TCRL - Refinamiento de casos de prueba para sistema de testing automatizado

Diego Ariel Hollmann

Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario
Argentina

Director: MSc. Maximiliano Cristiá

Marzo 2009

Resumen

El testing funcional basado en especificaciones es el proceso de testear en forma estructurada y ordenada un software a partir de su especificación. Este proceso 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 las fases anteriores. Para ello, una de las técnicas utilizadas es la de concretización, que busca refinar los casos de prueba abstractos, traduciéndolos desde el lenguaje utilizado para la especificación del sistema al lenguaje en el que éste fue implementado.

En este trabajo se introduce TCRL, un lenguaje que permite traducir casos de prueba abstractos escritos en el lenguaje de especificación Z a casos de prueba concretos escritos en el lenguaje de programación C. Se presenta además, un parser y un intérprete para el mismo y se lo integra con Fastest, un primer prototipo de una herramienta que automatiza el proceso de testing antes mencionado.

Agradecimientos

Quisiera agradecer a la empresa Flowgate Security Consulting por haberme brindado un espacio para poder desarrollar esta tesina con todas las comodidades necesarias y a CIFASIS por haber financiado el proyecto. A mi director Maximiliano Cristiá por haberme dedicado tanto de su tiempo ante cada consulta o necesidad que le planteaba. A Pablo Rodriguez Monetti y Federico Bergero por su gran ayuda desinteresada y aportes puntuales que me brindaron en momentos cruciales del desarrollo del presente trabajo. A todos mis amigos y compañeros que me dieron su ayuda concreta cada vez que la necesité. Y finalmente, pero no menos importante, a mi novia y a toda mi familia por su incondicional y continuo apoyo no solo durante el desarrollo de esta tesina sino a lo largo de toda la carrera, sin ellos no hubiese llegado hasta acá.

Índice general

1. Introducción **4**

2. Testing funcional basado en especificaciones **6**
- 2.1. Especificaciones formales . . . 6
- 2.2. El Lenguaje Z . . . 6
- 2.3. Testing funcional . . . 8
  - 2.3.1. Casos de prueba exitosos, espacio válido de entrada y funciones de refinamiento y abstracción . . . 9
  - 2.3.2. El proceso de testing . . . 10
  - 2.3.3. Ejemplo de Testing . . . 11

3. Fastest **13**

4. Concretización de casos de prueba **17**
- 4.1. Transformación . . . 18
- 4.2. Estudio del Estado del Arte . . . 19
- 4.3. Refinamiento en Fastest . . . 20
  - 4.3.1. TCRL - Test Case Refinement Language . . . 20
  - 4.3.2. Descripción del lenguaje - BNF . . . 20
  - 4.3.3. Ejemplo de una ley de refinamiento . . . 23

5. Módulo de Refinamiento **25**
- 5.1. Diseño . . . 25
- 5.2. Implementación . . . 29
  - 5.2.1. Parser de TCRL . . . 29
  - 5.2.2. Intérprete de TCRL . . . 30
  - 5.2.3. Integración del módulo de refinamiento con *Fastest* . . . 32

6. Casos de estudio **34**
- 6.1. CVS . . . 34
  - 6.1.1. Especificación . . . 35
  - 6.1.2. Implementación . . . 39
  - 6.1.3. Casos de prueba abstractos . . . 40
  - 6.1.4. Ley de refinamiento . . . 41
  - 6.1.5. Resultado del refinamiento . . . 42
- 6.2. Entradas Espectáculo de Teatro . . . 45
  - 6.2.1. Especificación . . . 45
  - 6.2.2. Implementación . . . 46

6.2.3. Casos de prueba Abstractos 47
6.2.4. Ley de refinamiento 48
6.2.5. Resultado del refinamiento 48
6.3. Base de datos de películas 50
6.3.1. Especificación 50
6.3.2. Implementación 52
6.3.3. Casos de prueba abstractos 53
6.3.4. Ley de refinamiento 53
6.3.5. Resultado del refinamiento 54
6.3.6. Otra implementación 55
6.4. Limitaciones de TCRL 56

7. Conclusiones y trabajo futuro **60**

A. Interfaces de los módulos **62**

B. Guía de módulos **68**

C. Código del archivo tcrl.jj **74**

Capítulo 1

Introducción

Desde principios de los años setenta, la comunidad de la ingeniería de software era consciente de la necesidad de aplicar procesos 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 estáticas (tales como análisis de diagramas de diseño y de código fuente) se habían vuelto ampliamente usadas, la técnica de verificación predominante era el testing, el que consiste en ejecutar un programa bajo condiciones controladas, usando datos de entrada reales y observando la salida y/o los resultados que se vayan obteniendo.

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 las especificaciones formales en el testing de software. Esto se debió a la concientización de que las especificaciones informales tenían cierta utilidad, requerida pero limitada en el proceso de testing, pero que los beneficios reales se obtenían desde las especificaciones formales, las que estaban alcanzando cierto grado de maduración y estabilidad.

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

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 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[3] y Stocks y Carrington[4], sea capaz de automatizar o semiautomatizar el proceso de testing funcional basado en especificaciones Z. Un primer prototipo de esta herramienta, ya ha sido desarrollado. El mismo implementa el marco mencionado anteriormente, hasta la etapa en la que los casos de prueba abstractos son generados, inclusive. A partir de acá comienza el trabajo de la presente tesina.

El objetivo es tomar los casos de prueba abstractos y encontrar la forma de generar, a partir de estos, los casos de prueba concretos, es decir, escritos en el lenguaje en el que ha sido implementado el sistema que está siendo testeado. De este modo, los mismos pueden ser ejecutados por el sistema, capturar su salida, abstraer estos resultados y de esta forma poder

compararlos con la especificación para determinar si se ha detectado o no alguna falla.

Este trabajo está motivado por el hecho de que, en la actualidad, no se conoce ningún método implementado (solo algunas ideas) que permita hacer esto para lenguajes de uso común en la industria del software, como puede ser C o Java.

La idea se centra en definir un lenguaje que le permita al usuario (*tester*) de esta herramienta describir en forma sencilla cómo ha sido implementada cada variable de la especificación. Luego, definir un intérprete para este lenguaje que toma un programa de refinamiento y un conjunto de casos de prueba abstractos para generar, a partir de estos, los correspondientes casos concretos, en función de la implementación. Por lo tanto, la salida de este proceso es código que puede ser ejecutado por el sistema testeado.

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

Finalmente, cabe aclarar que este trabajo ha sido financiado por el grupo de ingeniería de software del CIFASIS (http://www.cifasis-conicet.gov.ar/) y por la empresa Flowgate (*http : //www.flowgate.net/*).

Capítulo 2

Testing funcional basado en especificaciones

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 el sistema hace, sin entrar en los detalles de cómo lo hace. Las especificaciones definen aspectos fundamentales del software, mientras que información más detallada y estructurada es omitida. Las mismas 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 es un lenguaje de especificación formal. Fue desarrollado por J. R. Abrial conjuntamente al Grupo de investigación en Programación de la Universidad de Oxford. Z está basado en la teoría de conjuntos y en 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 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 lo modifica 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 conocer algunos conceptos y detalles del lenguaje, veamos un pequeño ejemplo de una especificación de una agenda telefónica y una operación para agregar un contacto.

$[NOMBRE]$

$[TENEFONO]$

$MESSAGE ::= ok \mid error$

Primero se definen dos tipos básicos: *NOMBRE* y *TELEFONO*. Es irrelevante el detalle sobre la estructura de estos tipos, eso queda para implementación. Basta con saber que Z permite armar conjuntos, secuencias y otras estructuras con los tipos básicos; y además, podemos 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*.

A continuación tenemos una *definición axiomática* que define una *constante* del tipo $\mathbb{Z}$ y que asume el valor 1000 (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.

$$\begin{array}{|l} \multicolumn{1}{l}{AgendaTelefonica} \\ \hline at : NOMBRE \pfun TELEFONO \\ \hline \end{array}$$

$$\begin{array}{|l} \multicolumn{1}{l}{Agregar_Ok} \\ \hline \Delta AgendaTelefonica \\ nom? : NOMBRE \\ tel? : TELEFONO \\ rep! : MESSAGE \\ \hline nom? \notin \operatorname{dom} at \\ \# \operatorname{dom} at < capMax \\ at' = at \cup \{nom? \mapsto tel?\} \\ rep! = ok \\ \hline \end{array}$$

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{Agregar_Error} \\
\Xi AgendaTelefonica \\
nom? : NOMBRE \\
tel? : TELEFONO \\
rep! : MESSAGE \\
\hline
\# \operatorname{dom} at \leq capMax \lor nom? \in \operatorname{dom} at \\
rep! = error \\
\hline
\end{array}
$$

$$Agregar \mathrel{\widehat=} Agregar_Ok \lor Agregar_Error$$

Si *nom?* no pertenece al dominio de *at* (no está agendado) y aun hay lugar en la agenda ($\# \operatorname{dom} at < capMax$), se agrega el par $nom? \mapsto tel?$ a la función; si en cambio, está completa ($\# \operatorname{dom} at = capMax$) o *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 *!*, 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*.

2.3. Testing funcional

Testear un sistema significa ejecutarlo bajo condiciones controladas tales que permitan observar su salida o resultados. El testing del 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 que 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 rigurosos.

Como mencionamos al inicio de este capítulo, las especificaciones definen los aspectos fundamentales del sistema. Es por esto que el tester tiene la información importante acerca de la funcionalidad del producto sin tener que extraerla removiendo muchos detalles innecesarios. Testear a partir 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, 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 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. En Z la especificación de un programa es un esquema de operación.

2.3.1. Casos de prueba exitosos, espacio válido de entrada y funciones de refinamiento y abstracción

El testing ve a un programa (o subrutina) 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 se llama *dominio de entrada* del programa y OD es el *dominio de salida*. Normalmente los dominios de entrada y salida son conjuntos de tuplas tipadas cuyos campos 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 forma, 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 $P(x)$. En el mismo sentido, 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 $x \in T$.

Diremos que un caso de prueba es exitoso si descubre un error en el programa. Pero para saber si es exitoso o no debemos recurrir a la especificación que es una descripción diferente del programa. Entonces surge el interrogante de saber cómo expresar los casos de prueba y los resultados del programa en términos de la especificación. De la misma forma que vemos a un programa como una función podemos ver a la especificación Z que le corresponde, es decir un esquema de operación, como otra función, esta vez parcial, que va desde el esquema definido por la declaración de las variables de entrada y de estado en el esquema definido por las variables de salida y de estado de la operación.

Es decir, una especificación Op es una función parcial de IS en OS, donde IS se llama espacio de entrada y OS se llama espacio de salida 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 de P las dimensiones de ID e IS no tienen por qué ser iguales, lo que también 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, definimos el espacio válido de entrada (VIS) de la especificación Op como el subconjunto de IS que satisface la precondición de Op, formalmente:

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

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

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

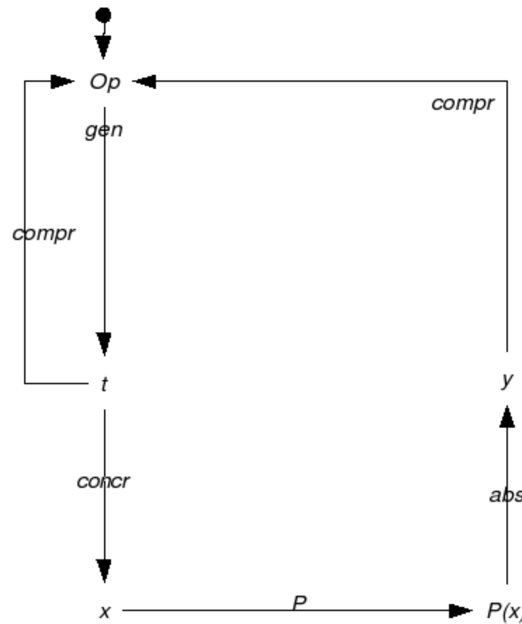

Figura 2.1: El proceso de testing basado en especificaciones formales.

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

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

Los nombres de las funciones refieren a que *concr* refina un elemento a nivel de la especificación en un elemento a nivel de la implementación; y, simétricamente, *abs* abstrae un elemento a nivel de la implementación en un elemento a nivel de la especificación. Por este motivo, las llamaremos funciones de refinamiento y abstracción, respectivamente.

Con todos estos elementos podemos definir el concepto de caso de prueba exitoso. Sea P un programa tal que $P : ID \rightarrow OD$ y sea Op la especificación Z de P con $Op : VIS \rightarrow OS$; sean $t \in VIS$ 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 presentamos una técnica y una herramienta para llevar a cabo la etapa *concr* para diferentes especificaciones e implementaciones.

2.3.2. El proceso de testing

El esquema mostrado en la figura 2.1 muestra el proceso para testear un programa utilizando testing basado en especificaciones. Este puede ser dividido en los siguientes 5 pasos:

1. **Generar** los casos de prueba abstractos a partir de la especificación del sistema.

2. **Concretizar** los casos de prueba abstractos para hacerlos ejecutables.

3. **Ejecutar** los casos concretizados en la implementación del sistema.

4. **Capturar** la salida y **abstraer** los resultados.

5. **Comparar** los resultados con la especificación.

El primer paso, es el correspondiente a *gen* en el ciclo de la figura 2.1. Este utiliza *Op* para generar casos de prueba a nivel de especificación, estos son, casos de prueba abstractos. No es el objetivo de esta tesina entrar en los detalles de cómo generar estos casos de prueba. Simplemente mencionaremos que existen varias *Tácticas de Testing*[9], como la *Forma Normal Disyuntiva* (FND), *Particiones Estándar* (PE), *Propagación de Subdominios* (PSD) y *Mutación de Especificaciones* (ME), que dividen el espacio válido de entrada en clases de prueba. Cada una de estas clases, expresa una alternativa funcional descrita 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 llamamos un *árbol de pruebas* [4].

El paso 2 es justamente el tema sobre el cual se desarrolla el presente trabajo. Es en este punto donde se *concretizan* los casos de prueba abstractos para obtener los casos de prueba concretos. En el capítulo 5 veremos que uno de los métodos para concretizar y ejecutar los casos de prueba abstractos, es la transformación o refinamiento. En la figura 2.1, está representado por *concr*.

Una vez ejecutado los casos ya refinados, se abstraen (*abs*) los resultados obtenidos para llevarlos al nivel de la especificación y poder compararlos con la misma (*compr*); y determinar finalmente si se ha descubierto o no un error.

2.3.3. Ejemplo de Testing

Veamos a continuación un ejemplo de la aplicación de esta metodología utilizando la operación *Actualizar* definida en el apartado 2.2.

Apliquemos ahora alguna táctica de testing a partir de $VIS_{Agregar}$ para generar conjuntos de casos de prueba.

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

Aplicando la *Forma Normal Disyuntiva*[9] obtenemos:

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

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 $\#(\mathrm{dom}\ at) = capTot$ 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 mencionamos más arriba, existen otras tácticas que se podrían seguir aplicando, posiblemente generando nuevas particiones. Luego, por cada partición, se genera un caso de prueba abstracto.

$Agregar_1^{FND}_TCase$

$Agregar_1^{FND}$

$nom? = nom0$
$tel? = tel0$
$at = \{(nom1, tel0)\}$

$Agregar_2^{FND}_TCase$

$Agregar_2^{FND}$

$nom? = nom0$
$tel? = tel0$
$at = \{(nom0, tel0), (nom1, tel1), (nom2, tel2)\}$

$Agregar_3^{FND}_TCase$

$Agregar_3^{FND}$

$nom? \notin \mathrm{dom}\, at$
$nom? = nom0$
$tel? = tel0$
$at = \{(nom0, tel0)\}$

Capítulo 3

Fastest

Fastest nace como 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 según la metodología mostrada en la figura 2.1. Es decir, una herramienta que sea capaz, probablemente con alguna intervención de un tester, de generar casos de prueba abstractos a partir de la especificación 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.

La idea de esta herramienta fue concebida por el MSc. Maximiliano Cristiá y está siendo ya desarrollada por Pablo Rodriguez Monetti [11] dentro del marco de su tesina de grado de la carrera Licenciatura en Ciencias de la Computación de la Universidad Nacional de Rosario. Fastest se desarrolla como parte del proyecto Flowx, el cual es co-financiado por la empresa Flowgate Security Consulting (http://www.flowgate.net/) y por la Agencia Nacional de Promoción Científica y Tecnológica (http://www.agencia.mincyt.org.ar) a través del Fondo Tecnológico Argentino (FONTAR).

Actualmente, está concluido un prototipo de Fastest hasta la fase de la generación de los casos de prueba abstractos inclusive. Justamente, la etapa siguiente, el refinamiento de los casos de prueba abstractos es el tema tratado en este trabajo.

El sistema Fastest fue desarrollado sobre una arquitectura *mixta* combinando los estilos arquitectónicos *cliente/servidor*[16] e *invocación implícita*[16]. Estos estilos se combinan jerárquicamente, dado que cada cliente de la arquitectura cliente/servidor es organizado de acuerdo a una arquitectura de invocación implícita. Además, se utilizó el framework CZT (Community Z Tools [13]), hecho en código abierto Java, utilizado para construir herramientas de métodos formales para el lenguaje Z.

El uso del estilo cliente/servidor fue motivado, principalmente, para aprovechar de manera eficiente los recursos provistos por una red de computadoras. La idea es 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 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 por otros o pueden agregarse nuevos sin afectar las interfaces de los ya existentes. Justamente, este es el caso del módulo de refinamiento, el cual, es añadido al sistema sin tener que hacer modificaciones en las interfaces de los demás módulos.

En cuanto al diseño, podemos mencionar la existencia de un módulo lógico, *TCaseRefinement*, que agrupa los elementos vinculados al refinamiento de casos de prueba abstractos en casos de prueba concretos. Es dentro de este módulo donde se deben incluir aquellos relacionados con el diseño del refinamiento.

