Fastest:
Automatizando el testing de software

Pablo Rodríguez Monetti
Director: Prof. MSc. Maximiliano Cristiá
Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario
Argentina

12 de marzo de 2009

Resumen

Aún con el uso creciente de los métodos formales en el desarrollo de software, el testing de software continúa siendo una técnica dominante para verificar y validar sistemas. Con el testing basado en especificaciones, la precisión de las especificaciones formales hace del testing una actividad mucho más sistemática. Este trabajo describe el primer prototipo de Fastest, una herramienta que facilita la derivación de casos de prueba a partir de especificaciones en el lenguaje Z.

Índice general

Agradecimientos

A mi familia, porque han sido mi sustento principal durante la carrera. Desde lo económico y lo afectivo me han apoyado incondicionalmente a lo largo de todos estos años. Siempre confiaron en mí; siempre me dieron total libertad y aliento para tomar decisiones académicas, profesionales y de la vida misma. Muchas gracias, sin ustedes esto no hubiese sido posible.

A las dos personas con las cuales inicié este camino hace ocho años, Federico Olmedo y Diego Llarrull. Dos tipos increíblemente capaces, pero fundamentalmente, personas de una enorme calidad humana. Les agradezco muchísimo por todo lo que compartimos juntos.

Al ambiente académico de LCC en general: a todos los compañeros, a todos los profesores, a todos los alumnos que he tenido. Gracias, porque sin duda aprender fue mucho más fácil con ustedes.

A mi director Maximiliano Cristiá, por haberme dado la oportunidad de trabajar con él y por haber estado siempre que lo he necesitado.

No puedo dejar de mencionar al resto de mis amigos. Con ellos compartí experiencias y momentos inolvidables que nos han unido y que han contribuido en mi formación como persona.

A todos, muchísimas gracias!

Capítulo 1

Introducción

La verificación y la validación del software (V & V) se ha vuelto esencial a medida que la complejidad de los sistemas informáticos ha ido en aumento, y es necesario que ésta se planifique desde el comienzo del ciclo de vida del desarrollo de software. En los últimos 30 a 40 años, el proceso de desarrollo fue evolucionando desde tareas muy sencillas, que involucraban pocas personas, hasta actividades más complejas, llevadas a cabo por recursos humanos mucho más numerosos. Debido a este cambio, la V & V también ha sufrido modificaciones. Previamente, la verificación y validación era un proceso informal, puesto en práctica por el propio programador. Sin embargo, con el crecimiento de la complejidad de los sistemas, se hizo evidente que continuar con este tipo de práctica iba a resultar en la obtención de productos poco confiables. Es así que se volvió necesario mirar a la V & V como una actividad separada en el marco del ciclo de vida del software. La verificación y validación de hoy en día, es considerablemente diferente a la del pasado, y se aplica sobre todo el proceso de desarrollo[1].

En general, la V & V agrupa a todas aquellas actividades necesarias para asegurar que un programa satisfaga su especificación y los requerimientos del usuario. El sistema debe ser verificado y validado en cada etapa del proceso de desarrollo, usando documentos producidos en etapas previas. Como los conceptos de verificación y validación suelen ser confundidos e incluso usados inconsistentemente [1], vale la pena apreciar la diferencia entre ellos de la siguiente manera:

- Validación: ¿se está construyendo el producto correcto?
- Verificación: ¿se está construyendo el producto correctamente?

Es decir, la verificación consiste en chequear que un programa satisface su especificación, mientras que la validación consiste en asegurar que el programa cumpla con lo que se espera de él, lo cual puede no estar reflejado apropiadamente en la especificación. Normalmente, se desea que a la validación la dirija el usuario y a la verificación algún encargado de la parte técnica del desarrollo del software.

Aunque en principio uno puede suponer que el proceso de V & V debe solamente comprobar la corrección del software analizado, en verdad, las cualidades no-funcionales y aquellas que son implícitas,

[1] Lamentablemente, esto es así sólo en la teoría. En la práctica industrial concreta, es usual que el testing no sea realizado como una actividad rigurosa.

también deben ser cubiertas por la validación y verificación del sistema en cuestión. Ejemplos de cualidades no-funcionales son el desempeño y la mantenibilidad. Es bastante frecuente que este tipo de características deban validarse y verificarse en relación a un nivel de aceptación previamente definido y no esperando una respuesta binaria, una respuesta "si o no", en los análisis aplicados [1].

Para satisfacer los objetivos del proceso de V & V, normalmente se sugiere utilizar técnicas de análisis tanto estáticas como dinámicas. Las técnicas estáticas tienen que ver con el análisis del producto o cualquier documentación de diseño relacionada con él, para evaluar su correcto funcionamiento como consecuencia lógica de haber tomado ciertas decisiones de diseño, de representación o de codificación. Éstas técnicas pueden aplicarse en cualquier momento del ciclo de vida del desarrollo. Sin embargo, al no requerir la ejecución del sistema, no pueden asegurar la utilidad operacional del producto. Por otro lado, las técnicas de tipo dinámico, como el testing, se basan en correr el programa para observar su comportamiento y, obviamente, sólo pueden ponerse en práctica si se cuenta con un prototipo o con un ejecutable del sistema de interés. Los dos enfoques son complementarios [1] [2].

