connectionID*⟩ ::=⟨*text*⟩

⟨*tableName*⟩ ::=⟨*text*⟩

⟨*columnID*⟩ ::=⟨*identifier*⟩**:**⟨*columnType*⟩

⟨*columnType*⟩ ::=**INT** | **CHAR** | **VARCHAR**

⟨*identifier*⟩ ::=*letter*{*letter* | *digit* | **?** | **!** | _ | **-**}

⟨*text*⟩ ::=⟨*character*⟩{**,**⟨*character*⟩}

⟨*number*⟩ ::=*digit*{*digit*}

⟨*digit*⟩ ::=0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

⟨*letter*⟩ ::=*a* | *b* | *c* | *d* | *e* | *f* | *g* | *h* | *i* | *j* | *k* | *l* | *m* | *n* | *o* | *p* | *q* | *r* | *s* | *t* | *u* | *v* | *w* | *x* | *y* | *z* |
A | *B* | *C* | *D* | *E* | *F* | *G* | *H* | *I* | *J* | *K* | *L* | *M* | *N* | *O* | *P* | *Q* | *R* | *S* | *T* | *U* | *V* | *W* | *X* | *Y* | *Z*

⟨*character*⟩ ::=⟨*number*⟩ |
⟨*letter*⟩ |
! | # | $ | % | & | ′ | (|) |
∗ | + | , | − | . | / | | | : |; |<|=|>|? |
[|] | _ | ′ | { | }

Apéndice B

Más implementaciones de módulos de diseño

A continuación se muestra el código fuente en JAVA correspondiente a las clases que implementan algunos módulos de diseño que no se expusieron en la sección 6.3.

Para obtener una copia completa del código fuente del sistema de refinamiento a JAVA desarrollado en esta tesis consultar a:

- **Pablo D. Coca** (autor de esta tesis) o
- **Maximiliano Cristiá** (director de esta tesis)

ConcreteTCase

```
1  /**
2   * Representa un CASO de PRUEBA CONCRETO
3   *
4   *
5   * @author  Pablo D. Coca
6   * @since   v1.0
7   * @version 2.0
8   */
9  public class ConcreteTCase
10 {
11
12     private String preamble;
13     private String epilogue;
14     private List<TCaseAssignment> tCaseAssignments = new ArrayList<TCaseAssignment>();
15
16     public void setPreamble(String preable)
17     {
18         this.preamble = preable;
19     }
20
21     public void setEpilogue(String epilogue)
22     {
23         this.epilogue = epilogue;
24     }
25
26     //-------------------------------------------------------------------------
27     // 2 METHODS for ADD tCaseAssignments
28     //-------------------------------------------------------------------------
29     public void addTCaseAssignment(TCaseAssignment tCaseAssignment)
30     {
```

```
31          tCaseAssignments.add(tCaseAssignment);
32      }
33
34      public void addTCaseAssignment(List<TCaseAssignment> tCaseAssignments)
35      {
36          this.tCaseAssignments.addAll(tCaseAssignments);
37      }
38      //-------------------------------------------------------------------------------
39
40      public String getPreamble()
41      {
42          return preamble;
43      }
44
45      public String getEpilogue()
46      {
47          return epilogue;
48      }
49
50      public List<TCaseAssignment> getAssigments()
51      {
52          return tCaseAssignments;
53      }
54  }
```

Código B.1: Clase que implementa el módulo ConcreteTCase

TCRL_AST

```
1   /**
2    * CLASE que representa la raíz del ÁRBOL de SINTAXIS ABSTRACTA. Contiene
3    * el NOMBRE de la LEY de REFINAMIENTO, el texto del PRÉAMBULO, el texto del EPÍLOGO
4    * y las REGLAS de REFINAMIENTO contenidas en la LEY de REFINAMIENTO.
5    *
6    * @author  Diego Hollman
7    * @author  Pablo D. Coca
8    *
9    * @since   v1.0
10   *
11   * @version 2.0
12   */
13  class TCRL_AST{
14
15      private String      name;
16      private String      preamble;
17      private NodeRules   rules;
18      private String      epilogue;
19
20     /**
21      * CONSTRUCTOR de la clase TCRL_AST
22      *
23      * @param    lawName         NOMBRE de la LEY de REFINAMIENTO
24      * @param    preamble        PREÁMBULO de la LEY de REFINAMIENTO
25      * @param    rules           REGLAS de REFINAMIENTO contenidas en la LEY de REFINAMIENTO
26      * @param    epilogue        EPÍLOGO de la LEY de REFINAMIENTO
27      */
28      TCRL_AST(String lawName, String preamble, NodeRules rules, String epilogue) {
29
30          this.name        = lawName;
31          this.preamble    = preamble;
32          this.rules       = rules;
33          this.epilogue    = epilogue;
34
35      }
36
37     /**
38      * Devuelve  el NOMBRE de la LEY de REFINAMIENTO
```

```
39      *
40      * @return   el NOMBRE de la LEY de REFINAMIENTO
41      */
42      public String getName(){
43          return name;
44      }
45
46     /**
47      * Devuelve  el PREÁMBULO de la LEY de REFINAMIENTO
48      *
49      * @return   el PREÁMBULO de la LEY de REFINAMIENTO
50      */
51      public String getPreamble(){
52          return preamble;
53      }
54
55     /**
56      * Devuelve  el NOMBRE de la LEY de REFINAMIENTO
57      *
58      * @return   el EPÍLOGO de la LEY de REFINAMIENTO
59      */
60      public String getEpilogue(){
61          return epilogue;
62      }
63
64     /**
65      * Devuelve  las REGLAS de REFINAMIENTO contenidas en la LEY de REFINAMIENTO
66      *
67      * @return   las REGLAS de REFINAMIENTO contenidas en la LEY de REFINAMIENTO
68      */
69      public NodeRules getRules(){
70          return rules;
71      }
72  }
```

Código B.2: Clase que implementa el módulo TCRL_AST

NodeRules

```
1   /**
2    * Este nodo del ÁRBOL de SINTAXIS ABSTRACTA contiene el conjunto
3    * de todas las REGLAS de REFINAMIENTO, cada una como una instancia de la clase NodeRule.
4    *
5    * @see NodeRule
6    *
7    * @author  Diego Hollman
8    * @author  Pablo D. Coca
9    *
10   * @since   v1.0
11   *
12   * @version 2.0
13   */
14  class NodeRules{
15
16
17      // el 1er. ARGUMENTO del HashMap es el NOMBRE de la REGLA (RuleName)
18      private HashMap<String, NodeRule> rules = new HashMap<String, NodeRule>();
19
20      // Sirve para generar los NOMBRES a REGLAS que fueron agregadas
21      // sin NOMBRE.
22      // NOTA: El NOMBRE de la REGLA (RuleName) es OPCIONAL en TCRL
23      private Integer COUNTER = 0;
24
25     /**
26      * Agrega una REGLA de REFINAMIENTO
27      *
28      * @param    ruleName        NOMBRE de la REGLA de REFINAMIENTO [OPCIONAL]
```

```
29      * @param    rule            REGLA de REFINAMIENTO asociada al NOMBRE de la
30      *                           REGLA de REFINAMIENTO
31      *
32      * @see NodeRule
33      */
34      public void addRule(String ruleName, NodeRule rule){
35          rules.put(ruleName,rule);
36      }
37
38     /**
39      * Agrega una REGLA de REFINAMIENTO
40      *
41      * @param    rule            REGLA de REFINAMIENTO.
42      *                           Se asociada a un NOMBRE de
43      *                           REGLA de REFINAMIENTO generado automáticamente
44      *                           de la forma: RULE_NAME_GENERATED_#number,
45      *                           donde number es un NÚMERO generado automáticamente
46      *                           por la IMPLEMENTACIÓN de la CLASE NodeRules.
47      *
48      * @see NodeRule
49      */
50      public void addRule(NodeRule rule){
51
52          String RULE_NAME_GENERATED  = new String("RULE_NAME_GENERATED_#" + COUNTER.toString())
                ;
53
54          COUNTER = COUNTER + 1;
55
56          addRule(RULE_NAME_GENERATED, rule);
57      }
58
59     /**
60      * Obtiene la REGLA de REFINAMIENTO con NOMBRE ruleName
61      *
62      * @param    ruleName    NOMBRE de la REGLA de REFINAMIENTO
63      *
64      * @return   la REGLA de REFINAMIENTO asociada
65      */
66      public NodeRule getRule(String ruleName){
67          return rules.get(ruleName);
68      }
69
70     /**
71      * Obtiene una LISTA de los NOMBRES de todas las REGLAS de REFINAMIENTO
72      *
73      * @return   la LISTA de los NOMBRES de todas las REGLAS de REFINAMIENTO
74      */
75      public Set<String> getKeys(){
76          return rules.keySet();
77      }
78  }
```

Código B.3: Clase que implementa el módulo NodeRules

NodeRule

```
1   /**
2    * Representa a una REGLA (ya sea de SINÓNIMO o de REFINAMIENTO)
3    * para una VARIABLE de la ESPECIFICACIÓN FORMAL en Z.
4    *
5    * @author  Diego Hollman
6    * @author  Pablo D. Coca
7    *
8    * @since   v1.0
9    *
10   * @version 2.0
11   */
```

```
12 class NodeRule{
13
14 }
```