Otros módulos que interesan al refinamiento son *TCaseRefined* y *TCaseGenerated*. *TCaseRefined* representa al evento de sistema que debe ser lanzado después de haberse terminado el intento de generación del caso de prueba abstracto. Esto significa que el módulo de refinamiento debe subscribirse a dicho evento para saber cuándo y qué casos de prueba refinar.

Por su parte, *TCaseGenerated* representa al evento de sistema que debe ser lanzado después de haber refinado un caso de prueba abstracto. Es decir, el evento que debe emitir el módulo de refinamiento una vez que termina de refinar un caso de prueba.

Fastest fue implementado en el lenguaje Java. Se utilizó este lenguaje por varias razones, entre las que podemos destacar que se priorizó la utilización de un lenguaje orientado a objetos y también por el hecho de que el framework de CZT ofrece utilidades escritas en Java.

Veamos, ahora, cómo se utiliza Fastest aplicándola al caso de estudio del capítulo 2 y los resultados que se obtienen:

Ejecutamos *Fastest*:

```
> java -jar Fastest.jar

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

Cargamos la especificación (Fastest recibe la especificación Z en formato Latex):

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

Mostramos las operaciones cargadas:

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

Seleccionamos la operación que queremos testear, en este caso, *Agregar*; que es la operación total (disyunción de las otras dos operaciones parciales)

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

Generamos los casos de prueba abstractos. Por defecto, utiliza la *Forma Normal Disyuntiva* como táctica de testing. Justamente, es la que queremos que aplique, dado que es la que usamos nosotros en el ejemplo.

```
Fastest> genalltca
Generating test tree for 'Agregar' operation..
Agregar_DNF_1 test case generation -> SUCCESS.
Agregar_DNF_3 test case generation -> SUCCESS.
Agregar_DNF_2 test case generation -> SUCCESS.
Fastest>
```

Esto nos indica que pudo generar con éxito los tres casos de prueba, uno por cada una de las clases de prueba obtenidas al aplicar la *Forma Normal Disyuntiva*.

Finalmente, mostramos los esquemas de los casos de prueba abstractos generados:

```
Fastest> showsch -tca

\begin{schema}{Agregar\_ DNF\_ 1\_ TCASE}\\
 Agregar\_ DNF\_ 1
\where
 tel? = 0 \\
 nom? = nombre0 \\
 at = \{ ( nombre1 , 0 ) \}
\end{schema}


\begin{schema}{Agregar\_ DNF\_ 2\_ TCASE}\\
 Agregar\_ DNF\_ 2
\where
 tel? = 0 \\
 nom? = nombre0 \\
 at = \{ ( nombre0 , 0 ) , ( nombre1 , 0 ) , ( nombre2 , 0 ) \}
\end{schema}


\begin{schema}{Agregar\_ DNF\_ 3\_ TCASE}\\
 Agregar\_ DNF\_ 3
\where
 tel? = 0 \\
 nom? = nombre0 \\
 at = \{ ( nombre0 , 0 ) \}
\end{schema}

Fastest>
```

Analizando cada caso de prueba, podemos ver que, salvo el nombre de las constantes y de las clases de prueba, los casos de prueba que obtuvo Fastest, en este sencillo ejemplo, son iguales a los que hemos obtenido realizando manualmente el proceso de testing.

Queda entonces, tomar estos casos de prueba abstractos, y generar, a partir de ellos los concretos. Esto es lo que veremos en los próximos capítulos.

Capítulo 4

Concretización de casos de prueba

Volviendo al proceso de testing desarrollado en el capítulo 2, veamos más profundamente ahora la etapa de concretización de los casos de prueba.

Dentro del proceso de testing de software basado en especificaciones, la fase de *refinar* o *concretizar* los casos de prueba abstractos es una etapa importante y puede demandar un gran esfuerzo. En algunas aplicaciones de sistemas de testing basados en especificaciones, el tiempo empleado en el refinamiento puede ser tanto como el tiempo empleado en la modelización, en otras aplicaciones, puede ser entre un 25 % y un 45 % en relación al tiempo de la modelización [7].

El hecho de hacer testing de software basado en especificaciones nos permite pensar en que podemos automatizar o semiautomatizar el proceso. Para esto, se debe automatizar también la ejecución de los casos de prueba generados. El problema radica en que estos casos de prueba generados son altamente abstractos, como la especificación; entonces, por lo general, no contienen suficiente detalles concretos para ser ejecutados directamente.

Para ejecutar los casos generados se debe primero inicializar la implementación, agregar los detalles faltantes en los casos abstractos y arreglar las diferencias entre la API de la especificación y la de la implementación. Además se debe resolver qué hacer con los valores abstractos del modelo y los valores reales de la especificación.

Todo esto no es más que reducir la brecha existente entre la especificación y la implementación del sistema a ser testeado. Para esto existen varias alternativas, las que se podrían clasificar en 3 grupos o enfoques diferentes para reducir esta brecha [7] (Figura 4.1).

- *Adaptación*, es escribir manualmente código para reducir la brecha. 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*, transformar los casos de prueba abstractos en casos de prueba concretos.

- *Mixto*, que es 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.

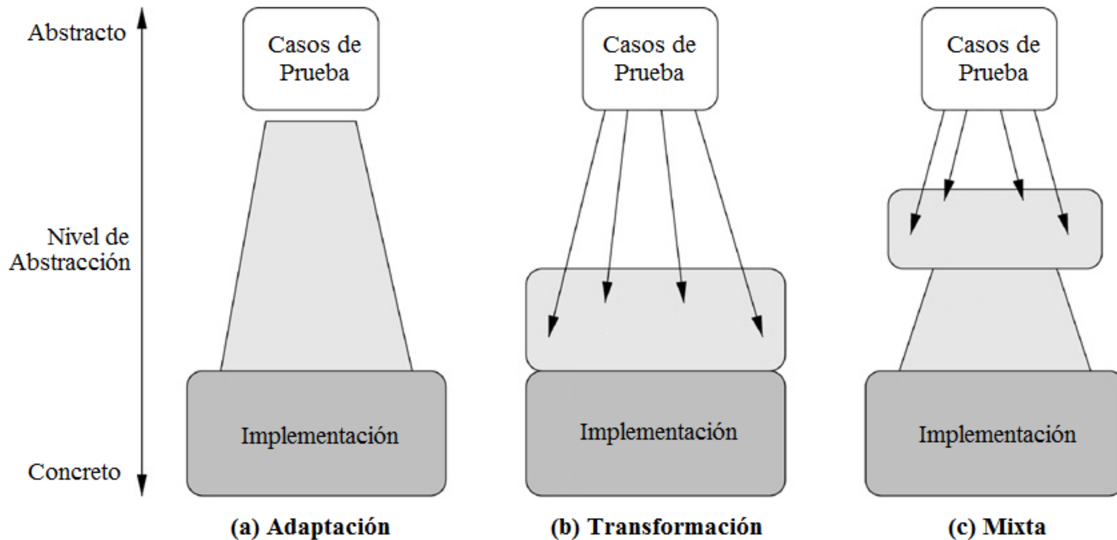

Figura 4.1: Tres enfoques para crear un puente sobre la brecha semántica entre casos de prueba abstractos y la implementación del sistema

4.1. Transformación

De las diferentes alternativas, mencionadas anteriormente, para la concretización de los casos de prueba abstractos, la transformación es justamente la que elegimos para desarrollar este trabajo.

El enfoque de la transformación requiere transformar cada caso de prueba abstracto en un script ejecutable. Esto puede ser en un lenguaje de programación estándar, como C o Java; en un lenguaje de sripts, como TCL o JavaScripts o en alguna notación propietaria.

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

El proceso de transformación es realmente algo más complejo, debido a que deben considerarse los siguientes factores.

- Muchas veces se necesita una configuración o un seteo inicial del entorno y un conjunto de instrucciones para destructores o código que debe ser ejecutado al finalizar el testing. Este podría escribirse en forma manual y ser insertado automáticamente por el motor de transformación al inicio y al final de cada secuencia o ejecución de test.

- 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 de las operaciones de la implementación. Por ejemplo, puede que sea necesario llamar varias operaciones de la implementación para implementar las operaciones abstractas.

- El modelo utiliza constantes y valores abstractos, estos deben ser traducidos a los valores concretos utilizados en la implementación. Esto suele hacerse por medio de un mapeo entre cada valor de la abstracción y un correspondiente valor concreto o una expresión concreta que genera un apropiado valor.

- 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 mas sofisticado para manejar estas estructuras. Este debe generar código fuente ejecutable que incluya sentencias condicionales que chequee la salida de la implementación y tome la rama adecuada hacia el árbol de la siguiente parte a testear.

- 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.

La estructura del test transformado es casi tan importante como el código de cada script de test. Cuando se tienen cientos de miles de tests, las convenciones de nombres y estructuras dentro de una jerarquía de carpetas se vuelve sumamente importante. Muchas compañías que tienen un proceso de testing maduro, tienen estándares para la organización y control de versiones de sus conjuntos de test, y es deseable que los conjuntos de test ejecutables generados basados en una especificación sigan los mismos estándares.

4.2. Estudio del Estado del Arte

Desde la década del '80, se han realizado extensas investigaciones en el área del testing basado en especificaciones. Por otra parte, muchas compañías están usando, actualmente, herramientas para la ejecución automática de testing.

El testing basado en especificaciones aumenta el nivel de automatización aun más. Las herramientas de testing basado en especificaciones generan automáticamente los casos de prueba desde la especificación del software.

Existen varias herramientas ya desarrollas, como comentábamos al principio, para automatizar el proceso de testing basado en especificaciones. Pero, ninguna de ellas parte de una especificación Z y genera casos de prueba concretos en algún lenguaje de programación como C o Java, lo que permitiría ejecutarlos directamente con la implementación del sistema que se está testeando.

Muchas de estas herramientas, crean una *envoltura* alrededor de la implementación para achicar la brecha que existe entre los niveles de abstracción de la especificación y la implementación. Ejemplos de estas son los frameworks de generación de casos de prueba Torkx [5] y TGV [15]. 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 [17], que genera scripts para casos de prueba que son creados desde una especificación hecha en B.

Además, Mark Utting y Bruno Legeard, en [7] hacen una descripción bastante 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 [6], 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 de *Completitud del refinamiento de casos de prueba.*

Finalmente, y más actual, es el trabajo de Sebastien Benz [2] 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 reúso en diferentes contextos de testing. Esta primera aproximación es implementada e integrada en un framework de testing existente.

Luego de investigar y dar con estos resultados, nos encontramos con que aun no se ha desarrollado nada que implemente el refinamiento bajo la idea que se propone en este trabajo.

4.3. Refinamiento en Fastest

Como comentábamos en el capítulo 4, la implementación de Fastest ha sido realizada hasta la obtención de los casos de prueba abstractos. Estos quedan encapsulados dentro de Objetos Java de tipo AbstractTCase. A partir de acá, es donde comienza a desarrollarse nuestro trabajo. Es decir, tomar los casos abstractos e idear una forma para que, en función de cómo se ha implementado el sistema que va a ser testeado, generar los casos concretos para poder luego ser ejecutado.

La idea que tuvimos e implementamos fue la de desarrollar un lenguaje que le permita al tester (el usuario de Fastest) describir cómo fue implementada cada variable de la especificación y demás detalles de la implementación (forma en la que se pide memoria, librerías incluidas, etc). Al código escrito con este lenguaje, lo llamaremos, dentro de Fastest, *Ley de refinamiento.*

4.3.1. TCRL - Test Case Refinement Language

El lenguaje que desarrollamos es el Test Case Refinement Language (TCRL), que es un simple lenguaje declarativo, que permite por medio de unos pocos comandos y una sintaxis muy sencilla describir los detalles de la implementación de un sistema a partir de una especificación. Este, en un principio, toma como lenguaje fuente a Z y como destino, C. Es decir, que el sistema ha sido especificado en Z e implementado en C. Sin embargo, la idea fue diseñarlo de forma tal que en un futuro, puedan agregársele otros leguajes fuente y/o destino.

Por otra parte, es necesario aclarar que el mismo es un prototipo y no intenta cubrir, en un principio, todas las posibles implementaciones de un sistema en C (dado que hay infinidad de formas solamente para implementar, por ejemplo, una función de Z). Pero sí se cubre un amplio espectro dentro de los que consideramos las formas más usuales de programación en C (utilización de arreglos, estructuras; tipos básicos como int, char; manejo de archivos, etc).

4.3.2. Descripción del lenguaje - BNF

Veamos entonces, la descripción de TCRL. Para esto daremos su Backus Naur Form (BNF), forma normal de Backus, que es la manera formal para describir la gramática de un lenguaje; más adelante daremos una descripción informal de la semántica.

Las palabras en mayúscula son *Tokens*, palabras reservadas del lenguaje, y deben escribirse tal cual (el lenguaje es *case-sensitive*). Además, cada sentencia se separa por medio de un *fin de línea.*

La estructura general de una *Ley de Refinamiento* está formada por un identificador, seguido de un preámbulo, las reglas de refinamiento y termina con un epílogo. El identificador es el

⟨*RefFuncion*⟩ ::=⟨*identifier*⟩ *eol*
⟨*preamble*⟩ *eol*
⟨*rules*⟩ *eol*
⟨*epilogue*⟩
⟨*preamble*⟩ ::=**PREAMBLE** *eol*
⟨*CCode*⟩
⟨*epilogue*⟩ ::=**EPILOGUE** *eol*
⟨*CCode*⟩
⟨*rules*⟩ ::=**@BEGINFUNCTION** *eol*
⟨*rule*⟩{*eol* ⟨*rule*⟩}
⟨*rule*⟩ ::=⟨*refinement*⟩ | ⟨*synonym*⟩
⟨*refinement*⟩ ::=⟨*identifier*⟩==>⟨*identifier*⟩**:**⟨*type*⟩
⟨*synonym*⟩ ::=⟨*identifier*⟩==⟨*type*⟩
⟨*type*⟩ ::=⟨*CType*⟩
|⟨*pointer*⟩
|⟨*record*⟩
|⟨*array*⟩
|⟨*list*⟩
|⟨*ZType*⟩
|⟨*rfr*⟩
|⟨*file*⟩
|⟨*db*⟩
⟨*pointer*⟩ ::=**POINTER[**⟨*type*⟩**]**
⟨*record*⟩ ::=**RECORD[**⟨*identifier*⟩,⟨*element*⟩{,⟨*element*⟩}**]**
⟨*list*⟩ ::=**LIST[**⟨*identifier*⟩,⟨*linkType*⟩,⟨*identifier*⟩[,⟨*identifier*⟩],⟨*element*⟩{,⟨*element*⟩}[,⟨*text*⟩]**]**
⟨*element*⟩ ::=⟨*identifier*⟩**:**⟨*CType*⟩
|⟨*identifier*⟩**:**⟨*ZType*⟩
|⟨*identifier*⟩**:**⟨*record*⟩
|⟨*identifier*⟩**:**⟨*array*⟩
|⟨*identifier*⟩**:**⟨*list*⟩
|⟨*identifier*⟩**(** =)⟨*identifier*⟩**:**⟨*type*⟩
⟨*array*⟩ ::=**ARRAY[**⟨*type*⟩, ⟨*number*⟩**]**
⟨*identifier*⟩ ::=*letter*{*letter* | *digit* | **?** | **!** | _ | **-**}
⟨*number*⟩ ::=*digit*{*digit*}
⟨*text*⟩ ::=**"**⟨*character*⟩{,⟨*character*⟩}**"**
⟨*ZType*⟩ ::=**ZTYPE** ⟨*identifier*⟩
⟨*CType*⟩ ::=**CTYPE** ⟨*text*⟩
⟨*linkType*⟩ ::=**SLL** | **DLL** | **CLL** | **DCLL**
⟨*file*⟩ ::=**FILE[**⟨*identifier*⟩,⟨*text*⟩,⟨*text*⟩,⟨*text*⟩,⟨*structure*⟩[, **ENDOFLINE**⟨*text*⟩][, **ENDOFFILE**⟨*text*⟩][,⟨*fieldsRelation*⟩]**]**
⟨*structure*⟩ ::=**LINEAR** | **RPL** | **FPL**
⟨*fieldsRelation*⟩ ::=⟨*identifier*⟩**:**⟨*number*⟩{,⟨*identifier*⟩**:**⟨*number*⟩}
⟨*db*⟩ ::=**DB[**⟨*identifier*⟩,⟨*identifier*⟩,⟨*identifier*⟩,⟨*column*⟩{,⟨*column*⟩}**]**
⟨*column*⟩ ::=⟨*identifier*⟩,⟨*colType*⟩
⟨*colType*⟩ ::=**INT** | **CHAR** | **VARCHAR**
⟨*rfr*⟩ ::=**RFR[**⟨*identifier*⟩,⟨*indentifier*⟩**]**

Figura 4.2: Test Case Refinement Language - BNF

nombre que se le asigna a la ley y se utilizará luego para hacer referencia a ella, desde otras leyes de refinamiento, o para otros fines dentro de Fastest.

En el preámbulo se listan las librerías necesarias para ejecutar el script generado para el test, declarar e inicializar las variables de la implementación, setear conexiones a una base de datos y cualquier otro particular que sea necesario y no se detalle dentro de las reglas de refinamiento. Todo esto es código C, el cual se copia textualmente en el script (⟨*CCode*⟩). En forma análoga al preámbulo, en el prólogo también se escribe código C. En este caso, se utiliza para liberar memoria y demás tareas que haya que hacer una vez finalizada la ejecución del caso de prueba.

Hay dos tipos de reglas. Las de refinamiento: ⟨*identifier*⟩==>⟨*identifier*⟩:⟨*type*⟩ y las de sinónimo: ⟨*identifier*⟩==⟨*type*⟩. Las primeras definen el refinamiento propiamente dicho y las segundas son utilizadas para facilitar la tarea del tester a la hora de escribir las reglas no teniendo que escribir varias veces un mismo tipo de dato.

Veamos ahora en detalle las reglas de refinamiento y el significado de las palabras claves:

- ⟨*identifier*⟩ puede representar cualquier identificador, ya sea el nombre de una variable en la especificación (por eso el hecho de que acepte los caracteres *?* u *!*), el nombre de una variable en la implementación, nombres de los campos de una base de datos, etc.

- ⟨*text*⟩ es mucho mas genérico que ⟨*identifier*⟩. Este acepta cualquier caracter (incluyendo espacios) a excepción de las comillas (“”), que son las que delimitan el contenido.

- ⟨*CType*⟩ se utiliza para describir las variables que se refinan como tipos básicos en C, como *int*, *char*, etc, junto a sus modificafores, *long*, *short*, etc. Por el hecho de que se toma el tipo básico junto a su modificador, se utiliza ⟨*text*⟩, el cual, acepta espacios en blanco a diferencia de ⟨*identifier*⟩.

- ⟨*pointer*⟩ es para representar punteros. De la misma forma que con las otras estructuras de TCRL, pueden definirse términos re cursivos, por lo tanto, el ⟨*type*⟩ que se le pasa a *pointer* puede ser cualquiera.

- ⟨*record*⟩ permite al tester describir variables implementadas como una estructura. El primer parámetro representa el nombre del tipo *struct* que se utiliza en C para definir la estructura, y los subsiguientes parámetros son los campos de la misma. Estos deben ser listados en el orden en que la componente que es representada aparece en la especificación. Es decir, si se utiliza para representar un par ordenado, el primer ⟨*element*⟩ implementa el primer elemento del par; y el siguiente, el segundo.

  Además, el lenguaje le permite al tester definir constantes para algunos campos en particular, indicando también el tipo. Es decir, que en los casos de prueba concretos, esos campos siempre se refinen con el valor constante indicado.

- ⟨*array*⟩ es mucho más sencillo, se utiliza para implementar arreglos, y sus argumentos son el tipo del arreglo y el tamaño.

- ⟨*list*⟩ es ya un caso más complejo. Sirve para indicar si una variable ha sido implementada como una lista enlazada. El primer parámetro, al igual que en *record*, indica el nombre que recibe la estructura con la que se implementa cada nodo. Se detalla luego el tipo de enlace que se utiliza ⟨*linkType*⟩, es decir, si es simplemente enlazada (***SLL***), doblemente enlazada (***DLL***), simplemente enlazada circular(***CLL***) o doblemente enlazada circular(***DCLL***). Los parámetros que le siguen dependen del tipo de lista. Si es ***SLL*** o ***DLL*** se indica el

nombre del campo de la estructura que se utiliza como enlace al siguiente nodo, y si es ***DLL*** o ***DCLL***, además el que enlaza al nodo anterior.

A continuación se listan los campos de la misma forma que en *record*, respetando el orden de la especificación y permitiendo, también, los campos *constantes*. Finalmente, se puede describir, de forma opcional, la función que se utiliza para alojar memoria; esto es el código C de una función que devuelve un puntero. Si no se pone este parámetro, se utiliza *malloc* como forma predeterminada para alojar memoria.

- ⟨*ZType*⟩ sirve para que el tester referencie un sinónimo o alguna variable de la cual ya haya sido descrita su implementación, para permitir reutilizar código TCRL.

- ⟨*rfr*⟩ es similar a *ZType*, con la diferencia que, en este caso, puede hacerse referencia a una variable en otra ley de refinamiento. El primer parámetro es el nombre de la ley a la que se hace referencia, y el segundo, la variable.

- ⟨*file*⟩ es, como en el caso de *list*, más complejo en cuanto a las variantes de sus parámetros. Se utiliza cuando una variable ha sido implementada como un archivo de texto plano. Los cuatro primeros parámetros son obligatorios. Estos son, respectivamente, el nombre del archivo junto a su extensión, la ruta del mismo, y el indicador que se utiliza como delimitador entre registros (tabulación, equis cantidad de espacios en blanco, caracteres especiales, etc.) y la estructura del archivo. Este último indica cómo se deben ir escribiendo los datos de los casos de prueba abstractos dentro del archivo.

  El archivo puede estar estructurado de tres formas, un dato al lado del otro, separados siempre por el delimitador, pero todos sin fin de linea (***LINEAR***); un registro por línea, separando cada campo por el delimitador y cada registro por un fin de línea (***RPL***); o un campo por línea (***FPL***).

  Luego, existen tres parámetros opcionales. Uno para indicar si el caracter de fin de línea es diferente al estándar, otro de forma similar, para el fin de archivo y por último, el orden, o la relación que existe entre cada campo escrito en el archivo y cada componente del tipo representado de la especificación.

- ⟨*db*⟩ es más sencillo que *file*. Utilizado para indicar la implementación de variables en bases de datos. Sus parámetros son, respectivamente, el indentificador de *DBMS* que se utiliza, el identificador de la conexión a la base, el nombre de la tabla y, finalmente, se listan los nombres de las columnas con sus respectivos tipos (***INT***, ***CHAR*** o ***VARCHAR***).

4.3.3. Ejemplo de una ley de refinamiento

Para dar un ejemplo de cómo se escribiría una ley de refinamiento, tomemos nuevamente el ejemplo del capítulo 2. Supongamos que se ha implementado de la siguiente forma.

- *nom*? y *rep*! se implementaron con punteros a string, *nombre* y *respuesta*; y *tel*? como un entero, *telefono*.

- La función *at*, se implementó como el arreglo *agenda* de tamaño *capMax*, o sea, 1000 y cada elemento del arreglo como una estructura, *contacto*.

- La estructura tiene dos campos, uno que implementa *NOMBRE* y el otro *TELEFONO*

- *NOMBRE* se implementa con un *puntero a char* y *TELEFONO* como *long int*

Entonces, la ley de refinamiento quedaría de la siguiente forma:

$$contacto == RECORD[contacto,\ nombre : POINTER[CTYPE\ char], tel : CTYPE\ \text{``}long\ int''] \\ at ==> agenda : ARRAY[contacto,\ 1000]$$

Primero se define un sinónimo, *contacto*, como una estructura tel tipo *struct contacto* para luego ser utilizado en la definición del arreglo *agenda*.

Capítulo 5

Módulo de Refinamiento

En este capítulo describiremos el diseño y la implementación del módulo de refinamiento de Fastest, junto al intérprete y al parser de TCRL que forman parte del mismo.

5.1. Diseño

Una vez definido el lenguaje, pasamos al desarrollo del módulo de refinamiento, que se integrará posteriormente a Fastest.

Como primera fase del desarrollo, definimos el diseño del sistema. Una cosa importante que había que tener en cuenta era la arquitectura de Fastest, para que, una vez que el subsistema de refinamiento estuviera listo, pueda ser incorporado fácilmente dentro del sistema principal. Por lo tanto, antes de diseñar el módulo de refinamiento, tuvimos que estudiar la documentación del diseño de Fastest.

Otro detalle importante y que requirió un período de investigación fue estudiar cómo los casos de prueba abstractos son representados por CZT. El proyecto Community Z Tools (CZT) es un framework hecho en código abierto Java para construir herramientas de métodos formales para el lenguaje Z. Este incluye un conjunto de herramientas para el chequeo de tipos, *parseo* e impresión en LaTex de especificaciones formales escritas en Z estándar.

En nuestro caso particular nos interesaba solamente conocer las estructuras de datos utilizadas para describir las especificaciones de modo de poder obtener los valores de los casos de prueba abstractos. Para conocer más sobre este proyecto ver [13].

El tipo de diseño elegido es el de Diseño basado en ocultación de información, utilizando la metodología de Parnas [10] y aprovechando las ventajas que ofrece el diseño orientado a objetos, como la herencia, el polimorfismo, etc.

Para ayudar a comprender el diseño veamos las figuras 5.1, 5.2, 5.3, 5.4 y 5.5.

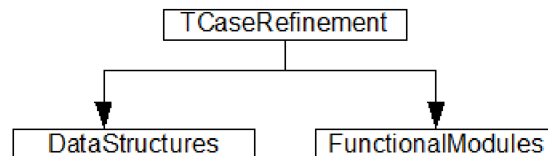

Figura 5.1: Estructura de Módulos - Parte 1

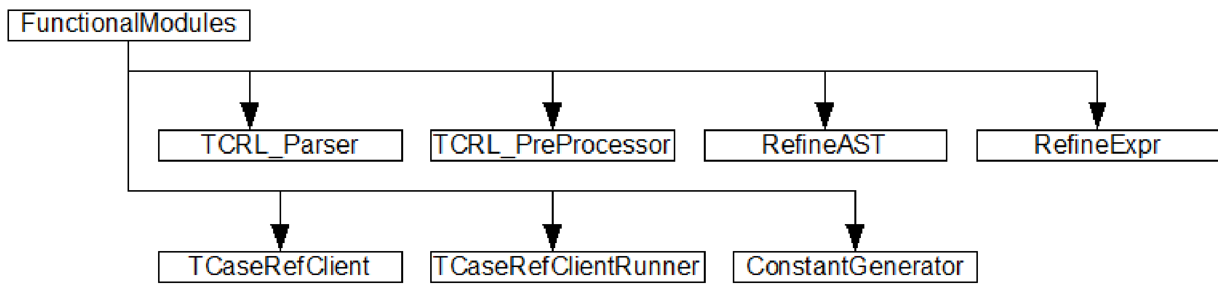

Figura 5.2: Estructura de Módulos - Parte 2

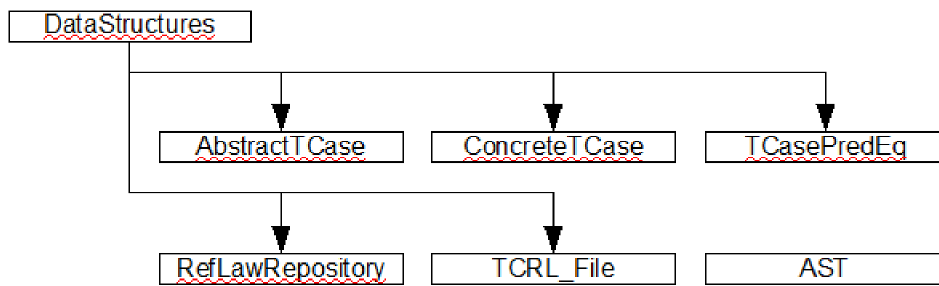

Figura 5.3: Estructura de Módulos - Parte 3

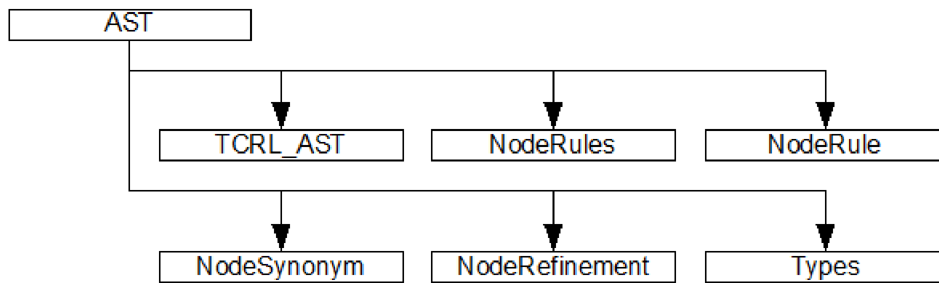

Figura 5.4: Estructura de Módulos - Parte 4

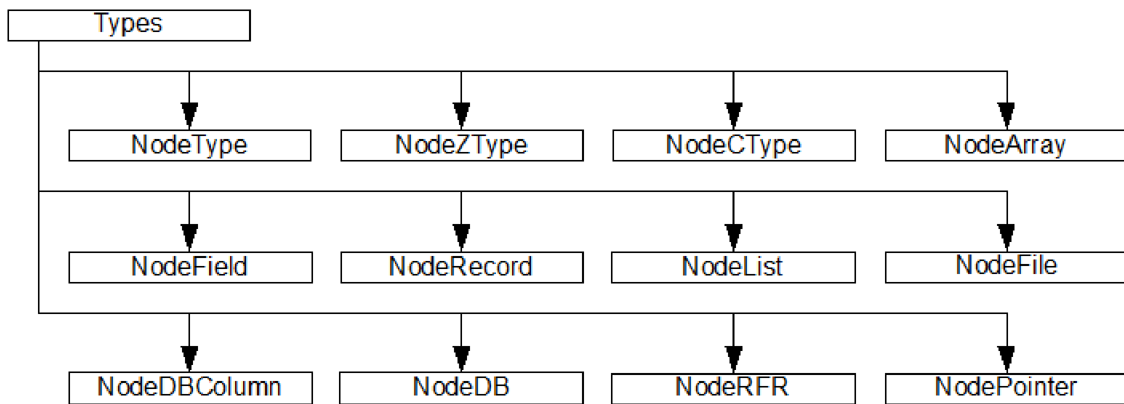

Figura 5.5: Estructura de Módulos - Parte 5

Comentemos ahora un poco el diseño, describiendo algunos de los módulos y sus funcionalidades. Para conocer más en detalle el diseño ver los apéndices A, donde se describen las interfaces de los módulos; y B, donde se describe la guía de módulos.

La estructura de módulos nace a partir del módulo de Fastest *TCaseRefinement*. En este módulo lógico incluimos dos módulos lógicos más: *DataStructures* y *FunctionalModules*. En ellos se incluyen todos los módulos relativos al refinamiento en Fastest. El primero de estos, *DataStructures*, agrupa a los que representan estructuras de datos; y el segundo, *FunctionalModules*, los que representan el funcionamiento del subsistema de refinamiento.

Dentro de los módulos funcionales podemos mencionar, entre los mas importantes, a *RefineAST* y a *TCRL_Parser*.

Module	**TCRL_Parser**
imports	TCRL_File, TCRL_AST
exportsproc	parse(**i** TCRL_File): TCRL_AST

Module	**RefineAST**
imports	TCRL_File, TCRL_AST, TCRL_Parser, AbstractTCase, ConcretetTCase
exportsproc	refine(**i** TCRL_AST, **i** AbstractTCase): ConcretetTCase

El primero es el que encapsula el algoritmo del parser de TCRL y exporta un método utilizado justamente para parsear un programa escrito en este lenguaje. Este método, *parse()* recibe el archivo con el código TCRL y devuelve el árbol de sintaxis abstracta que genera. El segundo módulo es el que, a partir del árbol generado por el parser y de los casos de prueba abstractos, obtiene los casos de prueba concretos. Para esto se vale de otro módulos como *RefineExpr*, que refina expresiones de Z, y *ConstantGenerator* que genera constantes en el lenguaje destino (en nuestro caso, C) a partir de constantes de Z.

Por otra parte, entre los módulos que representan tipos, o estructuras de datos, podemos destacar a *AbstractTCase*, *ConcreteTCase* y *TCRL_AST* que representan, respectivamente, un caso de prueba abstracto, uno refinado y un árbol de sintaxis abstracta generado por el parser. Estos módulos ocultan la implementación de los casos prueba y el árbol, y exportan métodos

que permiten la manipulación de los mismos.

Module	**AbstractTCase**
imports	AxPara
exportsproc	getMyAxPara(): AxPara
comments	AxPara es un módulo de CZT (http://czt.sourceforge.net/)

Module	**ConcreteTCase**
imports	TCasePreEq
exportsproc	setPreamble(**i** String)
	setEpilogue(**i** String)
	addTCasePreEq(**i** TCasePreEq)
	getPreable(): String
	getEpilogue(): String
	getTCasePreEqs(): List(TCasePreEq)

Module	**TCRL_AST**
imports	NodeRules
exportsproc	getName(): String
	getEpilogue(): String
	getPreamble(): String
	getRules(): NodeRules
	TCRL_AST(**i** String, **i** String, **i** String, **i** NodeRules)

Para terminar esta breve descripción del diseño del refinamiento en Fastest, tenemos los módulo *NodeRule* y *NodeType* que junto a sus herederos, representan los nodos del árbol de sintaxis abstracta. Del primero heredan *NodeRefinement* y *NodeSynonym* que representan, respectivamente, leyes de refinamiento y de sinónimo. Del segundo, en cambio, heredan todos los módulos que representan los tipos o estructuras de datos del lenguaje destino utilizados para refinar las diferentes variables que aparecen en la especificación, como por ejemplo arreglos, enteros, archivos, etc.

Module	**NodeRule**
imports	NodeType
exportsproc	getType():NodeType

Module	**NodeType**

5.2. Implementación

5.2.1. Parser de TCRL

Parsear, o analizar sintácticamente, es el proceso de analizar un texto en busca de *tokens* (palabras reservadas del lenguaje) para determinar la estructura gramatical con respecto a una gramática formal dada.

Un parser es el componente de un intérprete o compilador, que chequea sintaxis correcta y construye estructuras de datos (árboles de sintaxis abstracta, u otras estructuras jerárquicas).

Como mencionamos en el capítulo 3, *Fastest* está siendo desarrollado en Java y este es el mismo lenguaje en el que se desarrollaría el módulo de refinamiento. Por lo tanto, la decisión tomada fue la de utilizar también Java para el desarrollar el parser.

Antes de comenzar con el desarrollo del intérprete de TCRL, pensamos cual sería la mejor forma de hacerlo y tratar de hacer un diseño para el mismo. La primer alternativa que se nos ocurrió y que parecía la mas acertada, era la de utilizar el Patrón de Diseño *Interpreter*[12]. Este es utilizado cuando, dado un lenguaje, se define una representación para su gramática a través de un intérprete que utiliza la representación para interpretar sentencias en el lenguaje. Luego de avanzar un tiempo sobre esta idea, diseñando el Intérprete de TCRL siguiendo este patrón, surgió una alternativa que nos pareció más cómoda y que se presentaba como la más acorde a lo que necesitábamos.

Esta alternativa era la de utilizar un lenguaje para generar parsers, *Javacc* (Java Compiler Compiler). Un generador de parsers es una herramienta que lee la especificación de una gramática y la convierte en un programa que puede reconocer correspondencias con la gramática. En este caso en particular, Javacc genera un programa en Java.

Javacc toma como entrada un archivo en el que se describe la gramática del lenguaje, usualmente en BNF, escrito en una notación particular de Javacc y su salida es el parser del lenguaje en cuestión, como código Java. Para conocer más sobre Javacc ver [14].

Como ya teníamos la gramática del lenguaje TCRL en su BNF, era relativamente intuitivo el paso a la notación propia de Javacc. Para hacer esto, necesitábamos definir previamente las clases que formarían parte del árbol de sintaxis abstracta.

Debido a que Javacc permite la inclusión de código Java en su sintaxis (de hecho, es necesario para que la salida del parser sea útil) agregamos la definición de estas clases en el mismo archivo que toma como parámetro Javacc (jj es la extensión asociada a estos archivos) y de esta forma el parser generado por Javacc, respeta el diseño que planteamos para el módulo de refinamiento de Fastest en la sección anterior.

Veamos algunos detalles del código del archivo tcrl.jj. El mismo se encuentra detallo en forma completa en el apéndice C.

Primero se *setean* las diferentes opciones que ofrece Javacc (sección options{}), como por ejemplo, el ir mostrando el *debug* a medida que el parser recorre el texto de una ley de refinamiento escrita en TCRL. Luego, se declaran las clases que serán utilizadas para ir creando el árbol de sintaxis abstracta. Estas fueron declaradas respetando el diseño de las mismas. A continuación, se enumeran los *Tokens*, o palabras reservadas, del lenguaje TCRL y finalmente, se describe la gramática al mismo tiempo que se le indica al Javacc cómo debe ir armando el árbol de sintaxis abstracta en función de la misma.

5.2.2. Intérprete de TCRL

Al igual que el parser, y el resto de Fastest, el intérprete de TCRL se desarrolla en Java.

Para conocer detalles sobre la implementación, comentemos brevemente parte del código. Por cada módulo del diseño se creó una clase Java respetando sus nombres. Además, por cada subrutina que un módulo exporta, se definió un método, dentro de la clase correspondiente, que implementa la función de la subrutina.

Como sucede en la mayoría de los proyectos, estas no fueron las únicas clases creadas ni los únicos métodos definidos sino que se implementaron otras clases y otros métodos que facilitan la implementación del diseño. Las interfaces de los módulos, permanecieron sin modificación. Solamente se agregaron subrutinas en el interior de los módulos para ir resolviendo problemas cada mas pequeños y más simples así como también algunas clases auxiliares para facilitar la resolución de estos subproblemas. De esta forma, se respetó el diseño en su totalidad.,

La clase *TCRL_Parser* se obtuvo al compilar el código de tcrl.jj con *javacc*. Este, crea además otras clases que se utilizan luego durante el parseo de código para generar el árbol de sintaxis abstracta. Algunas de esta son *TCRL_ParserTokenManager*, que se encarga del manejo de los *tokens* del lenguaje, y *TCRL_ParserConstants* que implementa las constantes utilizadas por el parser. Algunos de los métodos de la interfaz de esta clase son los que definimos nosotros en el archivo tcrl.jj, para describir como ir armando el árbol de sintaxis abstracta en función de las clases que representan a los nodos y las leyes de refinamiento que se van parseando.

Las clases declaradas en tcrl.jj que implementan a los módulos que representan nodos del árbol, son bastantes simples. Tomemos una, por ejemplo, la clase que representa el nodo raíz, *TCRL_AST*:

```
class TCRL_AST {
     private String preamble;
     private String epilogue;
     private String name;
     private NodeRules rules;
     TCRL_AST(String name, String preamble, String epilogue ,NodeRules rules) {
        this.name = name;
        this.preamble = preamble;
        this.epilogue = epilogue;
        this.rules = rules;
     }
     public String getName(){
        return name;
     }
     public String getEpilogue(){
        return epilogue;
     }
     public String getPreamble(){
        return preamble;
     }
     public NodeRules getRules(){
        return rules;
     }
}
```

En este caso, se declaran como privados los miembros que representan el nombre de la ley de refinamiento, el preámbulo, el epílogo y las reglas. Además, se define el constructor de la clase, que recibe como parámetros los valores que se le asignan a sus componentes, y los métodos para obtener estos valores desde otra clase. Todas las demás clases que representan nodos del árbol, siguen esta línea de implementación; exceptuando *NodeRules* que implementa un método utilizado para agregar una nueva regla, *NodeRule*.

Para la representación de los casos de prueba concretos, se implementó el módulo *ConcreteTCase* como la siguiente clase Java:

```
public class ConcreteTCase {
    private String preamble;
    private String epilogue;
    private List<TCasePredEq> tCasePredEqs = new ArrayList<TCasePredEq>();
    public void setPreamble(String preable){
        this.preamble = preable;
    }
    public void setEpilogue(String epilogue){
        this.epilogue = epilogue;
    }
    public void addTCasePredEq(TCasePredEq tCasePreEq){
        TCasePreEqs.add(tCasePreEq);
    }

    public String getPreamble(){
        return preamble;
    }

    public String getEpilogue(){
        return epilogue;
    }

    public List<TCasePredEq> getPredEqs(){
        return tCasePredEqs;
    }
}
```

Además de la declaración de las componentes, que se utiliza para representar los datos de un caso de prueba refinado, se definen métodos para poder acceder a ellos desde otra clase y un método para agregar una nueva asignación, *TCasePredEq*.

Veamos ahora algunos detalles sobre la implementación del refinamiento propiamente dicho. El proceso se inicia con las clases *TCaseRefClient* y *TCaseRefClientRunner*. Tomamos la decisión de dividirlo en estas dos clases, ya desde el diseño del refinamiento, para seguir con la idea con la que fue diseñado Fastest, es decir, que pueda correr en varios clientes para optimizar el tiempo. Entonces, *TCaseRefClient* es quien se encarga de capturar los eventos que son de interés para el refinamiento, y crear las nuevas instancias de *TCaseRefClientRunner* por cada caso que se quiere refinar.

TCaseRefClient modifica sus parámetros en función de los eventos *TCaseGenerated* y *RefLawSelected*. Del primero, toma los datos del caso de prueba abstracto generado y que debe

refinarse; y con el segundo, setea con cuál de las leyes de refinamiento del repositorio debe ser refinado el caso de prueba.

Cada instancia de *TCaseRefClientRunner* primero parsea la Ley de Refinamiento que ha sido seleccionada utilizando el parser de TCRL y luego, para refinar el árbol que genera el parser, utiliza la clase *RefineAST*.

RefineAST toma un árbol de sintaxis abstracta y un caso de prueba abstracto y devuelve un caso de prueba concreto. Esta clase, recorrer la estructura del *AbstractTCase* y, en función de lo definido en la *Ley*, refina cada expresión que representa una asignación en un caso de prueba, por medio del método *RefineExpr.refineExprList()* (en CZT, los valores de una asignación son representados con una lista de expresiones, *ExprList*). Este devuelve un *TCaseAssignmnet* y el mismo es añadido a la lista de asignaciones refinadas del caso de prueba concreto (*addTCaseAssignmnet()*).

RefineExpr recibe como parámetros una regla de refinamiento para una variable en particular (*NodeRefinement*) y una lista de expresiones (*ZExprList*) que representan los valores de la asignación que se debe refinar. Este método de forma recursiva, es el que, finalmente, genera el código fuente correspondiente al caso de prueba concreto en el lenguaje de programación de destino, en nuestro caso C.

El hecho de que se ejecute recursivamente se debe a que también de esta forma son representados los datos dentro de las expresiones en CZT. Es decir, para representar en CZT un conjunto que por ejemplo, contiene otros conjuntos, se anidan estos valores por medio de una *Expr* que contienen otras *Expr*.

Para todo el proceso de generación de código C, *RefineExpr* se vale de métodos auxiliares, como por ejemplo, métodos que refinan casos de prueba abstractos a arreglos (*refineWithArray*), métodos que refinan casos de prueba abstractos a estructuras (*refineWithStruc*), métodos que refinan casos de prueba abstractos a bases de datos (*refineWithDB*), etc. Es decir, dependiendo de la implementación de la variable que se quiere refinar, se usa el correspondiente método auxiliar. Todos estos métodos se encuentran dentro del mismo módulo *RefineExpr*.

Finalmente, se cuenta con una clase *ConstantGenerator* donde se implementan otros métodos que son utilizados en diferentes procesos del refinamiento. Uno de ellos, traduce constantes de la especificación en valores de la implementación. Este era un tema bastante importante a resolver, y utilizamos, en este primer prototipo, una manera un tanto rudimentaria. Por ejemplo, cuando se necesita traducir una constante, como *NUM0*, en Z a un valor numérico del refinamiento, como *int* en C, utilizando los valores ascii de las letras de *NUM0* y concatenándolos para formar un número entero.

5.2.3. Integración del módulo de refinamiento con *Fastest*

Debido a que se han aplicado correctamente algunas técnicas de ingeniería de software durante el desarrollo de *Fastest*, estudiando detalladamente la documentación del mismo, se simplificó de manera notable, la integración del módulo de refinamiento con el código de *Fastest*.

Para que la integración pueda hacerse correctamente, había que tener en cuenta algunos aspectos durante el desarrollo del módulo de refinamiento. Principalmente, había que respetar el nombre de las clases ya definidas y, más importante aun, realizar una correcta implementación del manejo de los eventos que tienen injerencia en este módulo. Estos ya fueron mencionados cuando se detallaba el código, en el apartado anterior, *TCaseGenerated*, *TCaseRefined*, *RefLawLoaded* y *RefLawSelected*.

En cuanto al uso del módulo de refinamiento dentro de Fastest, los comandos necesarios

son: AddRefinementLaw, para agregar al repositorio una nueva ley de refinamiento; SelRefinementLaw, para seleccionar una ley del repositorio para el refinamiento; y GenTCaseRefined, para generar el caso de prueba refinado correspondiente a la operación y ley de refinamiento seleccionada.

Capítulo 6

Casos de estudio

En este capítulo mostraremos, a través de algunos casos de estudio, como funciona el refinamiento en Fastest, partiendo de las especificaciones de pequeños sistemas y una breve descripción de sus requerimientos o su funcionamiento.

6.1. CVS

Sistema de Control de Versiones (Cotrol Version System - CVS). Los requerimientos del mismo son:

- El CVS administra un conjunto de programas que constan de un nombre, un encabezado y un cuerpo. Las dos últimas partes son modificables por el usuario.
- Un cierto conjunto de usuarios son lo únicos autorizados a trabajar con el conjunto de programas que el CVS administra.
- Cada usuario debe tener uno de los siguientes roles: Lector, Editor o Autor. Autor implica Lector y Editor.
- El CVS lleva un registro de todos los programas en el conjunto que debe administrar y los usuarios y roles autorizados a trabajar. Además lleva la historia de todas las versiones de cada uno de los programas. El CVS identifica un programa por su nombre.

Las operaciones que el sistema debe ofrecer son:

- Create: un Autor agrega al conjunto de programas uno nuevo.
- Get: un Lector toma del conjunto de programas la última versión de uno de estos pero no puede devolverla. (En cualquier operación "tomar" significa que se hace una copia del programa fuera del conjunto administrado por el CVS pero en ningún caso se elimina el programa de este conjunto).
- Edit: un Editor toma del conjunto de programas la última versión de uno de estos y puede retornarla con modificaciones. Al retornarla está generando una nueva versión del programa. Una vez que un programa fue editado por un usuario ningún otro usuario (incluido él mismo) podrá editarlo o crear otro con el mismo nombre hasta que se efectúe el Delta correspondiente.

- Delta: el Editor que realizó un Edit de cierto programa lo retorna al conjunto de programas administrado por el CVS. Una devolución se registrará como nueva versión siempre y cuando difiera de la anterior.

6.1.1. Especificación

$[NAME]$

$[TEXT]$

$[USER]$

$[TIME]$

$MESSAGE ::= OK \mid UserNotExists \mid UserNotAllowed \mid ProgramAlreadyExists \mid ProgramNotExists \mid ProgramBeingEditing \mid ProgramNotBeingEditing$

$Role ::= Reader \mid Editor \mid Author$

Program

$name : NAME$
$header : TEXT$
$body : TEXT$

Version

$prog : Program$
$changed : TIME$

CVS

$history : NAME \pfun \operatorname{seq} Version$
$rights : USER \pfun Role$
$editing : NAME \pfun USER$

CreateOk

ΔCVS
$p? : Program$
$u? : USER$
$rep! : MESSAGE$

$p?.name \notin \operatorname{dom} history$
$u? \in \operatorname{dom} rights$
$rights(u?) = Author$
$(\exists v : Version \bullet$
$\quad v.prog = p?$
$\quad (\exists now : TIME \bullet$
$\quad\quad v.changed = now$
$\quad\quad history' = history \cup \{p?.name \mapsto \langle v! \rangle\}$
$\quad)$
$)rights' = rights$
$rep! = OK$
$editing' = editing$

CreateError1

ΞCVS
$u? : USER$
$rep! : MESSAGE$

$u? \notin \operatorname{dom} rights$
$rep! = UserNotExists$

CreateError2

ΞCVS
$u? : USER$
$rep! : MESSAGE$

$u? \in \operatorname{dom} rights$
$rights(u?) \neq Author$
$rep! = UserNotAllowed$

CreateError3

ΞCVS
$p? : Program$
$rep! : MESSAGE$

$p?.name \in \operatorname{dom} history$
$rep! = ProgramAlreadyExists$

$Create \triangleq CreateOk \lor CreateError1 \lor CreateError2 \lor CreateError3$

GetOk

$$\Xi CVS$$
$$n? : NAME$$
$$u? : USER$$
$$p! : Program$$

$$u? \in \operatorname{dom} rights$$
$$n? \in \operatorname{dom} history$$
$$p! = (head(history(n?))).prog$$

GetError1

$$\Xi CVS$$
$$n? : NAME$$
$$rep! : MESSAGE$$

$$n? \notin \operatorname{dom} history$$
$$rep! = ProgramNotExists$$

GetError2

$$\Xi CVS$$
$$u? : USER$$
$$rep! : MESSAGE$$

$$u? \notin \operatorname{dom} rights$$
$$rep! = UserNotExists$$

$$Get \triangleq GetOk \lor GetError1 \lor GetError2$$

EditOk

$$\Delta CVS$$
$$u? : USER$$
$$n? : NAME$$
$$rep! : MESSAGE$$

$$rights(u?) = Editor \lor rights(u?) = Author$$
$$n? \notin \operatorname{dom} editing$$
$$editing' = editing \cup \{n? \mapsto u?\}$$
$$history' = history$$
$$rights' = rights$$
$$rep! = OK$$

EditError1

$$\Xi CVS$$
$$u? : USER$$
$$rep! : MESSAGE$$

$$\neg\ (rights(u?) = Editor \lor rights(u?) = Author)$$
$$rep! = UserNotAllowed$$

EditError2

ΞCVS
$n? : NAME$
$rep! : MESSAGE$

$n? \notin \operatorname{dom} history$
$rep! = ProgramNotExists$

EditError3

ΞCVS
$n? : NAME$
$rep! : MESSAGE$

$n? \in \operatorname{dom} editing$
$rep! = ProgramBeingEditing$

$Edit \mathrel{\widehat=} EditOk \lor EditError1 \lor EditError2 \lor EditError3$

DeltaOk

ΔCVS
$u? : USER$
$p? : Program$
$rep! : MESSAGE$

$(p? \mapsto u?) \in editing$
$editing' = editing \setminus (p? \mapsto u?)$
$(\exists v : Version \bullet$
$\quad v.prog = p?$
$(\exists now : TIME \bullet$
$\qquad v.changed = now$
$\qquad history' = history \oplus \{p?.name \mapsto (history(p?.name)) \frown v\}$
$\quad)$
$)$
$rep! = OK$

DeltaError1

ΞCVS
$u? : USER$
$p? : Program$
$rep! : MESSAGE$

$p? \in \operatorname{dom} editing$
$(p? \mapsto u?) \notin editing$
$rep! = UserNotAllowed$

$DeltaError2$

ΞCVS
$p? : Program$
$rep! : MESSAGE$

$p? \notin \operatorname{dom} editing$
$rep! = ProgramNotBeingEditing$

$$Detla \triangleq DeltaOk \lor DeltaError1 \lor DeltaError2$$

6.1.2. Implementación

Para todas las operaciones las variables se implementan de la misma forma. Supongamos, en este ejemplo, que se hizo de la siguiente manera (Código C):

- **u?**

```
int userID;
```

- **n?**

```
char *progName;
```

- **Progam**:

```
struct program{
        char *name;
        char *header;
        char *body;
        }program;
```

- **Version**:

```
struct version{
        struct program prog;
        int changed;
        struct version *next;
};
```

- **history**:

```
struct history{
        char *progName;
        struct version *vers;
        struct history *next;
};
```

- **rights**:

```
struct rights{
        int userID;
        char *role;
};
```

- **editing**:

```
struct editing{
        char *progName;
        int userID;
};
```

6.1.3. Casos de prueba abstractos

Luego de aplicar la táctica *Forma Normal Disyuntiva* sobre las operaciones totales *Create*, *Get*, *Edit* y *Delta*, obtenemos 15 clases de prueba (4 por *Create*, 3 por *Get*, 5 por *Edit* y 3 por *Delta*). Para este ejemplo, tomamos solo una de cada operación y generamos un caso de prueba abstracto por cada clase.

Create_FND_TCASE_1

$VIS_Create_1^{FND}$

$history = \{(prog1, \langle((prog1, header1, body1), time1)\rangle)\}$
$rights = \{(user1, Author)\}$
$editing = \{\}$
$p? = (prog2, header2, body2)$
$u? = user1$

Get_FND_TCASE_2

$VIS_Get_2^{FND}$

$history = \{(prog1, \langle((prog1, header1, body1), time1)\rangle)\}$
$rights = \{(user1, Author)\}$
$editing = \{\}$
$n? = prog2$

Edit_FND_TCASE_4

$VIS_Edit_4^{FND}$

$history = \{\}$
$rights = \{(user1, Reader)\}$
$editing = \{\}$
$u? = user1$

$Delta_FND_TCASE_1$

$VIS_Delta_1^{FND}$

$history = \{(prog1, \langle((prog1, header1, body1), time1)\rangle)\}$
$rights = \{(user1, Editor)\}$
$editing = \{(prog1, user1)\}$
$u? = user1$
$p? = (prog1, header1, body1)$

6.1.4. Ley de refinamiento

Las leyes de refinamiento correspondientes a las operaciones descritas en la sección 6.1.1 y en función de la implementación de la sección 6.1.2 comparten, todas ellas, el mismo preámbulo y las mismas reglas, y en lo único que difieren es en el epílogo. Por lo tanto, primero describimos la parte que tiene en común y a continuación, las particulares.