Código B.4: Clase que implementa el módulo NodeRule

RuleSynonym

```
1  /**
2   * Representa el TIPO de NODO en el ÁRBOL de SINTAXIS ABSTRACTA
3   * que almacena una REGLA de SINÓNIMO.
4   * Extiende a la CLASE NodeRule.
5   *
6   * @see NodeRule
7   *
8   * @author  Diego Hollman
9   * @author  Pablo D. Coca
10  *
11  * @since   v1.0
12  *
13  * @version 2.0
14  */
15 class RuleSynonym extends NodeRule{
16
17     private     String          name;
18     private     NodeType        nodeType;
19
20    /**
21     * CONSTRUCTOR de la CLASE
22     *
23     * @param    synonymName             NOMBRE de la REGLA de SINÓNIMO
24     * @param    nodeType                NODO que CONTIENE la REGLA de SINÓNIMO
25     *
26     * @see NodeType
27     */
28     RuleSynonym(String synonymName, NodeType nodeType){
29
30         this.name       = synonymName;
31         this.nodeType   = nodeType;     // type está declarada en NodeRule
32
33     }
34
35    /**
36     * Devuelve el NOMBRE que se le asigno a la REGLA de SINÓNIMO
37     *
38     * @return   el NOMBRE que se le asigno a la REGLA de SINÓNIMO
39     */
40     public String getName(){
41         return name;
42     }
43
44    /**
45     * Devuelve el TIPO de NODO asociado a la REGLA de SINÓNIMO
46     *
47     * @return   el TIPO de NODO asociado a la REGLA de SINÓNIMO
48     */
49     public NodeType getNodeType(){
50         return nodeType;
51     }
52 }
```

Código B.5: Clase que implementa el módulo RuleSynonym

RuleRefinement

```
/**
 * Representa el TIPO de NODO en el ÁRBOL de SINTAXIS ABSTRACTA
 * que almacena una REGLA de REFINAMIENTO.
 * Extiende a la CLASE NodeRule.
 *
 * @see NodeRule
 *
 * @author  Diego Hollman
 * @author  Pablo D. Coca
 *
 * @since   v1.0
 *
 * @version 2.0
 */
class RuleRefinement extends NodeRule{

    // VARIABLES de la ESPECIFICACIÓN
    private ArrayList<String> specIDs;

    // VARIABLES de la ESPECIFICACIÓN
    private HashMap<String, NodeType> implIDs;

    // VARIABLES de la ESPECIFICACIÓN asociadas a código JAVA
    private Refinement  refinement;


   /**
    * CONSTRUCTOR
    *
    * @param    specificationIDs        NOMBRES de la VARIABLESS en la ESPECIFICACIÓN
    * @param    implementationIDs       NOMBRE y TIPO de las VARIABLES en la IMPLEMENTACIÓN
    * @param    refinement
    *
    * @see NodeType
    */
    RuleRefinement(     ArrayList<String> specificationIDs,
                        HashMap<String, NodeType> implementationIDs,
                        Refinement refinement
                  )
    {
        this.specIDs            = specificationIDs;
        this.implIDs            = implementationIDs;
        this.refinement         = refinement;
    }

   /**
    * Devuelve los NOMBRES de la VARIABLES de la ESPECIFICACIÓN asociadas
    * a la REGLA de REFINAMIENTO
    *
    * @return   los NOMBRES de la VARIABLES de la ESPECIFICACIÓN asociadas
    *           a la REGLA de REFINAMIENTO
    */
    public ArrayList<String> getSpecIDs(){
        return specIDs;
    }

   /**
    * Devuelve los NOMBRES de las VARIABLES en la IMPLEMENTACIÓN asociadas
    * a la REGLA de REFINAMIENTO
    *
    * @return   los NOMBRES de las VARIABLES en la IMPLEMENTACIÓN asociadas
    *           a la REGLA de REFINAMIENTO
    */
    public Set<String> getImplIDs(){
        return implIDs.keySet();
    }

    public NodeType getNodeType(String implID){
```

```
69          return implIDs.get(implID);
70      }
71
72      public void setNodeType(String implID, NodeType nodeType){
73          implIDs.put(implID, nodeType);
74      }
75
76      public Abstraction getRefinement()
77      {
78          return refinement;
79      }
80
81  }
```

Código B.6: Clase que implementa el módulo RuleRefinement

NodePLType

```
1   /**
2    * Representa el TIPO de NODO en el ÁRBOL de SINTAXIS ABSTRACTA
3    * que almacena TIPOS PRIMITVOS del LENGUAJE de IMPLEMENTACIÓN (ej: int, boolean).
4    *
5    *
6    * @author  Pablo D. Coca
7    *
8    * @since   v2.0
9    *
10   * @version 2.0
11   */
12  class NodePLType extends NodeType{
13
14      private String type;
15
16      NodePLType(String type)
17      {
18          this.type = type;
19      }
20
21      public String getType()
22      {
23          return type;
24      }
25  }
```

Código B.7: Clase que implementa el módulo NodePLType

NodeStructure

```
1   /**
2    * Representa el TIPO de NODO en el ÁRBOL de SINTAXIS ABSTRACTA
3    * que almacena una VARIABLE de tipo ESTRUCTURA de TCRL.
4    *
5    *
6    * @author  Pablo D. Coca
7    *
8    * @since   v2.0
9    *
10   * @version 2.0
11   */
12  class NodeStructure extends NodeType{
13
14      private String name;        // Si es necesario el TESTER debe ingresar el NOMBRE
15                                  // de la CLASE precediéndola con el PAQUETE que la CONTIENE.
```

```
16                                  // EJEMPLO: java.lang.Integer
17
18      private List<NodeElement> elements;
19
20      // CONSTRUCTOR
21      NodeStructure(      String name,
22                          List<NodeElement> elements
23                    )
24      {
25          this.name       = name;
26          this.elements   = elements;
27      }
28
29      public String getName(){
30          return name;
31      }
32
33          public List<NodeElement> getElements(){
34          return elements;
35      }
36
37  }
```

Código B.8: Clase que implementa el módulo NodeStructure

Apéndice C

Especificaciones de los Casos de Estudio

A continuación se presentan las especificaciones completas de los casos de estudios expuestos en la sección 7.

C.1. Clases de Seguridad

Esta especificación formaliza el concepto de clase de seguridad. Una clase de seguridad es un par ordenado cuya primera componente es un número entero, llamado *nivel*, y la segunda es un conjunto de categorías. La especificación presenta un invariante de estado para cualquier clase de seguridad y una serie de operaciones que modifican o consultan una clase de seguridad.

Definición de Tipos Básicos, Parámetros y Estado

$CATEGORY$ todas las posibles categorías

scCatFull es devuelto cuando el tamaño del conjunto de categorías alcanza su máxima capacidad

scOk es devuelto cuando no hay errores en la invocación de alguna operación

scError es devuelto cuando un error no especificado previamente ocurre en la invocación de alguna operación

$[CATEGORY]$
$SCREPORT ::= scCatFull \mid scOk \mid scError$

$MAXLEVEL$ máximo valor posible de un nivel de seguridad

$MAXNCAT$ máximo tamaño de un conjunto de categorías

$\mid MAXLEVEL, MAXNCAT : \mathbb{N}$

Se modela una clase de seguridad como un esquema incluyendo a las variables de significado obvio.

SecClass

$$level : \mathbb{Z}$$
$$categs : \mathbb{P}\, CATEGORY$$

El invariante dice que el nivel (*level*) de una clase de seguridad (*SecClass*) debe pertenecer a un intervalo finito y que el tamaño de un conjunto de categorías debe ser menor o igual a *MAXNCAT*.

SCInv

$$SecClass$$

$$level \in 0 \,..\, MAXLEVEL$$
$$\#categs \leq MAXNCAT$$

En el estado inicial, una clase de seguridad se iguala a L, esto es la cota inferior del conjunto de clases de acceso.

SCInit

$$SecClass$$

$$level = 0$$
$$categs = \varnothing$$

Operaciones

SCGetSize retorna el número de categorías que hay dentro una clase de acceso dada.

SCGetSize

$$\Xi SecClass$$
$$size! : \mathbb{N}$$
$$rep! : SCREPORT$$

$$size! = \#categs$$
$$rep! = scOk$$

SCGetCat retorna una lista con las categorías de una clase de acceso dada.

SCGetCat

$$\Xi SecClass$$
$$lcategs! : \text{seq}\, CATEGORY$$
$$rep! : SCREPORT$$

$$\text{ran}\, lcategs! = categs$$
$$rep! = scOk$$

SCGetLevel retorna el nivel de una clase de acceso dada.

SCGetLevel

$\Xi SecClass$
$l! : \mathbb{Z}$
$rep! : SCREPORT$

$l! = level$
$rep! = scOk$

SCSetAddCat agrega una categoría al conjunto de categorías de una clase de acceso siempre y cuando la cantidad actual de categorías no iguale a $MAXNCAT$. La otra precondición ($c? \notin categs$) está para advertir al programador que la categoría por ser agregada no debe estar ya en el conjunto de categorías.

SCAddCatOk

$\Delta SecClass$
$c? : CATEGORY$
$rep! : SCREPORT$

$c? \notin categs$
$\#categs < MAXNCAT$
$categs' = categs \cup \{c?\}$
$level' = level$
$rep! = scOk$

Hay 2 posibles errores: cuando *categs* está lleno y cuando una categoría existente está por ser agregada.

*SCAddCatE*1

$\Xi SecClass$
$c? : CATEGORY$
$rep! : SCREPORT$

$c? \in categs$
$rep! = scError$

*SCAddCatE*2

$\Xi SecClass$
$rep! : SCREPORT$

$\#categs = MAXNCAT$
$rep! = scCatFull$

La operación total es resumida a continuación.