```
CVS
PREAMBLE

#include <stdio.h>
#include <CVS.c>

char *progName;

int userID;

char *progName;

struct program{
  char *name;
        char *header;
        char *body;
}program;

struct version{
        struct program prog;
        int changed;
    struct version *next;
};

struct history{
        char *progName;
        struct version vers;
    struct history *next;
}history;

struct rights{
        int userID;
        char *role;
    struct right *next;
}rights;

struct editing{
        char *progName;
```

```
        int userID;
    struct editing *next;
}editing;

@BEGINFUNCTION
string == POINTER[CTYPE "char"]
p? ==> program: RECORD[program, name: ZTYPE string, header: ZTYPE string, body: ZTYPE string]
u? ==> userID: CTYPE "int"
n? ==> progName: ZTYPE string
editing ==> editing: LIST[editing, SLL, next, progName: ZTYPE string, userID: ZTYPE u?]
rights ==> rights: LIST[rights, SLL, next, userID: ZTYPE u?, role: ZTYPE string]
version == LIST[version, SLL, next, prog: ZTYPE p?, changed: CTYPE "int"]
history ==> history: LIST[history, SLL, next, progName: ZTYPE string, vers: ZTYPE version]
```

- **Create**

```
EPILOGUE
create(program, userID);
```

- **Get**

```
EPILOGUE
get(progName, userID);
```

- **Edit**

```
EPILOGUE
edit(progName, userID);
```

- **Delta**

```
EPILOGUE
delta(program, userID);
```

6.1.5. Resultado del refinamiento

A continuación, detallamos el código C generado por el refinamiento. Omitimos los preámbulos ya que son iguales en todos los casos y son copiados en forma textual de la ley de refinamiento.

- **Create_FND_TCASE_1**

```
struct history *history_prev = NULL;
struct history *history_node0 = malloc(sizeof(struct history));
history_node0->progName = "prog1";

struct version *vers_prev0 = NULL;
struct version *vers_node0_0 = malloc(sizeof(struct version));

vers_node0_0->prog.name = "prog1";
vers_node0_0->prog.header = "header1";
```

```
vers_node0_0->prog.body = "body1";
vers_node0_0->changed = 110364420;
vers_node0_0->next = NULL;
vers_prev0 = vers_node0_0;

history_node0->vers = *vers_node0_0;
history_node0->next = NULL;
history_prev = history_node0;
history_prev->next = history_node0;
history = *history_node0;

struct rights *rights_prev = NULL;
struct rights *rights_node0 = malloc(sizeof(struct rights));

rights_node0->userID = 111578566;
rights_node0->role = "Author";
rights_node0->next = NULL;
rights_prev = rights_node0;
rights_prev->next = rights_node0;
rights = *rights_node0;

program.name = "prog2";
program.header = "header2";
program.body = "body2";

userID = 111578566;

create(program, userID);
```

- **Get_FND_TCASE_2**

```
struct history *history_prev = NULL;
struct history *history_node0 = malloc(sizeof(struct history));

history_node0->progName = "prog1";

struct version *vers_prev0 = NULL;
struct version *vers_node0_0 = malloc(sizeof(struct version));

vers_node0_0->prog.name = "prog1";
vers_node0_0->prog.header = "header1";
vers_node0_0->prog.body = "body1";
vers_node0_0->changed = 110364420;
vers_node0_0->next = NULL;
vers_prev0 = vers_node0_0;

history_node0->vers = *vers_node0_0;
history_node0->next = NULL;
history_prev = history_node0;
history_prev->next = history_node0;
history = *history_node0;

struct rights *rights_prev = NULL;
struct rights *rights_node0 = malloc(sizeof(struct rights));

rights_node0->userID = 111578566;
```

```
rights_node0->role = "Author";
rights_node0->next = NULL;
rights_prev = rights_node0;
rights_prev->next = rights_node0;
rights = *rights_node0;

progName = "prog2";

get(progName, userID);
```

- **Edit_FND_TCASE_4**

```
struct rights *rights_prev = NULL;
struct rights *rights_node0 = malloc(sizeof(struct rights));

rights_node0->userID = 111578566;
rights_node0->role = "Reader";
rights_node0->next = NULL;
rights_prev = rights_node0;
rights_prev->next = rights_node0;
rights = *rights_node0;

userID = 111578566;

edit(progName, userID);
```

- **Delta_FND_TCASE_1**

```
struct history *history_prev = NULL;
struct history *history_node0 = malloc(sizeof(struct history));

history_node0->progName = "prog1";

struct version *vers_prev0 = NULL;
struct version *vers_node0_0 = malloc(sizeof(struct version));

vers_node0_0->prog.name = "prog1";
vers_node0_0->prog.header = "header1";
vers_node0_0->prog.body = "body1";
vers_node0_0->changed = 110364420;
vers_node0_0->next = NULL;
vers_prev0 = vers_node0_0;

history_node0->vers = *vers_node0_0;
history_node0->next = NULL;
history_prev = history_node0;
history_prev->next = history_node0;
history = *history_node0;

struct rights *rights_prev = NULL;
struct rights *rights_node0 = malloc(sizeof(struct rights));

rights_node0->userID = 111578566;
rights_node0->role = "Editor";
rights_node0->next = NULL;
```

```
rights_prev = rights_node0;
rights_prev->next = rights_node0;
rights = *rights_node0;

struct editing *editing_prev = NULL;
struct editing *editing_node0 = malloc(sizeof(struct editing));

editing_node0->progName = "prog1";
editing_node0->userID = 111578566;
editing_node0->next = NULL;
editing_prev = editing_node0;
editing_prev->next = editing_node0;
editing = *editing_node0;

userID = 111578566;

program.name = "prog1";
program.header = "header1";
program.body = "body1";

delta(progName, userID);
```

6.2. Entradas Espectáculo de Teatro

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.

6.2.1. Especificación

$$[PERSONA, ASIENTO]$$

$$TipoFuncion ::= estandar \mid primera$$

$$MENSAJE ::= ok \mid error$$

TicketParaFunciones

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

$tipoFunc = primera \Rightarrow \operatorname{ran} vendidos \subseteq amigos$

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.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{VentaExitosa}} \\
\Delta \mathit{TicketParaFunciones} \\
p? : \mathit{PERSONA} \\
a? : \mathit{ASIENTO} \\
r! : \mathit{MENSAJE} \\
\hline
a? \in (\mathit{asientos} \setminus \operatorname{dom} \mathit{vendidos}) \\
\mathit{tipoFunc} = \mathit{primera} \Rightarrow (p? \in \mathit{amigos}) \\
\mathit{vendidos}' = \mathit{vendidos} \cup \{a? \mapsto p?\} \\
\mathit{asientos}' = \mathit{asientos} \\
\mathit{tipoFunc}' = \mathit{tipoFunc} \\
\mathit{amigos}' = \mathit{amigos} \\
r! = \mathit{ok}
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{AsientoNoDisponible}} \\
\Xi \mathit{TicketParaFunciones} \\
a? : \mathit{ASIENTO} \\
p? : \mathit{PERSONA} \\
r! : \mathit{MENSAJE} \\
\hline
a? \notin (\mathit{asientos} \setminus \operatorname{dom} \mathit{vendidos}) \lor (\mathit{tipoFunc} = \mathit{primera} \land p? \notin \mathit{amigos}) \\
r! = \mathit{error}
\end{array}
$$

$$\mathit{Venta} \mathrel{\widehat=} \mathit{VentaExitosa} \lor \mathit{AsientoNoDisponible}$$

Venta representa la operación total de la venta de una entrada. Si el asiento requerido no pertenece en 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.

6.2.2. Implementación

Supongamos que las variables de esta operación fueron implementadas de la siguiente manera (Código C):

- **p?**:

  ```
  char *comprador;
  ```

- **amigos**:

  ```
  char *amigos[100];
  ```

- **tipoFunc**

  ```
  char *tipoFunc;
  ```

- **a?**:

```
int asiento;
```

- **asientos**:

```
int asientos[500];
```

- **vendidos**:

```
struct vendidos{
        char *comprador;
        int asiento;
        struct vendidos *siguiente;
}vendidos;
```

- **r!**

```
char *respuesta;
```

La variable de estado *vendidos* se implementa utilizando una lista simplemente enlazada donde cada nodo de la lista es implementado por una estructura del tipo *struct vendidos* y el campo que se utiliza para enlazar los nodos es *siguiente*.

6.2.3. Casos de prueba Abstractos

Aplicando la táctica *Forma Normal Disyuntiva* se generan varias clases de prueba. Tomamos solo algunas para este ejemplo y, por cada una, generamos su correspondiente caso de prueba abstracto.

$Venta_FND_TCase1$

$VIS_Venta_1^{FND}$

$amigos = \{persona1\}$
$asientos = \{asiento1, asiento2\}$
$vendidos = \{\{asiento1, persona1\}\}$
$a? = asiento2$
$tipoFunc = estandar$
$p? = persona1$

$Venta_FND_TCase2$

$VIS_Venta_2^{FND}$

$tipoFunc = primera$
$p? = persona1$
$asientos = \{asiento1, asiento2\}$
$amigos = \{persona2, persona3\}$
$vendidos = \{(asiento1, persona3)\}$
$a? = asiento1$

$Venta_FND_TCase3$

$VIS_Venta_3^{FND}$

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

6.2.4. Ley de refinamiento

A continuación detallamos la ley de refinamiento para la operación *Venta* en función de la implementación de la sección 6.2.2.

```
VentaEntradas
PREAMBLE
#include <stdio.h>
#include <ventaEntradas.c>

char *comprador;
char[100] *amigos;
char *tipoFunc;
int asiento;
int asientos[500];

struct vendidos{
        char *comprador;
        int asiento;
        strcut vendidos *siguiente;
}vendidos;

char *respuesta;

@BEGINFUNCTION
string == POINTER[CTYPE "char"]
p? ==> comprador: ZTYPE string
amigos ==> amigos: ARRAY[ZTYPE string, 100]
tipoFunc ==> tipoFunc: ZTYPE string
a? ==> asiento: CTYPE "int"
asientos ==> asientos: ARRAY[CTYPE "int",500]
vendidos ==> vendidos: LIST[vendidos,SLL,siguiente,asiento:CTYPE "int",comprador:ZTYPE string]
EPILOGUE

venta(comprador, asiento);
```

6.2.5. Resultado del refinamiento

Veamos entonces el código C obtenido por cada caso de prueba abstracto. Omitimos el preámbulo y el epílogo, dado que estos se copian textualmente de la ley de refinamiento y esta es, en este ejemplo, la misma para todos los casos de prueba abstractos.

- **Venta_FND_TCase1**

```
amigos[0] = "persona1";

asientos[0] = -659652682;
asientos[1] = -659652681;

struct vendidos *vendidos_prev = NULL;
struct vendidos *vendidos_node0 = malloc(sizeof(struct vendidos));

vendidos_node0->asiento = -659652682;
vendidos_node0->comprador = "persona1";
vendidos_node0->siguiente = NULL;
vendidos_prev = vendidos_node0;
vendidos_prev->siguiente = vendidos_node0;
vendidos = *vendidos_node0;

asiento = -659652681;
tipoFunc = "estandar";
comprador = "persona1";
```

- **Venta_FND_TCase2**

```
tipoFunc = "primera";
comprador = "persona1";
asientos[0] = -659652682;
asientos[1] = -659652681;
amigos[0] = "persona2";
amigos[1] = "persona3";

struct vendidos *vendidos_prev = NULL;
struct vendidos *vendidos_node0 = malloc(sizeof(struct vendidos));

vendidos_node0->asiento = -659652682;
vendidos_node0->comprador = "persona3";
vendidos_node0->siguiente = NULL;
vendidos_prev = vendidos_node0;
vendidos_prev->siguiente = vendidos_node0;
vendidos = *vendidos_node0;
```

- **Venta_FND_TCase3**

```
asientos[0] = -659652682;
asientos[1] = -659652681;
asiento = -659652681;

struct vendidos *vendidos_prev = NULL;
struct vendidos *vendidos_node0 = malloc(sizeof(struct vendidos));

vendidos_node0->asiento = -659652682;
vendidos_node0->comprador = "persona1";
vendidos_node0->siguiente = NULL;
vendidos_prev = vendidos_node0;
vendidos_prev->siguiente = vendidos_node0;

struct vendidos *vendidos_node1 = malloc(sizeof(struct vendidos));
vendidos_node1->asiento = -659652681;
vendidos_node1->comprador = "persona2";
```

```
vendidos_node1->siguiente = NULL;
vendidos_prev = vendidos_node1;
vendidos_prev->siguiente = vendidos_node1;
vendidos = *vendidos_node0;

comprador = "persona2";
tipoFunc = "primera";
```

6.3. Base de datos de películas

Sistema para almacenar datos sobre películas, con su nombre, director y escritor. La base de datos debe ser capaz de mantener información sobre quién dirigió un film y sobre el guionista. Todo film en la base debe tener asociado un director y un guionista. Se especifican tres operaciones, una que consulte la base de datos en orden a encontrar todas las películas dirigidas por una persona en particular, una para agregar una película a la base de datos y otra para eliminar una ya existente.

6.3.1. Especificación

$[NAME]$

$MESSAGE ::= ok \mid FilmnNotExists \mid FilmAllreadyExists$

Film

$name : NAME$
$dir : NAME$
$wrt : NAME$

FilmsDB

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

GetAllFilmsOfDir

$\Xi FilmBD$
$d? : NAME$
$films! : \mathbb{P}\, Film$

$films! = \{ d : NAMEf : Film \mid d = d? \land f.dir = d \bullet f \}$

ChangeDirOk

$\Delta FilmsDB$
$d? : NAME$
$n? : NAME$
$rep! : MESSAGE$

$(\exists\, p1 : Film \bullet$
$\quad f1.name = n?$
$\quad f1 \in films$
$\quad (\exists\, p2 : Film \bullet$
$\qquad f2.name = n?$
$\qquad f2.dir = d?$
$\qquad f2.wrt = f1.writer$
$\qquad films' = films \setminus \{f1\} \cup \{f2\}$
$\quad)$
$)$
$rep! = ok$

ChangeDirError

$\Xi FilmsDB$
$n? : NAME$
$rep! : MESSAGE$

$\neg\ (\exists f\ : Film \bullet f \in filmslandf.name = n?$
$))$
$rep! = FilmnNotExists$

$ChangeDir \triangleq ChangeDirOk \vee ChangeDirError$

AddFilmOk

$\Delta FilmsDB$
$p? : FILM$
$rep! : MESSAGE$

$p? \notin films$
$films' = films \cup \{p?\}$
$rep! = OK$

AddFilmError

$\Xi FilmsDB$
$f? : FILM$
$rep! : MESSAGE$

$f? \in films$
$rep! = FilmAllreadyExists$

$AddFilm \triangleq AddFilmOk \lor AddFilmError$

$$
\begin{array}{|l}
\hline
\textit{DelFilmOk} \\
\Delta FilmsDB \\
f? : FILM \\
rep! : MESSAGE \\
\hline
f? \in films \\
films' = films \setminus \{f?\} \\
rep! = OK \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{DelFilmError} \\
\Xi FilmsDB \\
f? : FILM \\
rep! : MESSAGE \\
\hline
f? \notin films \\
rep! = FilmNotExists \\
\hline
\end{array}
$$

$DelFilm \triangleq DelFilmOk \lor DelFilmError$

6.3.2. Implementación

Veamos entonces, como puede ser implementado este sistema en C.

- **film**

```
struct film{
        char *name;
        char dirs[10];
        char writers[10];
}film;
```

- **films**

  Implementado como un archivo: “films.db”

- **d?**

```
char *dirName;
```

- **n?**

```
char *filmName;
```

6.3.3. Casos de prueba abstractos

En este ejemplo aplicando la táctica *Forma Normal Disyuntiva* se obtienen 2 clases de prueba por cada operación, excepto para *getAllFilmsOfDir* que tiene un único caso de prueba puesto que para cualquier entrada el predicado de la operación siempre se *evalúa* a **verdadero**.

Detallemos entonces algunos de ellos:

$GetAllFilmsOfDir_FND_TCASE$

$VIS_getAllFilmsOfDir^{FND}$

$films = \{(film1, dir1, writer1)\}$
$d? = dir1$

$ChangeDir_FND_TCASE_2$

$VIS_ChangeDir_2^{FND}$

$films = \{(film1, dir1, writer1)\}$
$n? = film2$
$d? = dir2$

$AddFilm_FND_TCASE_1$

$VIS_AddFilm_1^{FND}$

$films = \{\}$
$f? = (film1, dir1, writer1)$

6.3.4. Ley de refinamiento

A continuación se detallan las leyes de refinamiento escrita en TCRL basándose en la implementación de la sección 6.3.2. Nuevamente, el preámbulo y las reglas son comunes para todas las operaciones, solo difieren en el epílogo.

```
FilmsDB
PREAMBLE
#include <stdio.h>
#include <filmsDB.c>

char *dirName;
char *filmName;
struct film{
    char *name;
    char *dir;
    char *wrt;
}film;

@BEGINFUNCTION
string == POINTER[CTYPE "char"]
films ==> films: FILE["films.db", "\home\user\", " \t ", RPL]
d? ==> dirName: ZTYPE string
n? ==> filmName: ZTYPE string
f? ==> film: RECORD[film,name: ZTYPE string, dir: ZTYPE string, wrt: ZTYPE string]
```

- **GetAllFilmsOfDir** EPILOGUE

  getAllFilmsOfDir(dirName);

- **ChangeDir** EPILOGUE

  changeDir(dirName, filmName);

- **AddFilm** EPILOGUE

  addFilm(film);

6.3.5. Resultado del refinamiento

Veamos entonces, el resultado, es decir, el código C generado por el refinamiento. Nuevamente omitimos los preámbulos.

- **GetAllFilmsOfDir_FND_TCASE**

```
FILE *file_filmsdb;
file_filmsdb = fopen("\home\user\films.db", "w");
if (file_filmsdb == NULL) {
        prinf("OPEN ERROR!"); EXIT(-1);
}
if (fprintf(file_filmsdb, "%s", "film1 \t dir1 \t writer1\n ") < 0) {
        printf("WRITE ERROR!"); EXIT(-1);
}

dirName = "dir1";

getAllFilmsOfDir(dirName);
```

- **ChangeDir_FND_TCASE_2**

```
FILE *file_filmsdb;
file_filmsdb = fopen("\home\user\films.db", "w");
if (file_filmsdb == NULL) {
        prinf("OPEN ERROR!"); EXIT(-1);
}
if (fprintf(file_filmsdb, "%s", "film1 \t dir1 \t writer1\n ") < 0) {
        printf("WRITE ERROR!"); EXIT(-1);
}

filmName = "film2";
dirName = "dir2";

changeDir(dirName, filmName);
```

- **AddFilm_FND_TCASE_1**

```
FILE *file_filmsdb;
file_filmsdb = fopen("\home\user\films.db", "w");
if (file_filmsdb == NULL) {
        prinf("OPEN ERROR!"); EXIT(-1);
}
if (fprintf(file_filmsdb, "%s", "") < 0) {
```

```
                printf("WRITE ERROR!"); EXIT(-1);
        }

        film.name = "film1";
        film.dir = "dir1";
        film.wrt = "writer1";

        addFilm(film);
```

6.3.6. Otra implementación

Supongamos ahora que en lugar de utilizar un archivo, se ha implementado *films* con una tabla de base de datos. La table utilizada es *film* de la base de datos *students*. La conexión a la base se setea en el preámbulo. Entonces, las nueva ley de refinamiento quedarían de la siguiente manera (los epílogos no cambian):

```
FilmsDB
PREAMBLE
#include <mysql.h>
#include <stdio.h>
#include <filmsDB.c>

MYSQL *conn;
MYSQL_RES *res;
MYSQL_ROW row;

char *server = "mysql-server.fceia.unr.edu.ar";
char *user = "fceia";
char *password = "secret";
char *database = "students";

char *dirName;
char *filmName;

struct film{
    char *name;
    char *dir;
    char *wrt;
}film;

@BEGINFUNCTION
string == POINTER[CTYPE "char"]
films ==> films: DB[MYSQL, conn, films, name: CHAR, director: CHAR, writer: CHAR]
d? ==> dirName: ZTYPE string
n? ==> filmName: ZTYPE string
f? ==> film: RECORD[film, name: ZTYPE string, dir: ZTYPE string, wrt: ZTYPE string]
EPILOGUE
```

Obteniendo así, como salida del refinamiento ahora los siguientes resultados (también acá omitimos los preámbulos):

- **GetAllFilmsOfDir_FND_TCASE**

```
conn = mysql_init(NULL);

if (!mysql_real_connect(conn, server,user, password, database, 0, NULL, 0)){
      fprintf(stderr, "%s\n", mysql_error(conn));
      exit(0);
}

if (!sql_query(conn, "INSERT INTO films VALUES('film1','dir1','writer1')")){
        fprintf(stderr, "%s\n", mysql_error(conn));
        exit(0);
}

dirName = "dir1";

getAllFilmsOfDir(dirName);
```

- **ChangeDir_FND_TCASE_2**

```
conn = mysql_init(NULL);

if (!mysql_real_connect(conn, server,user, password, database, 0, NULL, 0)){
      fprintf(stderr, "%s\n", mysql_error(conn));
      exit(0);
}

if (!sql_query(conn, "INSERT INTO films VALUES('film1','dir1','writer1')")){
        fprintf(stderr, "%s\n", mysql_error(conn));
        exit(0);
}
filmName = "film2";
dirName = "dir2";

changeDir(dirName, filmName);
```

- **AddFilm_FND_TCASE_1**

```
conn = mysql_init(NULL);

if (!mysql_real_connect(conn, server,user, password, database, 0, NULL, 0)){
      fprintf(stderr, "%s\n", mysql_error(conn));
      exit(0);
}

film.name = "film1";
film.dir = "dir1";
film.wrt = "writer1";

addFilm(film);
```

6.4. Limitaciones de TCRL

A medida que desarrollábamos los casos de estudio nos fuimos encontrando con algunas limitaciones del lenguaje TCRL, las cuales las descubrimos justamente en este momento y no

fueron pensadas en un principio, durante la creación del lenguaje. Se sugiere, luego, corregir las mismas en un trabajo futuro junto a otras mejoras que se describen en el capítulo 7.

Veamos ahora algunos ejemplos simples sin detallar una especificación completa, a los fines de poder mostrar algunas de estas limitaciones.

- Ejemplo A
    - Especificación

$$\begin{array}{l} \textbf{StudentDB} \\ \hline known : \mathbb{P}\, ID \\ sname : ID \nrightarrow NAME \\ \hline \end{array}$$

$$\begin{array}{l} \textbf{AddStudent} \\ \hline \Delta StudentDB \\ id? : ID \\ name? : NAME \\ \hline id? \notin known \\ sname' = sname \cup \{id? \mapsto name?\} \\ \hline \end{array}$$

    - Impelmentación
        - id?:

```
int id;
```

        - name?:

```
struct name{
        char *first;
        char *last;
}name;
```

        - known:

```
int known[500];
```

        - sname:

```
struct sname{
        int id;
        struct name name;
        struct sname* next;
}sname;
```

    - Caso de prueba abstracto

$$
\begin{array}{|l}
\hline
\textit{AddStudent_TCase} \\
known : \mathbb{P}\, ID \\
sname : ID \rightarrow\!\!\!\!\!\!+ NAME \\
id? : ID \\
name? : NAME \\
\hline
known = \{id1, id2\} \\
sname = \{(id1, name1), (id2, name2)\} \\
id? = id3 \\
name? = name3 \\
\hline
\end{array}
$$

- Ley de refinamiento

```
EjemploA
PREAMBLE
#include <stdio.h>
#include <ejemploA.c>

int id;

struct name{
        char *first;
        char *last;
}name;

int known[500];

struct sname{
        int id;
        struct name name;
        struct sname* next;
}sname;

@BEGINFUNCTION
id? ==> id : CTYPE "int"
name? ==> name: RECORD[name, first: POINTER[CTYPE "char"], last: POINTER[CTYPE "char"]]
known ==> known: ARRAY[CTYPE "int", 500]
sname ==> sname: LIST[sname, SLL, next, id: CTYPE "int", name: ZTYPE name?]