$SCAddCatE == SCAddCatE1 \lor SCAddCatE2$
$SCAddCat == SCAddCatOk \lor SCAddCatE$

SCSetLevel setea el nivel de una clase de acceso siempre y cuando el nivel de entrada se establezca en el intervalo apropiado.

SCSetLevelOk

$\Delta SecClass$
$l? : \mathbb{Z}$
$rep! : SCREPORT$

$0 \leq l? \leq MAXLEVEL$
$level' = l?$
$categs' = categs$
$rep! = scOk$

SCSetLevelE

$\Xi SecClass$
$l? : \mathbb{Z}$
$rep! : SCREPORT$

$l? < 0 \lor MAXLEVEL < l?$
$rep! = scError$

$SCSetLevel == SCSetLevelOk \lor SCSetLevelE$

La siguiente operación permite setear el nivel y el conjunto de una categoría en el mismo momento. Sus precondiciones son obvias si *SCAddCat* y *SCSetLevel* han sido leídas.

SCSetSCOk

$\Delta SecClass$
$l? : \mathbb{Z}$
$C? : \mathbb{P}\, CATEGORY$
$rep! : SCREPORT$

$0 \leq l? \leq MAXLEVEL$
$\#C? \leq MAXNCAT$
$level' = l?$
$categs' = C?$

$SCSetSCE1 == SCSetLevelE$

SCSetSCE2

$\Xi SecClass$
$C? : \mathbb{P}\, CATEGORY$
$rep! : SCREPORT$

$\#C? > MAXNCAT$
$rep! = scError$

$SCSetSCE == SCSetSCE1 \lor SCSetSCE2$
$SCSetSC == SCSetSCOk \lor SCSetSCE$

La interfaz de clases de seguridad es resumida a continuación.

$$
\begin{aligned}
&SCInterface == \\
&\quad SCGetLevel \\
&\lor\ SCGetSize \\
&\lor\ SCGetCat \\
&\lor\ SCSetLevel \\
&\lor\ SCAddCat \\
&\lor\ SCSetSC
\end{aligned}
$$

C.2. Protocolo de Comunicación EOCP para el Microsatélite Científico SACI-1[12]

Modelo formal del EXP-OBDH Communication Protocol (EOCP) para el microsatélite científico SACI-1, el cual fue escrito a partir de su especificación informal.

A la hora de realizar la lectura del modelo, es importante notar lo siguiente:

1. De todos los componentes del protocolo, sólo uno, cuyo nombre es SLAVE, ha sido especificado. Esto se debe a que SLAVE es probablemente el módulo más complejo del sistema.

2. Dado que es difícil, sino imposible, especificar requisitos de tiempo en el lenguaje Z, y dado que la especificación informal de EOCP incluye tales requisitos, el modelo formal aquí presentado no es lo suficientemente preciso como debería ser. Sin embargo, en este caso, los requisitos de tiempo son muy simples.

3. Las prioridades son también difíciles de especificar en Z y EOCP requiere que el componente EXP asigne más prioridad para recibir interrupciones para adquirir datos que para reaccionar a comandos enviados por el componente OBDH. No se ha formalizado este requerimiento.

4. Dado que Fastest (ver sección 3) todavía no implementa completamente el lenguaje Z, la especificación ha sido forzada para utilizar sólo el subconjunto del lenguaje implementado en la herramienta. Esto puede conducir a una especificación innecesariamente compleja o ilegible.

5. El protocolo formalizado es principalmente un sistema reactivo, por lo que Z no es el lenguaje de especificación formal más apropiado.

6. Se asumirá que todas las operaciones son atómicas.

7. No se ha modelado la verificación de la integridad de un mensaje.

[1] `http://www.globalsecurity.org/space/world/brazil/saci.htm`
[2] `http://www2.dem.inpe.br/ijar/4spaperF.html`

Tipos básicos y tipos libres

EXP captura información de su entorno, la cual, posiblemente, deba ser enviada hacia OBDH. Además, EXP almacena su propio estado en memoria. El siguiente tipo básico representa ambas clases de información:

$[MDATA]$

OBDH puede enviar distintos comandos a EXP para que éste realice determinada acción. Estos comandos se representan a través del tipo enumerado que sigue:

$$CTYPE ::= RM \mid SC \mid IDA \mid SDA \mid TD \mid RC \mid MD \mid ML \mid LP$$

Los diferentes tipos de respuestas que envía EXP también se representan con un tipo enumerado:

$$RTYPE ::= MR \mid CD \mid EDP \mid ND \mid MDD$$

Los modos de configuración en los cuales EXP puede trabajar son representados como otro tipo enumerado.

$$CMODE ::= COF \mid CON$$

OBDH puede solicitarle a EXP actuar sobre sus diferentes dispositivos a través de un comando llamado LoadParam (Cargar Parámetros) con un valor particular en el campo DATA. El tipo $PARAM$ se refiere a esos valores.

$$PARAM ::= LLDHiOff \mid LLDHiOn \mid TMOff \mid TMOn \mid HiV50 \mid HiV45 \mid HiV40 \mid HiVOff \mid RU \mid TS18 \mid TS13$$

En este modelo el tiempo se considera discreto. Teniendo en cuenta que el menor requisito de tiempo está dado en milisegundos, se asumirá que cada unidad de tiempo representa un milisegundo.

$$TIME == \mathbb{N}$$

BIN representa un tipo binario[3].

$$BIN ::= no \mid yes$$

Espacio de estado del sistema, estados iniciales e invariantes de estado

Las variables de estado del modelo se listan a continuación:

$ctime \approx$ El tiempo actual en EXP

$memd \approx$ La memoria en que EXP almacena registros de los experimentos realizados.

[3] `El lenguaje Z no trae definido un tipo para la representación de tipo binario.`

memp ≈ La memoria en que EXP almacena la imagen del programa y sus variables locales.

Se asume que EXP tiene espacio para 6 buffers de 43 elementos de tipo *MDATA*. Los primeros 5 buffers se reservan para el código del programa, sus variables, su pila, etc (*memp*), y el buffer restante es para información vinculada a experimentación (*memd*).

mdp ≈ Puntero de envío de memoria.

Es utilizado para mantener una indicación de cuántas celdas de memoria han sido enviadas al OBDH en respuesta a un comando de envío de memoria.

ped ≈ Puntero de memoria de datos de experimentación.

Es usada para mantener una indicación de cuántas celdas de memoria, para registros de experimentación, han sido enviadas al OBDH en respuesta a un comando de transmisión de datos.

mep ≈ Puntero que indica la última celda usada en la memoria para registros de experimentación.

Es usado para mantener una indicación de cuántas celdas de memoria están siendo usadas para almacenar registros de experimentación.

acquiring ≈ Si EXP está en modo de adquisición o no.

waiting ≈ Si EXP está esperando por el resto de un comando OBDH.

mode ≈ Modo de configuración en el que EXP está trabajando actualmente.

ccmd ≈ El próximo comando a ser procesado por EXP.

sending ≈ Si EXP está enviando datos a OBDH o no.

dumping ≈ Si EXP está enviando su memoria o no.

waitsignal ≈ Si EXP está esperando para enviar una señal al telescopio, que le indique que debe adquirir nuevos registros de experimentación.

Se han agrupado éstas variables de estado en tres esquemas Z: el primero, agrupa variables relacionadas a la memoria; el segundo, variables vinculadas al tiempo; y el tercero, todas las variables restantes.

Memory

$$memp, memd : \text{seq}\, MDATA$$
$$mdp, mep, ped : \mathbb{N}$$

$$Time == [ctime : TIME]$$

Status

$acquiring, waiting, sending, dumping, waitsignal : BIN$
$mode : CMODE$
$ccmd : CTYPE$

El **esquema de estado** es la inclusión de los tres esquemas previos.

$ExpState == [Memory;\ Time;\ Status]$

El **esquema de estado inicial**, el cual representa un conjunto infinito de estados, establece valores razonables para algunas de las variables de estado.

ExpInitState

$ExpState$

$\#memp = 5 * 43$
$\#memd = 43$
$mdp = mep = ped = 0$
$acquiring = waiting = sending = dumping = waitsignal = no$

Los **invariantes de estado** del sistema se describen en el siguiente esquema:

StateInvariants

$ExpState$

$\#memp = 5 * 43$
$\#memd = 43$
$mep \in 0 \,..\, 43$
$ped \in 0 \,..\, 43$
$mdp \in 0 \,..\, 5 * 43$
$acquiring = no \Rightarrow waitsignal = no$

Operaciones

Por simplicidad se asumirá que, desde un punto de vista abstracto, los comandos se envían desde OBDH a EXP en cuatro etapas (sin importar cuántos paquetes de datos de bajo nivel deban ser enviados):

1. OBDH inicia un comando.
2. OBDH envía el tipo de comando.
3. OBDH envía los parámetros (de ser necesario) de acuerdo al tipo de comando.
4. OBDH termina el comando.

EXP sincronizará con OBDH de la siguiente manera:

1. EXPR esperará a OBDH para iniciar un comando.

2. EXP esperará por un tipo de comando.
3. EXP esperará por el fin del comando.
4. EXP ejecutará una entre varias operaciones internas, dependiendo del tipo de comando recibido en el segundo paso.

   Notar que no se está modelando la recepción de parámetros: simplemente se asume que esto ocurre de algún modo.

Las operaciones correspondientes a los primeros tres pasos se describen en las sección C.2 y las operaciones internas (cuarto paso) son descriptas en las secciones C.2 y C.2. Sin embargo, la siguiente sección (C.2) muestra algunas operaciones relacionadas a aspectos temporales, dado que algunas de las otras operaciones tienen requisitos de tiempo.

EXP también tiene que sincronizar con los sensores del satélite para controlar la adquisición de datos. Esto se muestra en la sección C.2.

Debe notarse que todas las operaciones, salvo cuando se diga explícitamente lo contrario, constituyen la interfaz que EXP expone a ODBH y otros componentes tales como los sensores y el reloj.

Operaciones relacionadas a aspectos temporales

Tick es una operación externa llevada a cabo por el hardware de EXP y representa el avance del reloj en una unidad de tiempo. De alguna forma, EXP recibe este nuevo valor, o el hardware modifica una de las celdas de memoria de EXP. Esto es irrelevante en este nivel de abstracción.

$$Tick == [\Delta ExpState;\ \Xi Memory;\ \Xi Status \mid ctime' = ctime + 1]$$

Cuando EXP recibe de OBDH el comienzo de un nuevo comando, se establece un timer que expirará despues de 500 ms. $Timeout500ms$ representa esa expiración (es decir, el timer (externo) llamará a $Timeout500ms$ cuando termine). después de la ejecución de esta operación EXP no esperará más por el resto del mensaje. $Set500msTimer$ y $Stop500Timer$ constituyen la interfaz que EXP debe usar par establecer y detener el timer, y por ahora se las deja sub-especificadas.

*Timeout*500*ms*

$\Delta ExpState;\ \Xi Memory;\ \Xi Time$

$waiting' = no$
$acquiring' = acquiring$
$sending' = sending$
$dumping' = dumping$
$waitsignal' = waitsignal$
$mode' = mode$
$ccmd' = ccmd$

*Set*500*msTimer*

$x : \mathbb{Z}$

*Stop*500*msTimer*

$x : \mathbb{Z}$

El otro requisito de tiempo para EXP tiene que ver con la recolección de registros experimentales. EXP debe emitir una interrupción hacia el hardware del telescopio cada 700 ms. Se procederá de forma análoga a lo anterior:

*Timeout*700*ms*

$\Delta ExpState;\ \Xi Memory;\ \Xi Time$

$waitsignal' = no$
$sending' = sending$
$waiting' = waiting$
$acquiring' = acquiring$
$dumping' = dumping$
$mode' = mode$
$ccmd' = ccmd$

*Set*700*msTimer*

$x : \mathbb{Z}$

*Stop*700*msTimer*

$x : \mathbb{Z}$

Inicio de comando, tipo de comando y fin de comando

El documento de requerimientos dice que EXP puede detectar el comienzo de un nuevo comando, momento a partir del cual debe esperar 500 ms. por el resto del comando. Si este no llega, EXP esperará por otro nuevo comando.

Las siguientes operaciones son ofrecidas por EXP para que OBDH puede ejecutarlas para transmitir sus comandos:

CommandStartOk

$\Delta ExpState;\ \Xi Memory;\ \Xi Time$

$waiting = no$
$waiting' = yes$
$acquiring' = acquiring$
$sending' = sending$
$dumping' = dumping$
$waitsignal' = waitsignal$
$mode' = mode$
$ccmd' = ccmd$

$$CommandStartE == [\Xi ExpState \mid waiting = yes]$$
$$CommandStart == (CommandStartOk \land Set500msTimer) \lor CommandStartE$$

Cuando EXP está esperando, OBDH puede establecer el tipo del comando que quiere ejecutar en EXP. Aquí se tienen tres porque hay dos comandos que envían sus respuestas en más de un paso. Estos comandos son: transmitir datos (TD) y enviar memoria (MD).

*CommandTypeOk*1

$\Delta ExpState;\ \Xi Memory;\ \Xi Time$
$type? : CTYPE$

$waiting = yes$
$type? \notin \{TD, MD\}$
$ccmd' = type?$
$waiting' = waiting$
$acquiring' = acquiring$
$sending' = sending$
$dumping' = dumping$
$waitsignal' = waitsignal$
$mode' = mode$

*CommandTypeOk*2

$\Delta ExpState;\ \Xi Memory;\ \Xi Time$
$type? : CTYPE$

$waiting = yes$
$type? = TD$
$ccmd' = type?$
$sending' = yes$
$acquiring' = acquiring$
$waiting' = waiting$
$dumping' = dumping$
$waitsignal' = waitsignal$
$mode' = mode$

*CommandTypeOk*3

$\Delta ExpState;\ \Xi Memory;\ \Xi Time$
$type? : CTYPE$

$waiting = yes$
$type? = MD$
$ccmd' = type?$
$dumping' = yes$
$sending' = sending$
$acquiring' = acquiring$
$waiting' = waiting$
$waitsignal' = waitsignal$
$mode' = mode$

$$CommandTypeE == [\Xi ExpState \mid waiting \neq yes]$$
$$CommandType == CommandTypeOk1 \lor CommandTypeOk2 \lor CommandTypeOk3 \lor CommandTypeE$$

De alguna forma se asume que OBDH comunica a EXP que el comando actual ha finalizado, pero EXP prestará atención a esta señal si está esperando por ella ($waiting = yes$).

*CommandFinishOk*1

$\Delta ExpState;\ \Xi Memory;\ \Xi Time$

$waiting = yes$
$waiting' = no$
$sending' = sending$
$acquiring' = acquiring$
$dumping' = dumping$
$waitsignal' = waitsignal$
$mode' = mode$
$ccmd' = ccmd$

$$CommandFinishE == [\Xi ExpState \mid waiting \neq yes]$$
$$CommandFinish == (CommandFinishOk1 \land Stop500msTimer) \lor CommandFinishE$$

Notar que tanto *CommandFinishOk* como $Timeout500ms$ pueden cambiar el valor de *waiting*. Precisamente, el comportamiento de EXP con respecto a si el resto de un nuevo comando es aceptado, depende de cuáles de esas operaciones se ejecuta primero.

Una vez que OBDH completa el comando, EXP puede ejecutar el cuerpo de la correspondiente operación interna (mostrado en las secciones C.2 y C.2).

Adquisición de datos

Una vez que OBDH le ordena a EXP la habilitación de la interrupción para adquisición de datos, EXP enviará la interrupción al telescopio cada 700 ms. Después que la interrupción fue

enviada, EXP es interrumpido por el telescopio cuando haya información para ser consultada. Esta información es almacenada en un buffer de 43 celdas. *RetrieveEData* es, entonces, la operación de EXP ejecutada por la interrupción enviada por el telescopio.

RetrieveEDataOk

$$\Delta ExpState;\ \Xi Time;\ \Xi Status$$
$$data? : \text{seq}\ MDATA$$

$$data? \neq \langle\rangle$$
$$mep + \#data? \leq 43$$
$$memd' = memd \oplus \{i : 1 \mathinner{..} \#data? \bullet mep + i \mapsto data?\ i\}$$
$$mep' = mep + \#data? + 1$$
$$ped' = 0$$
$$mdp' = mdp$$
$$memp' = memp$$

$$RetrieveEDataE == [\Xi ExpState;\ data? : \text{seq}\ MDATA \mid data? = \langle\rangle \lor 43 < mep + \#data?]$$
$$RetrieveEData == RetrieveEDataOk \lor RetrieveEDataE$$

No se ha incluido $waiting = no \land acquiring = yes$ en la precondición porque $acquiring = no$ significa que al telescopio no se le solicita realizar una adquisición de datos, y esto implica que el telescopio no enviará la interrupción a EXP.

Comandos OBDH simples

En esta sección se mostrará la especificación de aquellos comandos que necesitan respuestas de un paso para finalizar.

El comando ResetMicro (resetear el microcontrolador) no cambia el estado de EXP porque el microcontrolador es considerado un componente externo. Entonces, se puede sub-especificar una interfaz que esta operación puede llamar para reiniciar el microcontrolador. Posiblemente, esta operación externa puede reiniciar, por ejemplo, la memoria del sistema, pero esto no se encuentra en el documento de requerimientos.

MicroReset

$$x : \mathbb{Z}$$

ResetMicroOk

$$\Xi ExpState$$
$$rsp! : RTYPE$$

$$waiting = no$$
$$ccmd = RM$$
$$rsp! = MR$$

$$ResetMicroE == [\Xi ExpState \mid waiting \neq no \lor ccmd \neq RM]$$
$$ResetMicro == (ResetMicroOk \land MicroReset) \lor ResetMicroE$$

El comando que consulta el tiempo actual envía el valor actual de *ctime*.

SendClockOk

$\Xi ExpState$
$rsp! : RTYPE$
$rdata! : TIME$

$waiting = no$
$ccmd = SC$
$rsp! = CD$
$rdata! = ctime$

$SendClockE == [\Xi ExpState \mid waiting \neq no \lor ccmd \neq SC]$
$SendClock == SendClockOk \lor SendClockE$

La adquisición de datos se lleva a cabo en dos pasos: primero, OBDH envía a EXP el comando apropiado, y segundo, una operación interna de EXP se ejecuta cada 700 ms. Esta operación interna envía una interrupción al telescopio para ordenar la adquisición de datos. El primer paso se formaliza en el esquema *InitDataAcq* y el segundo en el esquema *InterruptTele*.