EPILOGUE

addStudent(id, name);
```

- Resultado del refinamiento

  En este caso, el módulo de refinamiento genera un error, debido a una limitación que surge de la implementación del mismo. El problema está en que al querer generar el el valor que debe asumir *sname* en el caso de prueba refinado, el algoritmo encargado de esto, espera encontrar dos valores constantes (sea en un conjunto, una secuencia, etc) cuando examina el valor de *name?* en el caso de prueba abstracto y en lugar de esto, encuentra un único valor, *name3*.

- Ejemplo B

- Especificación

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{StateSchema}} \\
a : \mathit{TIPOA} \\
b : \mathit{TIPOB} \pfun \mathit{TIPOC}
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Operation}} \\
\Delta \mathit{StateSchema} \\
var1? : \mathit{TIPOB} \\
var2? : \mathit{TIPOC} \\
\hline
b' = b \oplus (var1? \mapsto var2?) \\
a' = a
\end{array}
$$

- Implementación
  - a:

    ```
    int varA;
    ```
  - b:

    ```
    float varB[30];
    char *varC[30];
    ```
  - var1?:

    ```
    float var1;
    ```
  - var2?:

    ```
    char *var2;
    ```
- Caso de prueba abstracto

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Operation_TCase}} \\
a : \mathit{TIPOA} \\
b : \mathit{TIPOB} \pfun \mathit{TIPOC} \\
var1? : \mathit{TIPOB} \\
\hline
a = varA1 \\
b = \{(varB1, varC1)\} \\
var1? = varB2
\end{array}
$$

- Ley de refinamiento
  En este ejemplo, encontramos otra limitación del lenguaje TCRL ya que no permite indicar cuándo una variable de la especificación, b, se implementa con más de una variable, *float varB[30]* y *char *varC[30]*.

Capítulo 7

Conclusiones y trabajo futuro

El testing es un proceso muy importante en el desarrollo de software. Una manera de hacerlo más metódico, prolijo, y por lo tanto obteniendo mejores resultados, es a través del testing basado en especificaciones, en particular el testing funcional basado en especificaciones formales.

Dentro del testing funcional basado en especificaciones formales, una de las etapas es la de ejecutar los casos de prueba abstractos generados, utilizando el código de la implementación del sistema que se está testeando. Una forma de hacer esto es refinar o transformar los casos de prueba abstractos en concretos, es decir, llevarlos al nivel de la implementación.

En este trabajo presentamos a **TCRL**, un lenguaje para el refinamiento de casos de prueba abstractos; implementamos un intérprete para el mismo y lo integramos dentro del módulo de refinamiento de **Fastest**, una herramienta que permite automatizar o semiautomatizar el testing funcional basado en especificaciones.

Los resultados obtenidos se aproximaron bastante a lo que nos planteamos en un comienzo dado que la idea era presentar un prototipo que pudiera ser utilizado para el refinamiento de sistemas simples. Si bien encontramos algunas limitaciones en cuanto al tipo de especificaciones e implementaciones que acepta, alcanza para mostrar cómo podría hacerse esto dentro de una herramienta más profesional y de uso comercial.

Estas limitaciones abren la puerta justamente al trabajo futuro, es decir, los pasos a seguir para mejorar, tanto el lenguaje como el propio refinamiento. Entonces, se presentan dos líneas por las cuales seguir avanzado con este proyecto. Por un lado, extender el lenguaje para implementaciones de sistemas más complejos, dotándolo de más funcionalidades y haciéndolo más rico. Y por otro lado, extendiendo la clase **RefineExpr** para que acepte casos de prueba abstractos también más complejos y con otros niveles de anidamiento en sus estructuras.

Una idea para la primera línea de trabajo es darle *significado* a algunos términos de Z para utilizarlos dentro de la misma ley de refinamiento, como puede ser, utilizar # para hacer referencia al tamaño de algún tipo de datos, como un arreglo.

Otra idea dentro de esta misma línea, es la de permitir que en las reglas **RECORD[]** y **LIST[]**, cuando se describen los campos que conforman la estructura, se permita darle valores constantes a algunos de ellos para que cuando sea refinado algún caso de prueba mediante esta regla, esos campos siempre asuman esos valores asignados. Esto permitiría reducir notablemente, en algunos casos, la cantidad de código TCRL que el usuario de la herramienta debe escribir.

Siguiendo esta línea, otra funcionalidad que se le podría agregar a TCRL es poder hacer referencia a un bloque (preámbulo, reglas, epílogo) de una ley de refinamiento en particular

para poder así ser reutilizada en otra. (Elemplos: FilmsDB.preamble, CVS.rules, VentaEntradas.epilogue).

Dentro de la otra línea tenemos, como trabajo a futuro, hacer que la herramienta soporte más casos de prueba abstractos. Ejemplos de estos son estructuras de datos (conjuntos, secuencias, tuplas, etc.) anidadas entre ellas que representan un tipo de dato de la especificación que ha sido implementado como un tipo de dato simple, como puede ser una cadena de caracteres. Otro tema para mejorar, es el caso inverso, es decir, cuando en el caso de prueba abstracto se asigna un valor simple (una palabra, por ejemplo) a un tipo de datos que se implementó como algún tipo compuesto, como una estructura.

Otra limitación, que puede ser quitada en el futuro, es similar a las anteriores y se presenta cuando una variable abstracta es implementada por más de una variable concreta; o el caso inverso, cuando más de una variable abstracta se implementan como una única variable concreta. Esta limitación, así como también la mencionada anteriormente, fueron encontradas durante el desarrollo de los casos de estudio (ver sección 6.4).

Finalmente, otra cosa que queda como trabajo a futuro, es mejorar la generación de las constantes que se utilizan en la implementación. Este es un punto muy importante dentro del refinamiento de casos de prueba abstractos y para el cual existen numerosas formas de atacar el problema. En nuestro caso, optamos por un método un tanto rudimentario y que no es muy eficiente, pero alcanza como para mostrar este primer prototipo de la herramienta.

La generación de estas constantes se encuentra dentro de la clase **ConstantGenerator** y se podría, o bien modificarla o reemplazarla por otra u otras que generen estas constantes de otro modo más eficiente.

Apéndice A

Interfaces de los módulos

Para documentar las interfaces de los módulos que conforman el sistema, utilizaremos un lenguaje muy simple llamado 2MIL. Más detalles sobre el mismo, pueden en contrarse en [10] y en [8].

Module	**TCaseRefinement**
comprises	DataStructures, FunctionalModules

Module	**DataStructures**
comprises	ConcreteTCase, TCasePreEq, RefLawRepository, TCRL_File, AbstractTCase, AST

Module	**AbstractTCase**
imports	AxPara
exportsproc	getMyAxPara(): AxPara
comments	AxPara es un módulo de CZT (http://czt.sourceforge.net/)

Module	**ConcreteTCase**
imports	TCasePreEq
exportsproc	setPreamble(**i** String) setEpilogue(**i** String) addTCasePreEq(**i** TCasePreEq) getPreable(): String getEpilogue(): String getTCasePreEqs(): List(TCasePreEq)

Module	**TCasePredEq**
exportsproc	getSpecName(): String getRefText(): String TCasePreEq(**i** String, **i** String)

Module	**RefLawRepository**
imports	TCRL_File
exportsproc	addRefLaw(**i** String, **i** TCRL_File)
	getRefLaw(**i** String): TCRL_File

Module	**TCRL_File**
exportsproc	read(): String
	write(**i** String
	TCRL_File(**i** String, **i** char)

Module	**AST**
comprises	TCRL_AST, NodeRules, NodeRule, NodeSynonym, NodeRefinement, Types

Module	**TCRL_AST**
imports	NodeRules
exportsproc	getName(): String
	getEpilogue(): String
	getPreamble(): String
	getRules(): NodeRules
	TCRL_AST(**i** String, **i** String, **i** String, **i** NodeRules)

Module	**NodeRules**
imports	NodeRule
exportsproc	addRule(**i** String, **i** NodeRule)
	getRule(**i** String): NodeRule
	getKeys(): Set(String)

Module	**NodeRule**
imports	NodeType
exportsproc	getType():NodeType

Module	**NodeSynonym inherits from** NodeRule
exportsproc	getName(): String
	NodeSynonym(**i** String, **i** NodeType)

Module	**NodeRefinement inherits from** NodeRule
exportsproc	getSpecName(): String
	getRefName(): String
	NodeREfinement(**i** String, **i** String, **i** NodeType)

Module	**Types**
comprises	NodeType, NodeZType, NodeCType, NodeArray, NodeField, NodeRecord, NodeList, NodeFile, NodeDBColumn, NodeDB, NodeRFR

Module	**NodeType**

Module	**NodePointer inherits from** NodeType
exportsproc	getType(): NodeType
	NodePointer(**i** NodeType)

Module	**NodeZType inherits from** NodeType
exportsproc	getType(): String
	NodeZType(**i** String)

Module	**NodeCType inherits from** NodeType
exportsproc	getType(): String
	NodeCType(**i** String)

Module	**NodeArray inherits from** NodeType
exportsproc	getType(): NodeType
	getSize(): int
	NodeZType(**i** NodeType, **i** int)

Module	**NodeField inherits from** NodeType
exportsproc	getName(): String
	getType(): NodeType
	NodeFiled(**i** String, **i** NodeType)

Module	**NodeRecord inherits from** NodeType
imports	NodeField
exportsproc	getFields(): List(NodeField)
	getName(): String
	NodeRecord(**i** String, **i** (ListNodeField))

Module	**NodeList inherits from** NodeType
imports	NodeField
exportsproc	getName(): String
	getLinkNextName(): String
	getFields(): List(NodeField)
	getMemalloc(): String

Module	**NodeSLList inherits from** NodeList
imports	NodeField
exportsproc	getName(): String
	getLinkType(): String
	getLinkNextName(): String
	getFields(): List(NodeField)
	getMemalloc(): String
	NodeList(**i** String, **i** String, **i** List(NodeField), **i** String)

Module	**NodeDLList inherits from** NodeList
imports	NodeField
exportsproc	getName(): String
	getLinkType(): String
	getLinkNextName(): String
	getLinkPrevName(): String
	getFields(): List(NodeField)
	getMemalloc(): String
	NodeList(**i** String, **i** String, **i** String, **i** List(NodeField), **i** String)

Module	**NodeCLList inherits from** NodeList
imports	NodeField
exportsproc	getName(): String
	getLinkType(): String
	getLinkNextName(): String
	getFields(): List(NodeField)
	getMemalloc(): String
	NodeList(**i** String, **i** String, **i** List(NodeField), **i** String)

Module	**NodeDCLList inherits from** NodeList
imports	NodeField
exportsproc	getName(): String
	getLinkType(): String
	getLinkNextName(): String
	getLinkPrevName(): String
	getFields(): List(NodeField)
	getMemalloc(): String
	NodeList(**i** String, **i** String, **i** String, **i** List(NodeField), **i** String)

Module	**NodeFile inherits from** NodeType
exportsproc	getName(): String
	getPath(): String
	getDelimiter(): String
	getEol(): String
	getEof(): String
	getStructure(): String
	NodeList(**i** String, **i** String, **i** String, **i** String, **i** String, **i** String)

Module	**NodeDBColumn inherits from** NodeType
exportsproc	getColName(): String
	getColType(): String
	NodeDBColumn(**i** String, **i** String)

Module	**NodeDB inherits from** NodeType
imports	NodeDBColumn
exportsproc	getDBMSID(): String
	getConnID(): String
	getTable(): String
	getColumns(): List(NodeDBColumns)
	NodeList(**i** String, **i** String, **i** String, **i** List(NodeDBColumn))

Module	**NodeRFR inherits from** NodeType
exportsproc	getRFName(): String
	getZType(): String
	NodeZType(**i** String, **i** String)

Module	**FunctionalModules**
comprises	TCRL_Parser, TCRL_PreProcessor, TCaseRefClient, TCaseRefClientRunner, RefineAST, RefineExpr, Utils

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	TCRL_File, AbstractTCase, ConcreteTCase, TClass, Event_
exportsproc	TCaseRefClient()

Module	**TCaseRefClientRunner**
imports	TCRL_File, AbstractTCase, ConcreteTCase, TClass, TCRL_AST, Event_
exportsproc	TCaseRefClientRunner(**i** String, **i** TClass, **i** AbstractTCase, **i** TCRL_File)

Module	**RefineAST**
imports	TCRL_File, TCRL_AST, TCRL_Parser, AbstractTCase, ConcretetTCase
exportsproc	refine(**i** TCRL_AST, **i** AbstractTCase): ConcretetTCase

Module	**RefineExpr**
imports	NodeRefinement, ZExprList
exportsproc	refineExprList(**i** NodeRefinement, **i** ZExprList): TCaseAssignmnet
comments	ZExprList es un módulo de CZT (http://czt.sourceforge.net/)

Module	**ConstantGenerator**
imports	Expr, ZExprList
exportsproc	cTypeToString(**i** String, **i** Expr): String pointerToCType(**i** Expr): List(Expr)
comments	Expr y ZExprList son módulos de CZT (http://czt.sourceforge.net/)

Apéndice B

Guía de módulos

1. **DataStructures**

   Este es un módulo lógico que agrupa a todos los módulos que representan tipos de datos u objetos utilizados para guardar información, como el archivo de la ley de refinamiento o los nodos del árbol de sintaxis abstracta.

   1.1. **AbstractTCase**

   Módulo que oculta la estructura de datos y los algoritmos para implementar un caso de prueba del lenguaje Z. Está compuesto por varios *AxPara* donde cada uno representa un esquema Z. No incluimos a *AxPara* como un módulo 2MIL y no detallamos su interfaz debido a que es bastante complejo y habría que describir muchos módulos mas que conforman a este y a sus partes. El mismo se encuentra definido dentro del framework CZT [13].

   1.2. **ConcreteTCase**

   Módulo que representa un caso de prueba concreto. Este es el resultado obtenido por el subsistema de refinamiento de Fastest. El mismo, se compone de varias instancias de TCasePreEq.

   1.3. **TCasePredEq**

   Cada instancia de este módulo, almacena un predicado de igualdad de un caso de prueba refinado. Es decir, la representación en el lenguaje destino, en este caso C, del valor que asume una variable del caso de prueba en función de cómo ha sido implementada. La subrutina *getRefText()* devuelve el texto obtenido luego de refinar esa asignación en particular. Este el es el texto que debe escribirse en la salida del subsitema de refinamiento (en un archivo, por consola, etc). La subrutina *getSpecName()* devuelve el nombre de la variable dentro de la especificación.

   1.4. **RefLawRepository**

   Módulo físico que oculta la estructura de datos y los algoritmos implementados para representar el repositorio de las leyes de refinamiento. Exporta dos subrutinas. Una para agregar una nueva ley al repositorio, *addRefLaw()*, que toma como parámetros el nombre de la ley y el archivo donde se la describe (*TCRL_File*); y otra para obtener una ley a partir de su nombre, *getRefLaw()*.

   1.5. **TCRL_File**

Este módulo representa al archivo en el que se describe una ley de refinamiento. Es un archivo de texto plano y su contenido está escrito en TCRL. Se maneja en la forma estándar para el manejo de archivos.

1.6. **AST**

Módulo lógico que agrupa a aquellos módulos relacionados con la representación del árbol de sintaxis abstracta, es decir, los diferentes nodos que lo componen.

1.6.1. **TCRL_AST**

Módulo que representa la raíz del árbol de sintaxis abstracta. Es quien contiene el texto del preámbulo y del epílogo y las reglas de refinamiento, además del nombre de la ley de refinamiento. Exporta 4 subrutinas, *getName()*, *getEpilogue()*, *getPrologue()* y *getRules()*; que devuelven, cada una respectivamente, el nombre, el preámbulo, el epílogo y las reglas. Los parámetros del constructor son, justamente, cada una de estas componentes.

1.6.2. **NodeRules**

En este nodo del árbol se enlazan todas las reglas de refinamiento, cada una, como una instancia de *NodeRule*. Exporta tres subrutinas. Una para agregar un *NodeRule*, *addRule*, que toma como parámetro un identificador (el nombre que tiene en la especificación la variable correspondiente a esta regla) y el propio *NodeRule*. Otra, *getRule()*, que, dado el identificador, retorna el *NodeRule* correspondiente; y una última, *getKeys()*, que devuelve una lista con todos los identificadores dentro de *NodeRules*.

1.6.3. **NodeRule**

Cada *NodeRule* representa una regla de refinamiento para una variable de la especificación. Exporta una subrutina, *getType()*, que devuelve el tipo (una instancia de *NodeType*) de la variable refinada.

1.6.4. **NodeSynonym**

Módulo que representa las reglas que definen sinónimos. La rutina que exporta, *getName()*, devuelve el nombre que se le asignó al sinónimo.

1.6.5. **NodeRefinement**

Cada instancia de *NodeRefinement* es una regla de refinamiento. Sus subrutinas, *getSpecName()* y *getRefName()*, devuelven el nombre de la variable en la especificación y en la implementación, respectivamente. El constructor recibe estos nombres como parámetros.

1.6.6. **Types**

Este módulo lógico agrupa todos los módulos que representan a los tipos de datos con los que se implementa cada una de las variables de la especificación.

1.6.6.1. **NodeType**

El nodo que describe este módulo no representa ningún tipo particular, sino que dé él heredan todos los demás.

1.6.6.2. **NodePointer**

Módulo físico que oculta la estructura de datos utilizada para representar la implementación de una variable cuando se utiliza la regla de refinamiento de TCRL *POINTER[]*. La subrutina que exporta *getType* devuelve el tipo del puntero (representado con un *NodeType*).

1.6.6.3. **NodeZType**

Módulo físico que oculta la estructura de datos utilizada para representar la implementación de una variable cuando se utiliza la regla de refinamiento de TCRL *ZTYPE*. La subrutina *getType()* devuelve el nombre de la variable a la que hace referencia el sinónimo.

1.6.6.4. **NodeCType**
Módulo físico que oculta la estructura de datos utilizada para representar la implementación de una variable cuando se utiliza la regla de refinamiento de TCRL *CTYPE*. La subrutina *getType()* devuelve el tipo C (int, char, etc) al que hace referencia.

1.6.6.5. **NodeArray**
Módulo físico que oculta la estructura de datos utilizada para representar la implementación de una variable cuando se utiliza la regla de refinamiento de TCRL *ARRAY[]*. Exporta dos subrutinas, *getType()*, que devuelve un *NodeType*, el tipo del arreglo; y *getSize()*, el tamaño.

1.6.6.6. **NodeField**
Módulo físico que oculta la estructura de datos utilizada para representar la implementación de los campos de una estructura. Las subrutinas que exporta, *getName()* y *getType()* devuelven el nombre y el tipo (*NodeType*) del campo que representa.

1.6.6.7. **NodeRecord**
Módulo físico que oculta la estructura de datos utilizada para representar la implementación de una variable cuando se utiliza la regla de refinamiento de TCRL *RECORD[]*. La subrutina *getFields* retorna la lista de los campos (*NodeField*) que componen la estructura representada; y *getName()*, el nombre utilizado para definir la estructura.

1.6.6.8. **NodeList**
Módulo físico que oculta la estructura de datos utilizada para representar la implementación de una variable cuando se utiliza la regla de refinamiento de TCRL *LIST[]*. Exporta seis subrutinas:

- *getName()* devuelve el nombre utilizado para definir la estructura utilizada para representar los nodos.
- *getLinkNextName()* devuelve el nombre del campo de la estructura utilizado para enlazar al nodo siguiente.
- *getFields*, al igual que en *NodeRecord*, devuelve la lista de los campos (excepto los utilizados para los enlace).
- *getMemalloc()* retorna 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, es decir, se utiliza la forma estándar (*memalloc()*), devuelve NULL.

1.6.6.9. **NodeListSLL** Módulo físico heredado de NodeList que oculta la estructura de datos utilizada para representar la implementación de una variable con una lista simplemente enlazada.

1.6.6.10. **NodeListCLL** Módulo físico heredado de NodeList que oculta la estructura de datos utilizada para representar la implementación de una variable con una lista circular simplemente enlazada.

1.6.6.11. **NodeListDLL** Módulo físico heredado de NodeList que oculta la estructura de datos utilizada para representar la implementación de una variable con una lista dolemente enlazada. Agrega la subrutina *getLinkPrevName()* que devuelve el nombre del campo utilizado para enlazar al nodo anterior.

1.6.6.12. **NodeListDCLL** Módulo físico heredado de NodeList que oculta la estructura de datos utilizada para representar la implementación de una variable con una lista dolemente enlazada circular. Agrega la subrutina *getLinkPrevName()* que devuelve el nombre del campo utilizado para enlazar al nodo anterior.

1.6.6.13. **NodeFile**

Módulo físico que oculta la estructura de datos utilizada para representar la implementación de una variable cuando se utiliza la regla de refinamiento de TCRL *FILE[]*. Exporta seis subrutinas:

- *getName()* devuelve el nombre del archivo.
- *getPath()* devuelve la ruta del archivo.
- *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; NULL, si no se especifica y se utiliza el fin de línea estandar.
- *getEof()* devuelve el caracter o caracteres utilizados para indicar el fin de archivo; NULL, si no se especifica y se utiliza el fin de archivo estándar.
- *getStructure()* devuelve un identificador que indica la estructura que se utiliza en la implementación para almacenar datos en el archivos. Si no se indica y se utiliza la forma estándar, un registro por línea, devuelve NULL.

1.6.6.14. **NodeDBColumn**

Módulo físico que oculta la estructura de datos utilizada para representar las columnas de una base de datos. Las dos subrutinas que exporta, *getColName()* y *getColType()* devuelven el nombre y el tipo de la columna respectivamente.

1.6.6.15. **NodeDB**

Módulo físico que oculta la estructura de datos utilizada para representar la implementación de una variable cuando se utiliza la regla de refinamiento de TCRL *DB*. Exporta cuatro subrutinas:

- *getDBMSID()* devuelve el identificador que indica que DBMS (MySQL, Microsoft SQL Server, Informix, etc) se utiliza en la implementación
- *getConnID()* devuelve el identificador utilizado para la conexión a la base de datos.
- *getTable()* devuelve el nombre de la tabla que se utiliza para implementar la variable correspondiente.
- *getColumns()* devuelve la lista de las columnas (NodeDBColumn) que componen la tabla.

1.6.6.16. **NodeRFR**

Módulo físico que oculta la estructura de datos utilizada para representar la implementación de una variable cuando quiere hacer referencia a una

variable definida en otra ley de refinamiento. Las subrutinas que exporta, *getZType()* y *getRFName()*, devuelven, respectivamente, el nombre de la variable de la cual se copia la implementación y el nombre de la ley de refinamiento donde se define.

2. **FunctionalModules**

   Este es un módulo lógico que agrupa a todos los módulos que describen partes funcionales del sistema. Estos son *TCRL_Parser*, *TCRL_PreProcesor*, *TCaseRefClient*, *TCaseRefClientRunner*, *RefineAST*, *RefineExpr* y *Utils*.

   2.1. **TCRL_Parser**

   Módulo físico que oculta la implementación del parser de TCRL. La subrutina que exporta *parse()* toma el archivo con la ley de refinamiento (*TCRL_File*) y devuelve el árbol de sintaxis abstracta que genera (*TCRL_AST*).

   2.2. **TCRL_PreProcessor**

   Módulo físico que oculta la implementación del preprocesador del lenguaje. Este, en esta primera versión, es el encargado de reemplazar la utilización de los sinónimos por la regla de refinamiento de la variable a la que estos hacen referencia. Su subrutina exportada (*preProcess()*), justamente, es la encargada de hacer esto, tomando un *TCRL_AST* y devolviendo otro con las modificaciones correspondientes.

   2.3. **TCaseRefClient**

   Módulo físico que se encarga de manejar las solicitudes de refinamiento de casos de prueba en Fastest. Su constructor se encarga de instanciar las variables que lo componen. Este módulo captura eventos como *TCaseGenerated*, *RefLawSelected* y *TCaseRefineRequested*. Con los dos primeros, modifica y setea sus variables internas en función de los eventos y sus parámetros. El tercero, también modifica y setea algunas de sus variables, pero además, crea una nueva instancia de *TCaseRefClientRunner*. Importa un módulo, *TClass*, que no se describe en el presente trabajo. Esto se debe a que solamente se recibe como parámetro y de la misma forma se le pasa a otros módulos. Es decir, no se necesita conocer su estructura ya que no se trabaja con ella. Esta se encuentra detallada dentro del diseño de Fastest [11].

   2.4. **TCaseRefClientRunner**

   Módulo físico que oculta el algoritmo de refinamiento en su totalidad. Por cada caso de prueba abstracto llama a *RefineAST* para refinarlo. Cada instancia de este módulo puede ser ejecutado en un cliente distinto. Su constructor, instancia y setea las variables que lo componen e inicia el proceso de refinamiento. Cuando finaliza anuncia el evento *TCaseRefined* y por medio de este, devuelve el resultado del refinamiento. Con el módulo *TClass* que importa, sucede lo mismo que en *TCaseRefClient*, no se detalla su estructura porque no se la utiliza.

   2.5. **RefineAST**

   Módulo físico que oculta la implementación del algoritmo encargado de refinar cada caso de prueba abstracto. La subrutina que exporta (*refine()*), es la encargada del refinamiento, tomando como parámetros un caso de prueba abstracto (*AbstracTCase*) y el árbol de sintaxis abstracta (*TCRL_AST*) y retorna un caso de prueba concreto (*ConcreteTCase*).

2.6. **RefineExpr**

Módulo físico que oculta la implementación del algoritmo encargado tomar cada instancia de *Expr*[13], que representa un predicado de igualdad (o parte de él) de un caso de prueba abstracto y crear el correspondiente *TCasePredEq*. El método que exporta, *refineExprList()*, toma una lista de *Expr* (instancia de ZExprList[13]) y una regla de refinamiento (*NodeRefinement*) y devuelve un *TCasePreEq*.

2.7. **ConstantGenerator**

Módulo que oculta la implementación de algunos métodos útiles para generar constantes necesarias para el refinamiento. Exporta dos subrutinas, *cTypeToString()* que transforma en texto (código C) un elemento *Expr* en función del *NodeCType* correspondiente; y *pointerToCType()*, que es equivalente a *cTypeToString()* pero para punteros a tipos básicos de C.

Apéndice C

Código del archivo tcrl.jj

```
options {
  LOOKAHEAD = 3;
  CHOICE_AMBIGUITY_CHECK = 2;
  OTHER_AMBIGUITY_CHECK = 1;
  STATIC = true;
  DEBUG_PARSER = false;
  DEBUG_LOOKAHEAD = false;
  DEBUG_TOKEN_MANAGER = false;
  ERROR_REPORTING = true;
  JAVA_UNICODE_ESCAPE = false;
  UNICODE_INPUT = false;
  IGNORE_CASE = false;
  USER_TOKEN_MANAGER = false;
  USER_CHAR_STREAM = false;
  BUILD_PARSER = true;
  BUILD_TOKEN_MANAGER = true;
  SANITY_CHECK = true;
  FORCE_LA_CHECK = true;
}

PARSER_BEGIN(TCRL_Parser)