InitDataAcqOk

$\Delta ExpState;\ \Xi Memory;\ \Xi Time$
$rsp! : RTYPE$

$waiting = no$
$ccmd = IDA$
$acquiring = no$
$acquiring' = waitsignal' = yes$
$rsp! = MR$
$waiting' = waiting$
$sending' = sending$
$dumping' = dumping$
$mode' = mode$
$ccmd' = ccmd$

$InitDataAcqE == [\Xi ExpState \mid waiting \neq no \lor ccmd \neq IDA \lor acquiring \neq no]$
$InitDataAcq == (InitDataAcqOk \land Set700msTimer) \lor InitDataAcqE$

Cuando el timer de 700 ms. expira, la operación interna *InterruptTele* es habilitada. *InterruptTele* se deshabilita a sí misma después de ejecutarse, pero establece nuevamente el timer de 700 ms. para ser iniciada hasta que OBDH le indique a EXP que no debe adquirir más datos.

InterruptTeleOk

$\Delta ExpState;\ \Xi Memory;\ \Xi Time$
$signal! : \mathbb{N}$

$waiting = no$
$acquiring = yes$
$waitsignal = no$
$waitsignal' = yes$
$signal! = 1$
$waiting' = waiting$
$sending' = sending$
$dumping' = dumping$
$acquiring' = acquiring$
$mode' = mode$
$ccmd' = ccmd$

$InterruptTele == InterruptTeleOk \land Set700msTimer$

Cuando OBDH le solicita a EXP detener la adquisición de datos, simplemente cambia el valor de la variable *acquiring* de *yes* a *no*. Notar que esto implica que *InterruptTele* no será habilitada otra vez.

StopDataAcqOk

$\Delta ExpState;\ \Xi Memory;\ \Xi Time$
$rsp! : RTYPE$

$waiting = no$
$ccmd = SDA$
$acquiring = yes$
$acquiring' = waitsignal' = no$
$rsp! = MR$
$waiting' = waiting$
$sending' = sending$
$dumping' = dumping$
$mode' = mode$
$ccmd' = ccmd$

$StopDataAcqE == [\Xi ExpState \mid waiting \neq no \lor ccmd \neq SDA \lor acquiring \neq yes]$
$StopDataAcq == StopDataAcqOk \lor StopDataAcqE$

La reconfiguración es fácil de modelar porque sólo se tiene que cambiar el valor de la variable *mode*, como sigue:

ReconfigOk

$\Delta ExpState;\ \Xi Memory;\ \Xi Time$
$nmode? : CMODE$
$rsp! : RTYPE$

$waiting = no$
$ccmd = RC$
$mode' = nmode?$
$rsp! = MR$
$acquiring' = acquiring$
$sending' = sending$
$waiting' = waiting$
$dumping' = dumping$
$waitsignal' = waitsignal$
$ccmd' = ccmd$

$$ReconfigE == [\Xi ExpState \mid waiting \neq no \lor ccmd \neq RC]$$
$$Reconfig == ReconfigOk \lor ReconfigE$$

Cuando EXP recibe el comando de envío de memoria, se prepara para enviarla y devuelve el mensaje RECEIVED a OBDH. El envío real se produce cuando OBDH envía un comando de transmisión de datos.

MemoryDumpOk

$\Xi ExpState$
$rsp! : RTYPE$

$waiting = no$
$ccmd = MD$
$rsp! = MR$

$$MemoryDumpE == [\Xi ExpState \mid waiting \neq no \lor ccmd \neq MD]$$
$$MemoryDump == MemoryDumpOk \lor MemoryDumpE$$

Cargar la memoria con datos provenientes de OBDH es más complejo que las operaciones previas, porque es necesario formalizar cómo los 43 bytes dedicados a registros de experimentación pueden ser modificando evitando desbordamientos, dado que ellos pueden dar lugar a errores fatales de programa.

*MemoryLoadOk*1

$\Delta ExpState;\ \Xi Time;\ \Xi Status$
$addr? : \mathbb{N}$
$data? : \text{seq}\ MDATA$
$rsp! : RTYPE$

$waiting = no$
$ccmd = ML$
$0 < addr?$
$data? \neq \langle\rangle$
$addr? + \#data? \leq 43$
$addr? + \#data? \leq mep$
$memd' = memd \oplus \{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\}$
$rsp! = MR$
$mdp' = mdp$
$mep' = mep$
$ped' = ped$
$memp' = memp$

*MemoryLoadOk*2

$\Delta ExpState;\ \Xi Time;\ \Xi Status$
$addr? : \mathbb{N}$
$data? : \text{seq}\ MDATA$
$rsp! : RTYPE$

$waiting = no$
$ccmd = ML$
$0 < addr?$
$data? \neq \langle\rangle$
$addr? + \#data? \leq 43$
$mep < addr? + \#data?$
$memd' = memd \oplus \{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\}$
$mep' = addr? + \#data?$
$rsp! = MR$
$mdp' = mdp$
$ped' = ped$
$memp' = memp$

$MemoryLoadE1 == [\Xi ExpState;\ addr? : \mathbb{N};\ data? : \text{seq}\ MDATA \mid waiting \neq no \lor ccmd \neq ML]$

$MemoryLoadE2 == [\Xi ExpState;\ addr? : \mathbb{N};\ data? : \text{seq}\ MDATA;\ low? : BIN \mid$
$addr? = 0 \lor data? = \langle\rangle \lor 43 < addr? + \#data?]$

$MemoryLoad == MemoryLoadOk1 \lor MemoryLoadOk2 \lor MemoryLoadE1 \lor MemoryLoadE2$

Cuando un comando de carga de parámetros es recibido, EXP simplemente chequea si el parámetro es posible, y si lo es, EXP lo envía al dispositivo apropiado. La validación de parámetros es modelada por la espera de un valor de tipo *PARAM*. El envío del valor al hardware se representa con una variable de salida.

LoadParamOk

$\Xi ExpState$
$p?, actuate! : PARAM$
$rsp! : RTYPE$

$waiting = no$
$ccmd = LP$
$actuate! = p?$
$rsp! = MR$

$LoadParamE == [\Xi ExpState \mid waiting \neq no \land ccmd \neq LP]$
$LoadParam == LoadParamOk \lor LoadParamE$

Transmisión de datos

La transmisión de datos es la operación más compleja del sistema porque involucra el envío de diferentes partes de la memoria, afectándola a ella misma de diversas maneras. Más aún, la transmisión de datos posiblemente necesite más de una respuesta para finalizar.

La operación es invocada para transmitir registros de experimentación y el resultado es un envío de memoria. Los primeros dos esquemas que se presentan a continuación describen la comunicación de datos de experimentación y los últimos dos la transmisión después de un envío de memoria.

Una vez que OBDH solicita registros de experimentación, EXP envía un paquete de 43 bytes siempre que el buffer está completo. Si el buffer no está completo, EXP responderá con un mensaje NO DATA.
Enviar la memoria es similar a transmitir registros de experimentación, pero:

- Se ha interpretado que un envío de memoria no libera la memoria.
- EXP envía un paquete de 43 bytes como respuesta a cada comando de transmisión de datos enviado por OBDH, hasta que todos los *memp* han sido transmitidos.

*TransDataOk*1

$\Delta ExpState;\ \Xi Time$
$rsp! : RTYPE$
$rdata! : \text{seq}\ MDATA$

$waiting = no$
$ccmd = TD$
$sending = yes$
$dumping = no$
$ped = 0$
$mep = 43$
$rsp! = EDP$
$rdata! = memd$
$ped' = 43$
$sending' = no$
$acquiring' = acquiring$
$waiting' = waiting$
$dumping' = dumping$
$waitsignal' = waitsignal$
$ccmd' = ccmd$
$mode' = mode$
$memp' = memp$
$memd' = memd$
$mdp' = mdp$
$mep' = mep$

*TransDataE*1

$\Xi ExpState$
$rsp! : RTYPE$

$waiting = no$
$ccmd = TD$
$sending = yes$
$dumping = no$
$mep < 43 \lor ped = 43$
$rsp! = ND$

*TransDataOk*2

$\Delta ExpState;\ \Xi Time;\ \Xi Status$
$rsp! : RTYPE$
$rdata! : \text{seq}\ MDATA$

$waiting = no$
$ccmd = TD$
$sending = dumping = yes$
$mdp < 5 * 43$
$rsp! = MDD$
$rdata! = (mdp + 1\,..\,mdp + 43) \lhd memp$
$mdp' = mdp + 43$
$memp' = memp$
$memd' = memd$
$mep' = mep$
$ped' = ped$

*TransDataOk*3

$\Delta ExpState;\ \Xi Time$
$rsp! : RTYPE$

$waiting = no$
$ccmd = TD$
$sending = dumping = yes$
$mdp = 5 * 43$
$rsp! = ND$
$mdp' = 0$
$sending' = dumping' = no$
$memp' = memp$
$memd' = memd$
$mep' = mep$
$ped' = ped$
$acquiring' = acquiring$
$waiting' = waiting$
$sending' = sending$
$waitsignal' = waitsignal$
$mode' = mode$
$ccmd' = ccmd$

$$TransDataE2 == [\Xi ExpState \mid waiting \neq no \lor ccmd \neq TD \lor sending \neq yes]$$

$$\begin{aligned} TransData == \\ & TransDataOk1 \\ \lor\ & TransDataOk2 \\ \lor\ & TransDataOk3 \\ \lor\ & TransDataE1 \\ \lor\ & TransDataE2 \end{aligned}$$

Apéndice D

Interfaces de los módulos

Para documentar las interfaces de los módulos que conforman el sistema, se utilizó el lenguaje llamado **2MIL**. A continuación se describirá el significado de las palabras reservadas de dicho lenguaje.

- **Module**: indica el nombre del módulo. Puede ser una cadena de caracteres cualquiera. Este nombre se puede usar referencialmente.
- **imports**: hace accesible al módulo donde se cita los servicios de todo los módulos que se listan a continuación.
- **exportsproc**: son las signaturas de las subrutinas de la interfaz del módulo. La signatura de una subrutina, también llamada prototipo o encabezado, está constituida por el nombre, los parámetros y los valores de retorno de la misma. Cada subrutina se lista en una línea separada.
- **i**: antes de un parámetro de una signatura significa que dicho parámetro es de *entrada*.
- **comments**: texto libre para comentar aspectos del módulo o su interfaz que se consideran complicados o necesarios de aclarar.

Para más detalles sobre el lenguaje **2MIL**, consultar [9] y [7].

Module	**TCaseRefinement**
comprises	DataStructures, FunctionalModules

Module	**DataStructures**
comprises	AbstractTCase, ConcreteTCase, TCaseAssignment, TCRL_File, AST

Module	**AbstractTCase**
imports	AxPara
exportsproc	getMyAxPara(): AxPara
comments	AxPara es un módulo del framework **CZT** [21] (http://czt.sourceforge.net/). Ver también sección 6.1.3.

Module	**ConcreteTCase**
imports	TCaseAssignment
exportsproc	setPreamble(**i** String)
	setEpilogue(**i** String)
	addTCaseAssignment(**i** TCaseAssignment)
	getPreable(): String
	getEpilogue(): String
	getTCaseAssignments(): List(TCaseAssignment)

Module	**TCaseAssignment**
exportsproc	TCaseAssignment(**i** String, **i** String)
	setSpecName(**i** String)
	setRefText(**i** String)
	getSpecName(): String
	getRefText(): String

Module	**TCRL_File**
exportsproc	TCRL_File(**i** String, **i** char)
	read(**i** char): String
	write(**i** String, **i** char)

Module	**AST**
comprises	TCRL_AST, NodeRules, NodeRule, RuleSynonym, RuleRefinement, NodeTypes

Module	**TCRL_AST**
imports	NodeRules
exportsproc	TCRL_AST(**i** String, **i** String, **i** NodeRules, **i** String)
	getName(): String
	getPreamble(): String
	getRules(): NodeRules
	getEpilogue(): String

Module	**NodeRules**
imports	NodeRule
exportsproc	addRule(**i** String, **i** NodeRule)
	getRule(**i** String): NodeRule
	getKeys(): Set(String)

Module	**NodeRule**
imports	NodeType
exportsproc	getNodeType():NodeType

Module	**RuleSynonym inherits from** NodeRule
exportsproc	RuleSynonym(**i** String, **i** NodeType) getName(): String

Module	**RuleRefinement inherits from** NodeRule
imports	NodeType, RefinementOptions
exportsproc	RuleRefinement(**i** List(String), **i** List(String, NodeType), **i** RefinementOptions) getSpecIDs(): List(String) getImplIDs(): List(String) getNodeType(**i** String): List(NodeType) getRefinementOptions(): RefinementOptions
comments	RefinementOptions es un módulo que oculta las opciones complejas de refinamiento que puede tener una regla de refinamiento de TCRL v2.0. Ver sección 5.3.4

Module	**RefinementOptions**

Module	**NodeTypes**
comprises	NodeType, NodeSynonym, NodePLType, NodePointer, NodeArray, NodeStructure, NodeEnumeration, NodeList, NodeFile, NodeDB

Module	**NodeType**

Module	**NodeSynonym inherits from** NodeType
exportsproc	NodeSynonym(**i** String) getID(): String

Module	**NodePLType inherits from** NodeType
exportsproc	NodePLType(**i** String) getType(): String

Module	**NodePointer inherits from** NodeType
exportsproc	NodePointer(**i** NodeType)
	getType(): NodeType

Module	**NodeArray inherits from** NodeType
exportsproc	NodeArray(**i** NodeType, **i** int)
	getType(): NodeType
	getSize(): int

Module	**NodeElement inherits from** NodeType
exportsproc	NodeElement(**i** String, **i** NodeType, **i** String)
	getID(): String
	getType(): NodeType
	getConstant(): String

Module	**NodeStructure inherits from** NodeType
imports	NodeElement
exportsproc	NodeStructure(**i** String, **i** List(NodeElement))
	getName(): String
	getElements(): List(NodeElement)

Module	**NodeEnumeration inherits from** NodeType
imports	NodeElement
exportsproc	NodeEnumeration(**i** NodeType, **i** Map(String, String))
	getType(): NodeType
	getElements(): Map(String, String)
	getConstant(**i** String): String

Module	**NodeList inherits from** NodeType
imports	NodeElement
exportsproc	NodeList(**i** String, **i** String, **i** String, **i** String, **i** List(NodeElements), **i** String)
	getName(): String
	getLinkType(): String
	getLinkNextName(): String
	getLinkPrevName(): String
	getFields(): List(NodeElement)
	getMemalloc(): String

Module	**NodeFile inherits from** NodeType
exportsproc	NodeList(**i** String, **i** String, **i** String, **i** String, **i** String, **i** String) getName(): String getPath(): String getDelimiter(): String getEol(): String getEof(): String getStructure(): String

Module	**NodeDBColumn inherits from** NodeType
exportsproc	NodeDBColumn(**i** String, **i** String) getColName(): String getColType(): String

Module	**NodeDB inherits from** NodeType
imports	NodeDBColumn
exportsproc	NodeList(**i** String, **i** String, **i** String, **i** List(NodeDBColumn)) getDBMSID(): String getConnectionID(): String getTableName(): String getColumns(): List(NodeDBColumn)

Module	**FunctionalModules**
comprises	TCRL_Parser, TCRL_PreProcessor, TCaseRefClient, TCaseRefClientRunner, RefineAST, RefineExpr, ConstantGenerator

Module	**TCRL_Parser**
imports	TCRL_File, TCRL_AST
exportsproc	parse(**i** TCRL_File): TCRL_AST

Module	**TCRL_PreProcessor**
imports	TCRL_AST
exportsproc	preProcess(**i** TCRL_AST):TCRL_AST

Module	**TCaseRefClient**
imports	AbstractTCase, ConcreteTCase, TCRL_File, AbstractTCase, TCRL_AST, Event_
exportsproc	TCaseRefClient(**i** TCRL_File, **i** AbstractTCase)
comments	Event_ es un módulo de FASTest [10].

Module	**TCaseRefClientRunner**
imports	AbstractTCase, ConcreteTCase, TCRL_File, TClass, AbstractTCase, TCRL_AST, Event_
exportsproc	TCaseRefClientRunner(**i** String, **i** TCRL_File, **i** TClass, **i** AbstractTCase)
comments	TClass y Event_ son módulos de FASTest [10].

Module	**RefineAST**
imports	TCRL_AST, TCRL_Parser, AbstractTCase, ConcreteTCase
exportsproc	refine(**i** TCRL_AST, **i** AbstractTCase): ConcreteTCase

Module	**RefineExpr**
imports	RuleRefinement, ZExprList, TCaseAssignmnet
exportsproc	refineExprList(**i** RuleRefinement, **i** ZExprList): List(TCaseAssignmnet)
comments	ZExprList es un módulo del framework **CZT** [21] (http://czt.sourceforge.net/). Ver también sección 6.1.3.

Module	**ConstantGenerator**
imports	Expr
exportsproc	PLTypeToString(**i** String, **i** Expr): String
comments	Expr es un módulo del framework **CZT** [21] (http://czt.sourceforge.net/). Ver también sección 6.1.3.

Apéndice E

Guía de módulos

La **Guía de módulos** es un documento escrito en lenguaje natural que complementa al documento **Interfaz de los módulos** (ver sección D). El objetivo de este documento es permitir que los diseñadores, programadores y mantenedores del sistema puedan identificar partes del software que ellos necesitan entender sin tener que leer detalles irrelevantes sobre otras partes.

1. **DataStructures**

   Este es un módulo lógico que agrupa a todos los módulos que representan estructuras de datos utilizadas para guardar información inherente al proceso de refinamiento.

   1.1. **AbstractTCase**

   Módulo físico que representa la estructura de dato de un **caso de prueba abstracto** generado por FASTest (ver sección 3 y la bibliografía [10]). Incluye un a un *AxPara* el cual representa un esquema en el lenguaje de especificación Z (ver sección 2.2).
   No se incluye a *AxPara* como un módulo 2MIL debido a que se encuentra definido dentro del framework CZT (ver sección 6.1.3 y la bibliografía [21]).

   1.2. **ConcreteTCase**

   Módulo físico que representa la estructura de dato de un **caso de prueba concreto** generado por el sistema de refinamiento. El mismo incluye varias instancias de TCaseAssignment.

   1.3. **TCaseAssignment**

   Módulo físico que representa la asignación de una variable de la implementación del sistema a refinar con su respectivo texto de refinamiento escrito en el lenguaje de programación deseado para el refinamiento (en esta tesis JAVA).
   La subrutina *getRefText()* devuelve el texto de refinamiento de la asignación asociada.