////////////////////////////////////////////////////////////////////
///Declaraciones de las clases.
////////////////////////////////////////////////////////////////////

class TCRL_AST
{
  private String preamble;
  private String epilogue;
  private String name;
  private NodeRules rules;
  TCRL_AST(String name,String preamble,
       String epilogue,NodeRules rules)
  {
    this.name = name;
    this.preamble = preamble;
    this.epilogue = epilogue;
    this.rules = rules;
  }
  public String getName()
  {
    return this.name;
  }
  public String getEpilogue()
  {
    return this.epilogue;
  }
  public String getPreamble()
  {
    return this.preamble;
  }
  public NodeRules getRules()
  {
    return this.rules;
  }
}

class NodeRules
{
  private HashMap<String,NodeRule> rules;
  public void addRule(String id, NodeRule rule)
  {
    this.rules.put(id,rule);
  }
  public NodeRule getRule(String id)
  {
    return this.rules.get(id);
  }
  public Set getKeys()
  {
    return this.rules.keySet();
  }
  public NodeRules()
  {
    this.rules = new HashMap();
  }
}

class NodeRule
{
  protected NodeType type;
  public NodeType getType()
  {
    return this.type;
  }
}

class NodeSynonym extends NodeRule
{
  private String name;
  NodeSynonym(String name, NodeType type)
  {
    this.name = name;
    this.type = type;
  }
  public String getName()
  {
    return this.name;
  }
}

class NodeRefinement extends NodeRule
{
  private String specName;
  private String refName;
  NodeRefinement(String specName, String refName ,NodeType type)
  {
    this.specName = specName;
    this.refName = refName;
    this.type = type;
  }
  public String getSpecName()
  {
    return this.specName;
  }
  public String getRefName()
  {
    return this.refName;
  }
}

class NodeType
{

}

class NodePointer extends NodeType
{
  private NodeType type;
  public NodeType getType()
  {
    return this.type;
  }
  NodePointer(NodeType type)
  {
    this.type=type;
  }
}
```

```
class NodeZType extends NodeType
{
  private String type;
  NodeZType(String type)
  {
    this.type = type;
  }
  public String getType()
  {
    return this.type;
  }
}

class NodeCType extends NodeType
{
  private String type;
  NodeCType(String type)
    {
    this.type = type;
  }
  public String getType()
  {
    return this.type;
  }
}

class NodeArray extends NodeType
{
  private NodeType type;
  private int size;
  NodeArray(NodeType type, int size)
  {
    this.type = type;
    this.size = size;
  }
  public NodeType getType()
  {
    return this.type;
  }
  public int getSize()
  {
    return this.size;
  }
}

class NodeField extends NodeType
{
  private String name;
  private NodeType type;
  public String getName()
  {
    return this.name;
  }
  public NodeType getType()
  {
    return this.type;
  }
  NodeField(String name, NodeType type)
  {
      this.name = name;
    this.type = type;
  }
}

class NodeRecord extends NodeType
{
  private String name;
  private List<NodeField> fields;
  public String getName()
  {
    return this.name;
  }
  public List<NodeField> getFields()
  {
    return this.fields;
  }
  NodeRecord(String name, List<NodeField> fields)
  {
    this.name = name;
    this.fields = fields;
  }
}

class NodeList extends NodeType
{
  private String name;
  private String linkNextName;
  private String memalloc;
  List<NodeField> fields;
  public String getName()
  {
    return this.name;
  }
  public String getLinkNextName()
  {
    return this.linkNextName;
  }
  public List<NodeField> getFields()
  {
    return this.fields;
  }
  public String getMemalloc()
  {
    return this.memalloc;
  }
}

class NodeSLList extends NodeList
{
  NodeSLList(String name, String linkNextName,
         List<NodeField> fields, String memalloc)
  {
    this.name = name;
    this.linkNextName = linkNextName;
    this.fields = fields;
    this.memalloc = memalloc;
  }
}

class NodeDLList extends NodeList
{
  private String linkPrevName;
  public String getLinkPrevName()
  {
    return this.linkPrevName;
  }
  NodeDLList(String name, String linkNextName, String linkPrevName,
         List<NodeField> fields, String memalloc)
  {
    this.name = name;
    this.linkNextName = linkNextName;
    this.linkPrevName = linkPrevName;
    this.fields = fields;
    this.memalloc = memalloc;
  }
}

class NodeCLList extends NodeList
{
  NodeCLList(String name, String linkNextName,
         List<NodeField> fields, String memalloc)
  {
    this.name = name;
    this.linkNextName = linkNextName;
    this.fields = fields;
    this.memalloc = memalloc;
  }
}

class NodeDCLList extends NodeList
{
  private String linkPrevName;
  public String getLinkPrevName()
  {
    return this.linkPrevName;
  }
  NodeDCLList(String name, String linkNextName, String linkPrevName,
        List<NodeField> fields, String memalloc)
  {
    this.name = name;
    this.linkNextName = linkNextName;
    this.linkPrevName = linkPrevName;
    this.fields = fields;
    this.memalloc = memalloc;
  }
}

class NodeFile extends NodeType
{
  private String name;
  private String path;
  private String delimiter;
  private String eof;
  private String eol;
  private String structure;
  public String getName()
  {
    return this.name;
  }
  public String getPath()
  {
    return this.path;
  }
  public String getDelimiter()
  {
    return this.delimiter;
  }
  public String getEol()
  {
    return this.eol;
  }
  public String getEof()
```

```
  {
    return this.eof;
  }
  public String getStructure()
  {
    return this.structure;
  }
  NodeFile(String name, String path, String delimiter,
       String eol, String eof, String structure)
  {
    this.name = name;
    this.path = path;
    this.delimiter = delimiter;
    this.eol = eol;
    this.eof = eof;
    this.structure = structure;
  }
}

class NodeDBColumn extends NodeType
{
  private String colName;
  private String colType;
  public String getColName()
  {
    return this.colName;
  }
  public String getColType()
  {
    return this.colType;
  }
  NodeDBColumn(String colName, String colType)
  {
    this.colName = colName;
    this.colType = colType;
  }
}

class NodeDB extends NodeType
{
  private String dbmsID;
  private String connID;
  private String table;
  private List<NodeDBColumn> columns;
  public String getDBMSID()
  {
    return this.dbmsID;
  }
  public String getConnID()
  {
    return this.connID;
  }
  public String getTable()
  {
    return this.table;
  }
  public List<NodeDBColumn> getColumns()
  {
    return this.columns;
  }
  NodeDB(String dbmsID, String connID, String table,
       List<NodeDBColumn> columns)
  {
    this.dbmsID = dbmsID;
    this.connID = connID;
    this.table = table;
    this.columns = columns;
  }
}

class NodeRFR extends NodeType
{
  private String rfName, ztype;
  public String getRFName()
  {
    return this.rfName;
  }
  public String getZType()
  {
    return this.ztype;
  }
  NodeRFR(String rfName, String ztype)
  {
    this.rfName = rfName;
    this.ztype = ztype;
  }
}


public class TCRL_Parser
{
  public static void main(String args[]) throws ParseException
  {
    TCRL_AST ast;
    TCRL_Parser parser = new TCRL_Parser(System.in);
    ast = parser.parse();
  }
}

PARSER_END(TCRL_Parser)

////////////////////////////////////////////////////////////////////
/// Declaración de los TOKENS
////////////////////////////////////////////////////////////////////

SKIP :
{
  " "
  |"\t"
  |"\r"
}

TOKEN :
{
  <PREAMBLE: "PREAMBLE"> : CCodeState
  | <EPILOGUE: "EPILOGUE"> : CCodeState
  | <RECORD: "RECORD">
  | <LIST: "LIST">
  | <ARRAY: "ARRAY">
  | <SLL: "SLL">
  | <DLL: "DLL">
  | <CLL: "CLL">
  | <DCLL: "DCLL">
  | <FILE: "FILE">
  | <DB: "DB">
  | <CTYPE: "CTYPE">
  | <ZTYPE: "ZTYPE">
  | <POINTER : "POINTER">
  | <LINEAR: "LINEAR">
  | <RPL: "RPL">
  | <FPL: "FPL">
  | <RFR: "RFR">
  | <ENDOFFILE: "ENDOFFILE">
  | <ENDOFLINE: "ENDOFLINE">
  | <CHAR: "CHAR">
  | <INT: "INT">
  | <VARCHAR: "VARCHAR">
  | <NUMBER: (["0"-"9"])+>
  | <ID: ("*" ( ["a"-"z","A"-"Z","0"-"9","!","?","_","-"])*)
      | (["a"-"z","A"-"Z"] (["a"-"z","A"-"Z","0"-"9","!","?","_","-"])*)>
  | <COLON: ":">
  | <ARROW: "==>">
  | <DEQUAL: "==">
  | <LBRACKET: "[">
  | <RBRACKET: "]">
  | <LPAREN: "(">
  | <RPAREN: ")">
  | <COMMA: ",">
  | <EOL: "\n">
  | <STRING: "\"" (~["\""])* "\"" >
}

<CCodeState> TOKEN :
{
  <BEGINFUNCTION: "@BEGINFUNCTION"> : DEFAULT
  | <CCODE: (~["@"])+>
}

////////////////////////////////////////////////////////////////////
//// Descripción de la BNF junto a la creación de los nodos del AST
////////////////////////////////////////////////////////////////////

TCRL_AST parsear() :
{
  String p;
  String e;
  Token t;
  NodeRules rs;
}
{
  t = <ID> <EOL>
  p = Preamble()
  <BEGINFUNCTION> <EOL>
  rs = Rules()
  e = Epilogue()
  <EOF>
  {
    return new TCRL_AST(t.image, p, e, rs);
  }
}

String Preamble() :
{
  Token t;
}
{
  <PREAMBLE>
  t = <CCODE>
  {
    return t.image;
  }
}
```

```
String Epilogue() :
{
  Token t;
}
{
  <EPILOGUE> t = <CCODE>
  {
    return t.image;
  }
}

NodeRules Rules() :
{
  NodeRule rule;
  NodeRules rules = new NodeRules();
}
{
  (
    rule = Rule() <EOL>
    {
      if (rule instanceof NodeRefinement)
      {
        rules.addRule(((NodeRefinement)rule).getSpecName(),rule);
      }
      else
      {
        rules.addRule(((NodeSynonym)rule).getName(),rule);
      }
    }
    )*
  {
    return rules;
    }
}

NodeRule Rule():
{
  NodeRule r;
}
{
  r = Synonym()
  {
    return r;
  }
  | r = Refinement()
    {
    return r;
  }
}

NodeRefinement Refinement():
{
  Token zn;
  Token tn;
  NodeType ty;
}
{
  zn = <ID> <ARROW> tn = <ID> <COLON> ty = Type()
    {
    return new NodeRefinement(zn.image,tn.image,ty);
  }
}

NodeSynonym Synonym():
{
  Token t;
  NodeType ty;
}
{
  t = <ID> <DEQUAL> ty = Type()
  {
    return new NodeSynonym(t.image, ty);
  }
}

NodeType Type() :
{
  Token t;
  NodeType ty;
}
{
  ty = Array()
  {
    return ty;
  }
  | ty = ZType()
  {
    return ty;
  }
  | ty = CType()
  {
    return ty;
  }
  | ty = Pointer()
  {
    return ty;
  }
  | ty = Record()
  {
    return ty;
  }
  | ty = List()
  {
    return ty;
  }
  | ty = File()
  {
    return ty;
  }
  | ty = DB()
  {
    return ty;
  }
  | ty = RFR()
  {
    return ty;
  }
}

NodeType CType():
{
    Token t;
}
{
    <CTYPE>
    t = <STRING>
    {
    return new NodeCType((t.image).substring(1, (t.image).length() - 1));
  }
}

NodeZType ZType():
{
  Token t;
}
{
  <ZTYPE>
  t = <ID>
  {
    return new NodeZType(t.image);
  }
}

NodePointer Pointer():
{
  NodeType ty;
}
{
  <POINTER> <LBRACKET> ty = Type() <RBRACKET>
  {
    return new NodePointer(ty);
  }
}

NodeArray Array():
{
  NodeType ty;
  Token s;
}
{
  <ARRAY> <LBRACKET> ty = Type() <COMMA> s = <NUMBER> <RBRACKET>
  {
    Integer size = new Integer(s.image);
    return new NodeArray(ty,size.intValue());
  }
}

NodeRecord Record():
{
  List<NodeField> fields = new ArrayList<NodeField>();
  NodeField f;
  Token t;
}
{
  <RECORD> <LBRACKET>
  t = <ID>
  (
    <COMMA>
    f = Element()
    {
      fields.add(f);
    }
  )+
  <RBRACKET>
  {
    return new NodeRecord(t.image,fields);
  }
}

NodeField Element():
{
```

```
  Token n;
  NodeType ty;
}
{
  n = <ID> <COLON> ty = Pointer()
  {
    return new NodeField(n.image,ty);
  }
  | n = <ID> <COLON> ty = CType()
  {
    return new NodeField(n.image,ty);
  }
  | n = <ID> <COLON> ty = Record()
  {
    return new NodeField(n.image,ty);
  }
  | n = <ID> <COLON> ty = Array()
  {
    return new NodeField(n.image,ty);
  }
  | n = <ID> <COLON> ty = ZType()
  {
    return new NodeField(n.image,ty);
  }
  | n = <ID> <COLON> ty = List()
  {
    return new NodeField(n.image,ty);
  }
}

NodeList List():
{
  List<NodeField> fields = new ArrayList<NodeField>();
  Token n, lt, ln, lp=null, mem=null;
  NodeField f;
}
{
  <LIST> <LBRACKET> n=<ID> <COMMA>
  (
    lt=<SLL> <COMMA> ln=<ID>
    | lt=<DLL> <COMMA> ln=<ID> <COMMA> lp=<ID>
    | lt=<CLL> <COMMA> ln=<ID>
    | lt=<DCLL> <COMMA> ln=<ID> <COMMA> lp=<ID>
  )
  (
    <COMMA>
    f=Element()
    {
      fields.add(f);
    }
  )+
  [<COMMA> mem=<STRING>]
  <RBRACKET>
  {
    switch (lt.image)
    {
      case ``SLL'':
        return new NodeSLList(n.image, ln.image, fields,
                              (mem==null)?null:mem.image);
        break;
      case ``DLL'':
        return new NodeDLList(n.image,lt.image,ln.image,
                             ((lp==null)?null:lp.image),
                             fields,
                             (mem==null)?null:mem.image);
        break;
      case ``CLL'':
        return new NodeCLList(n.image, ln.image, fields,
                              (mem==null)?null:mem.image);
        break;
      case ``DCLL'':
        return new NodeDCLList(n.image,lt.image,ln.image,
                               ((lp==null)?null:lp.image),
                               fields,
                               (mem==null)?null:mem.image);
        break;
    }
  }
}

NodeFile File():
{
  Token n, p, d, eol=null, eof=null, s=null;
}
{
  <FILE> <LBRACKET> n=<STRING> <COMMA> p=<STRING> <COMMA> d=<STRING>
  [<COMMA> <ENDOFLINE> eol=<STRING>]
  [<COMMA> <ENDOFFILE> eof=<STRING>]
  [<COMMA> (s=<LINEAR> | s=<RPL> | s=<FPL>)]
  <RBRACKET>
  {
    return new NodeFile(n.image,p.image,d.image,
              ((eol==null)?null:eol.image),
              (eof==null)?null:eof.image,
              (s==null)?null:s.image);
  }
}

NodeDB DB():
{
  Token d,c,t,col;
  String type;
  List<NodeDBColumn> columns = new ArrayList<NodeDBColumn>();
}
{
  <DB> <LBRACKET> d = <ID> <COMMA> c = <ID> <COMMA> t = <ID>
  (
    <COMMA> col =< ID> <COLON> type = colType()
    {
      NodeDBColumn r = new NodeDBColumn(col.image,type);
      columns.add(r);
    }
  )*
  <RBRACKET>
  {
    return new NodeDB(d.image,c.image,t.image,columns);
  }
}

String colType():
{
  Token t;
}
{
  t = <INT>
  {
    return t.image;
  }
  | t = <CHAR>
  {
    return t.image;
  }
  | t = <VARCHAR>
  {
    return t.image;
  }
}

NodeRFR RFR():
{
  Token r,z;
}
{
  <RFR> <LBRACKET> r = <ID> <COMMA> z = <ID> <RBRACKET>
  {
    return new NodeRFR(r.image,z.image);
  }
}
```

Bibliografía

[1] P. Stocks, "Applying formal methods to software testing", Ph.D. dissertation, Department of Computer Science, University of Queensland, 1993.

[2] Sebastian Benz "AspectT: Aspect-Oriented Test Case Instantiation", BMW Car IT GmbH, *Proceedings of the 7th international conference on Aspect-oriented software development*, 2008.

[3] H-M. Hörcher y J. Peleska, "Using formal specifications to support software testing", Software Quality Journal, vol. 4, pp. 309-327, 1995.

[4] 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.

[5] 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.

[6] 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

[7] M. Utting, B. Legeard, *Practical Model-Based Testing. A tools aproach*, Morgan Kaufmann Publisher, 2007.

[8] M. Cristiá, "Catálogo Incompleto de Estilos Arquitectónicos", http://www.fceia.unr.edu.ar/ingsoft/estilos-cat.pdf, 2006.

[9] M. Cristiá, "Testing Funcional basado en Especificaciones Z", http://www.fceia.unr.edu.ar/ingsoft/testing-func-a.pdf, 2007.

[10] M. Cristiá, "Diseño de Software", http://www.fceia.unr.edu.ar/ingsoft/diseno-a.pdf, 2008.

[11] P. Rodríguez Monetti, "Fastest: Automatizando el testing de software", a publicarse, 2009.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Patrones de diseño*. Addison Wesley, 2003.

[13] Community Z Tools, http://czt.sourceforge.net/

[14] Java Compiler Compiler, https://javacc.dev.java.net/

[15] 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.

[16] Shaw, M., Garlan, D., *Software architecture: perspectives on an emerging discipline*, Prentice Hall, Upper Saddle River, 1996.

[17] 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.