   1.4. **TCRL_File**

   Módulo físico que representa a un archivo de texto plano que contiene una ley de refinamiento escrita en el lenguaje TCRL v2.0 (ver sección 5). Incluye las subrutinas estándar para el manejo de archivos de texto plano.

   1.5. **AST**

   Módulo lógico que agrupa a los módulos relacionados con la representación del árbol de sintaxis abstracta (AST, por Abstract Syntax Tree) de una ley de refinamieto

escrita en el lenguaje TCRL v2.0 (ver sección 5). Esto es, los diferentes nodos que componen el AST.

1.5.1. TCRL_AST

Módulo físico que representa la raíz del árbol de sintaxis abstracta (AST, por Abstract Syntax Tree). Contiene el nombre, el preámbulo, las reglas de refinamiento y el epílogo de una ley de refinamiento escrita en el lenguaje TCRL v2.o (ver sección 5). Los parámetros del constructor son justamente cada una de estas componentes.

Se exportan las siguientes subrutinas:

- *getName()*, devuelve el nombre de la ley de refinamiento.
- *getPreamble()*, devuelvo el texto de la sección PREAMBLE de la ley de refinamiento.
- *getRules()*, devuelve las reglas de refinamiento de la sección @RULES de la ley de refinamiento.
- *getEpilogue()*, devuelvo el texto de la sección EPILOGUE de la ley de refinamiento.

1.5.2. NodeRules

Módulo físico que representa a todas las reglas de refinamiento contenidas en una ley de refinamiento escrita en el lenguaje TCRL v2.0 (ver sección 5). Cada regla de refinamiento se representa como una instancia del módulo *NodeRule*.

Se exportan las siguientes subrutinas:

- *addRule()*, agrega una regla de refinamiento dando la instancia del módulo *NodeRule* correspondiente y opcionalmente el nombre de la regla.
- *getRule()*, dado el nombre de una regla de refinamiento devuelve la regla de refinamiento asociada.
- *getKeys()*, devuelve una lista de los nombres de todas las reglas de refinamiento contenidas en la ley de refinamiento.

1.5.3. NodeRule

Es un módulo físico que representa a una regla de refinamiento (ya sea de sinónimo o de refinamiento propiamente) del lenguaje TCRL v2.o (ver sección 5). De él heredan los módulos RuleSynonym y RuleRefinement.

1.5.4. RuleSynonym

Módulo que representa a una regla de sinónimo escrita en el lenguaje TCRL v2.0 (ver sección 5).
Los parámetros del constructor son el nombre del sinónimo para usarse dentro de TCRL 2.0 y el tipo de nodo asociado.

Se exportan las siguientes subrutinas:

- *getName()*, devuelve el nombre que se le asigno a la regla de sinónimo.
- *getNodeType()*, devuelve el tipo de nodo asociado a la regla de sinónimo.

1.5.5. RuleRefinement

Módulo que representa a una regla de refinamiento escrita en el lenguaje TCRL

v2.0 (ver sección 5).

Los parámetros del constructor son:

- la lista de nombres o identificadores de la especificación.
- la lista de nombres o identificadores de la implementación junto con sus correspondientes tipos de nodos *NodeType*.
- las opciones de refinamiento complejas que soporta TCRL v2.0. Para más información ver sección 5.

Se exportan las siguientes subrutinas:

- *getSpecIDs()*, devuelve los nombres de las variables o identificadores de la especificación asociadas a la regla de refinamiento.
- *getImplIDs()*, devuelve los nombres de las variables o identificadores de la implementación asociadas a la regla de refinamiento.
- *getNodeType()*, dando como parámetro un nombre de variable o identificador de la implementación, devuelve el tipo de nodo *NodeType* asociado.
- *getRefinementOptions()*, devuelve las opciones complejas para el refinamiento asociadas a la regla de refinamiento.

1.5.6. NodeTypes

Módulo lógico que agrupa a todos los módulos que representan a los diferentes tipos de variables para el refinamiento soportados por TCRL v2.0 (ver sección 5).

1.5.6.1. NodeType

Módulo físico que no representa ningún tipo en particular, sino que de él heredan todos los tipos de variables soportados por TCRL v2.0 (ver sección 5).

1.5.6.2. NodeSynonym

Módulo físico que oculta la estructura de datos utilizada para representar la implementación de una variable de sinónimo dentro de una ley de refinamiento (*SYNONYM[...]* de TCRL v2.0, ver sección 5).

Los parámetros del constructor son el nombre o identificador del sinónimo a usarse dentro de la ley de refinamiento de TCRL v2.0.

Se exporta la subrutina *getID()* que devuelve el nombre o identificador del sinónimo asociado dentro de TCRL v2.0.

1.5.6.3. NodePLType

Módulo físico que oculta la estructura de datos utilizada para representar a una variable que se refina como un tipo de dato primitivo del lenguaje de implementación del sistema a refinar.

Los parámetros del constructor son el texto correspondiente al tipo de dato primitivo del lenguaje de implementación del sistema.

Se exporta la subrutina *getType()* que devuelve el texto correspondiente al tipo de dato primitivo del lenguaje de implementación del sistema asociado.

1.5.6.4. **NodePointer**
Módulo físico que oculta la estructura de datos utilizada para representar a una variable que se refina como un puntero (*POINTER[...]* de TCRL v2.0, ver sección 5) en el lenguaje de implementación del sistema a refinar. Si el sistema se refina a JAVA no se debería utilizar ya que no existen los punteros en JAVA, según el usual uso del concepto en terminología de lenguajes de programación.

Los parámetros del constructor son el tipo de nodo *NodeType* al cual apunta el puntero.

Se exporta la subrutina *getType()* que devuelve el tipo de nodo *NodeType* al cual apunta el puntero.

1.5.6.5. **NodeArray**
Módulo físico que oculta la estructura de datos utilizada para representar a una variable que se refina como un arreglo (*ARRAY[...]* de TCRL v2.0, ver sección 5) en el lenguaje de implementación del sistema a refinar.

Los parámetros del constructor son:
- el tipo nodo *NodeType* del arreglo y
- la cantidad de componentes del arreglo.

Se exportan las siguientes subrutinas:
- *getType()*, devuelve el tipo de nodo *NodeType* del arreglo.
- *getSize()*, devuelve el tamaño del arreglo.

1.5.6.6. **NodeElement**
Módulo físico que oculta la estructura de datos utilizada para representar los campos *NodeElement* de una estructura (*STRUCTURE[...]* de TCRL v2.0) o una lista (*LIST[...]* de TCRL v2.0, ver sección 5).

Los parámetros del constructor son:
- el identificador del campo o elemento de la estructura en la implementación del sistema,
- el tipo de nodo *NodeType* al que se refinaría el campo y
- opcionalmente una constante para asignar al refinamiento del campo evitando el proceso automático de generación de constantes.

Se exportan las siguientes subrutinas:
- *getID()*, devuelve el identificador del campo.
- *getType()*, devuelve el tipo de nodo *NodeType* del campo.
- *getConstant()*, devuelve la constante asignada para el refinamiento en caso de haberla.

1.5.6.7. **NodeStructure**
Módulo físico que oculta la estructura de datos utilizada para representar a una variable que se refina como una estructura (*STRUCTURE[...]* de TCRL

v2.0, ver sección 5) en el lenguaje de implementación del sistema a refinar. Si el sistema se implementa a C se usaría para los *struct* de dicho lenguaje. Si el sistema se refina a JAVA se usaría para las *clases (class)* de dicho lenguaje.

Los parámetros del constructor son:

- el nombre o identificador de la estructura en la implementación del sistema,
- la lista de campos o elementos *NodeElement* de los que está compuesto.

Se exportan las siguientes:

- *getName()*, devuelve el nombre o identificador de la estructura en la implementación del sistema,
- *getElements()*, devuelve la lista de campos o elementos *NodeElement* de los que está compuesto la estructura.

1.5.6.8. **NodeList**

Módulo físico que oculta la estructura de datos utilizada para representar a una variable que se refina como una lista (*LIST[...]* de TCRL v2.0, ver sección 5) en el lenguaje de implementación del sistema a refinar.

Los parámetros del constructor son:

- el nombre o identificador de la lista en la implementación del sistema.
- el tipo de lista (*SLL, DLL, CLL, DCLL*) utilizada. Para más información ver la sección 5.
- el nombre o identificador del campo utilizado para enlazar al nodo siguiente.
- el nombre o identificador del campo utilizado para enlazar al nodo previo, si corresponde.
- la lista de campos *NodeElement* que contiene la lista, sin contar aquellos para el enlace de nodos.
- opcionalmente la función utilizada para pedir memoria en el sistema. Este parámetro es sólo válido para sistemas que se refinan al lenguaje de programación C.

Se exportan las siguientes subrutinas:

- *getName()*, devuelve el nombre o identificador de la lista en la implementación del sistema.
- *getLinkType()*, devuelve el tipo de lista (*SLL, DLL, CLL, DCLL*). Para más información ver la sección 5.
- *getLinkNextName()*, devuelve el nombre del campo utilizado para enlazar al nodo siguiente.
- *getLinkPrevName()*, devuelve el nombre del campo utilizado para enlazar al nodo previo, si es que lo hay.
- *getFields*, devuelve la lista de los campos de la lista (excepto los utilizados para los enlace).
- *getMemalloc()*, retorna, si se especificó, la llamada a la función (con sus respectivos parámetros ya fijados, si los tiene) utilizada para pedir memoria para la estructura que representa un nodo de la lista; si no se define

ninguna. Si no se especifica se utiliza de forma estándar la función de C (*memalloc()*). Esta opción es sólo válida para sistemas que se refinan al lenguaje de programación C.

1.5.6.9. **NodeFile**

Módulo físico que oculta la estructura de datos utilizada para representar a una variable que se refina como un archivo (*FILE[...]* de TCRL v2.0, ver sección 5) en el lenguaje de implementación del sistema a refinar.

Los parámetros del constructor son:

- el nombre o identificador del archivo en la implementación del sistema.
- la ruta del archivo en el sistema.
- el caracter o carecteres utilizados para separar un campo de otro dentro del archivo.
- el caracter o carecteres utilizados para indicar el fin de línea (eol) dentro del archivo.
- el caracter o carecteres utilizados para indicar el fin de archivo (eof) dentro del archivo.
- la estructura del archivo utilizada para almacenar los datos en el archivo (*LINEAR, RPL, FPL*). Para más información ver la sección 5.

Se exportan las siguientes subrutinas:

- *getName()*, devuelve el nombre del archivo en la implementación del sistema.
- *getPath()*, devuelve la ruta del archivo en el sistema.
- *getDelimiter()*, devuelve el caracter o caracteres utilizados para separar un campo de otro dentro del archivo.
- *getEol()*, devuelve el caracter o caracteres utilizados para indicar el fin de línea (eol) dentro del archivo.
- *getEof()*, devuelve el caracter o caracteres utilizados para indicar el fin de archivo (eof).
- *getStructure()*, devuelve una cadena que indica la estructura que se utiliza en la implementación del sistema para almacenar datos en el archivo.

1.5.6.10. **NodeDBColumn**

Módulo físico que oculta la estructura de datos utilizada para representar una columna en una base de datos (*DB[...]* de TCRL v2.0, ver sección 5).

Se exportan las siguientes subrutinas:

- *getColName()*, devuelve el nombre de la columna.
- *getColType()*, devuelve el tipo de la columna.

1.5.6.11. **NodeDB**

Módulo físico que oculta la estructura de datos utilizada para representar a una variable que se refina como una base de datos (*DB[...]* de TCRL v2.0, ver sección 5) en el lenguaje de implementación del sistema a refinar.

Se exportan las siguientes subrutinas:

- *getDBMSID()*, devuelve el identificador que indica que DBMS (MySQL, Microsoft SQL Server, Informix, etc) se utiliza en la implementación del sistema.
- *getConnectionID()*, devuelve el identificador utilizado para la conexión a la base de datos.
- *getTableName()*, devuelve el nombre de la tabla que se utiliza para implementar la variable en el sistema de implementación correspondiente.
- *getColumns()*, devuelve la lista de las columnas (*NodeDBColumn*) que componen la tabla.

2. **FunctionalModules**

Módulo lógico que agrupa a todos los módulos que representan partes funcionales del sistema.

2.1. **TCRL_Parser**

Módulo físico que oculta la implementación del analizador lexicográfico y parser de TCRL v2.0. Para más información consultar las secciones 6.3.1.

Se exporta la subrutina *parse()* que toma un archivo (*TCRL_File*) que contiene la ley de refinamiento escrita en el lenguaje TCRL v2.0 (ver sección 5) y devuelve el árbol de sintaxis abstracta (AST, por Abstract Syntax Tree) generado como una instancia de (*TCRL_AST*).

2.2. **TCRL_PreProcessor**

Módulo físico que se encarga del reemplazo de sinónimos (ver sección 5.3.3) especificados en las posibles reglas de sinónimo contenidas en una ley de refinamiento escrita en el lenguaje TCRL v2.0 (ver sección 5).

Se exporta la subrutina (*preProcess()*), que justamente es la encargada de hacer el preprocesamiento comentado, tomando como parámetro un *TCRL_AST* y devolviendo otro *TCRL_AST* con las modificaciones pertinentes.

2.3. **TCaseRefClient**

Módulo físico que se encarga de manejar las solicitudes de refinamiento de casos de pruebas abstractos generados por FASTest.
También captura los eventos relacionados con la petición y generación de casos de pruebas concretos, *TCaseGenerated* y *TCaseRefineRequested* respectivamente. Crea una instancia del módulo *TCaseRefClientRunner* por cada solicitud de refinamiento.

2.4. **TCaseRefClientRunner**

Módulo físico que oculta la invocación del algoritmo del sistema de refinamiento. Por cada caso de prueba abstracto que se necesita refinar llama a *RefineAST*.

Cada instancia de este módulo puede ser ejecutado en un cliente distinto aprovechando los recursos de cómputos de la red de computadoras disponibles.

Cuando se finaliza el refinamiento del caso de prueba anuncia el evento *TCaseRefined* y por medio de este, devuelve el resultado del refinamiento.

2.5. **RefineAST**

Módulo físico que oculta la implementación del algoritmo encargado de refinar cada caso de prueba abstracto generado por FASTest requerido por *TCaseRefClientRunner*.

Se exporta la subrutina *refine()* que es la encargada del proceso de refinamiento, tomando como parámetros:

- el árbol de sintaxis abstracta (*TCRL_AST*) de una ley de refinamiento escrita en TCRL v2.0 y
- el caso de prueba abstracto (*AbstracTCase*) que se quiere refinar

retornado el caso de prueba concreto (*ConcreteTCase*) correspondiente.

2.6. **RefineExpr**

Es un módulo físico que oculta la implementación del refinamiento de las expresiones de igualdad de un caso de prueba abstracto generado por FASTest basado en su correspondiente regla de refinamiento.

Se exporta la subrutina *refineExprList()* que toma como parámetros:

- una regla de refinamiento (*RuleRefinement*) y
- una lista de expresiones de igualdad (instancia de ZExprList del framework CZT [21]) de un caso de prueba generado por Fastest

y devuelve la asignación (*TCaseAssignment*) del refinamiento correspondiente.

2.7. **ConstantGenerator**

Módulo físico que oculta la implementación de métodos útiles para la traducción de las constantes incluidas en los casos de prueba abstractos generados por FASTest a sus correspondientes constantes en el lenguaje de implementación del sistema a refinar (en esta tesis JAVA).

Bibliografía

[1] P. Stocks, "**Applying formal methods to software testing**", Ph.D. dissertation, Department of Computer Science, University of Queensland, 1993.

[2] H-M. Hörcher y J. Peleska, "**Using formal specifications to support software testing**", Software Quality Journal, vol. 4, pp. 309-327, 1995.

[3] P. Stocks y D. Carrington, "**A framework for specification-based testing**", IEEE Transactions on Software Engineering, vol. 22, no. 11, pp. 777-793, Nov. 1996.

[4] M. Shaw y D. Garlan, "**Software architecture: perspectives on an emerging discipline**", Prentice Hall, Upper Saddle River, 1996.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "**Patrones de diseño**". Addison Wesley, 2003.

[6] "**Information technology - Z formal specification notation - Syntax, type system and semantics**", ISO/IEC 13568:2002(E).

[7] M. Cristiá, "**Catálogo Incompleto de Estilos Arquitectónicos**", http://www.fceia.unr.edu.ar/ingsoft/estilos-cat.pdf, 2006.

[8] M. Cristiá, "**Testing Funcional basado en Especificaciones Z**", http://www.fceia.unr.edu.ar/ingsoft/testing-func-a.pdf, 2007.

[9] M. Cristiá, "**Diseño de Software**", http://www.fceia.unr.edu.ar/ingsoft/diseno-a.pdf, 2008.

[10] Pablo Rodríguez Monetti, "**Fastest: Automatizando el testing de software**", 2009.

[11] Diego Ariel Hollmann, "**TCRL - Refinamiento de casos de prueba para sistema de testing automatizado**", 2009.

[12] Pablo Albertengo, "**Poda automática de árboles de testing**", 2010.

[13] A. Belinfante, J. Feenstra, R. G. de Vries, J. Tretmans, N. Goga, L. M. G. Feijs, S. Mauw, and L. Heerink. "**Formal test automation: A simple experiment**". *Proceedings of the IFIP TC6 12th International Workshop on Testing Communicating Systems*, páginas 179-196. Kluwer, 1999.

[14] C. Jard and T. Jéron. "**TGV: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for nondeterministic reactive systems**". *Int. J. Softw. Tools Technol. Transf.*, 7(4):297-315, 2005.

[15] F. Bouquet and B. Legeard. "**Reification of executable test scripts in formal specification-based test generation: The java card transaction mechanism case study**". *Proc. of FME' 03, Formal Method Europe*, volume 2805 of *LNCS*, pages 778-795, Italy, 2003.

[16] M. Utting, B. Legeard, "**Practical Model-Based Testing. A tools aproach**", Morgan Kaufmann Publisher, 2007.

[17] M. van der Bijl, A. Rensink and J. Tretmans, "**Atomic Action Refinement in Model Based Testing**" Technical Report, Centre for Telematics and Information Technology, University of Twente, Enschede, 2007.

[18] Sebastian Benz "**AspectT: Aspect-Oriented Test Case Instantiation**", BMW Car IT GmbH, *Proceedings of the 7th international conference on Aspect-oriented software development*, 2008.

[19] "**Java**", http://www.java.com/es/about/.

[20] "**Eclipse**", http://www.eclipse.org/.

[21] "**Community Z Tools**", http://czt.sourceforge.net/.

[22] "**Java Compiler Compiler**", https://javacc.dev.java.net/.

[23] "**JavaCC Eclipse Plug-in**", http://sourceforge.net/projects/eclipse-javacc/.