Este trabajo se enfocará exclusivamente en la tarea de verificar programas, dejando de lado la actividad de validación de los mismos. En particular, se hará un estudio sobre todo lo concerniente al testing. Sin embargo, quiere dejarse claro que el testing no es el único medio para encontrar errores en el software, y en verdad, se cuenta con diversas técnicas como:

- Prototipado. Es un mecanismo que permite presentar un modelo del producto final, implementando un subconjunto de los requerimientos. Éste modelo será lo que se llama "prototipo" y deberá crearse rápidamente y de manera económica. El prototipo deberá ser evaluado por el cliente del sistema final, para confirmar o proponer cambios en los requerimientos o en el diseño.
- Revisiones o inspecciones. Las revisiones consisten en efectuar lecturas de las distintas representaciones del sistema que fueron confeccionándose en el desarrollo de software. Entre éstas se consideran especificaciones de requerimientos, diagramas de diseño, código fuente, planes de testing, casos de prueba, guías de usuario, etc. Si bien las revisiones pueden hacerse como una actividad manual, para la mayoría de los items que pueden analizarse, existe la posibilidad de automatizar éstas evaluaciones a través de herramientas apropiadas.
- Verificación formal de programas. Es un conjunto de técnicas formales de comprobación en las que partiendo de un conjunto axiomático, reglas de inferencia y algún lenguaje lógico (como la lógica de primer orden, por ejemplo), se puede encontrar una demostración o prueba de corrección de un programa.
- Model checking. Es un método de verificación de sistemas concurrentes de estados finitos, tales como los protocolos de comunicación. Como ventaja sobre los enfoques anteriormente mencionados, se destaca el hecho de que el model checking es automático y usualmente muy rápido. En contrapartida, muchas de las herramientas actuales, basadas en técnicas combinatorias, exploran completamente el espacio de estados posibles del problema, lo que puede conducir a una inmanejable explosión de estados.

A fines de la década del '80, ya se habían difundido los métodos formales como técnicas para especificar y diseñar sistemas de software. En particular, aunque el rol principal de las especifica-

ciones formales era ser la base de pruebas de correctitud y metodologías de transformación rigurosas, comenzó a destacarse la posibilidad y conveniencia de usarlas 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.

Patrick A. Hall, un precursor del testing basado en especificaciones, mostró en [3] y [4] cómo derivar tests desde especificaciones formales Z. Conceptualmente, la idea se basaba en que la especificación de un programa define precisamente los aspectos fundamentales del software, mientras omite la información más detallada y estructural. Su propuesta consistía en realizar particiones simples del espacio de entrada, examinando las divisiones obvias de la entrada definida por los predicados de las operaciones. El análisis de casos que Hall realizaba era altamente estructurado, aunque no riguroso.

Dick y Faivre, en [5], usaron la forma normal disyuntiva (DNF), como la táctica base para dividir el espacio de entrada de operaciones a testear, partiendo de especificaciones VDM.

En su tesis de doctorado [6], Philip Stocks introdujo un marco formal para dirigir el testing basado en especificaciones formales Z, incluyendo un mecanismo que permitía definir y estructurar, de manera bastante rigurosa, los casos de prueba abstractos. Stocks también extendió el trabajo existente acerca de las tácticas de testing, desarrollando dos tácticas nuevas basadas en algunas anteriores. Las tácticas de testing son los mecanismos que permiten dividir el espacio de entrada en clases de prueba, construyendo lo que se conoce como árbol de pruebas.

Hans-Martin Hörcher y Jan Peleska mostraron en [7] cómo la especificación de una operación puede ser usada para derivar sistemáticamente casos de prueba abstractos y evaluar automáticamente los resultados que resultan de testear con ellos, al programa que supuestamente implementa la operación. Este enfoque, al igual que los de Stocks y de Hall, fue puesto en práctica con especificaciones escritas en el lenguaje de especificación Z.

Los autores recién mencionados han contribuido notablemente a la causa de automatizar la actividad de testing. En la práctica industrial, el testing suele hacerse de manera prácticamente artesanal, teniendo personal dedicado a la selección manual de casos de prueba, lo cual resulta ser una tarea tediosa y metódica que desaprovecha el potencial humano.

Esta tesina presenta el primer prototipo de una herramienta, a la que se ha dado el nombre de **Fastest**, y que implementa las ideas de Stocks, Hörcher y Peleska para poder generar casos de prueba abstractos de operaciones especificadas en Z. Fastest ha sido desarrollado como parte del Proyecto Flowx, en las instalaciones de la empresa Flowgate Security Consulting[2]. El Proyecto Flowx, el cual fue financiado en conjunto por Flowgate y FONTAR (Fondo Tecnológico Argentino) tiene como principal objetivo el implementar políticas de seguridad multinivel en el núcleo del sistema operativo Linux, y es una de las aplicaciones pensadas para Fastest la verificación de tal implementación.

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

El resto de este informe se divide en siete capítulos, donde se desarrollan los conceptos necesarios para comprender por completo todos los aspectos del prototipo de Fastest. El capítulo 2 presentará las ideas generales de la actividad de testing, como una etapa fundamental del ciclo de vida del software. El capítulo 3 explicará, en particular, los conceptos relacionados al testing funcional basado en especificaciones formales Z. A continuación, el capítulo 4, hará un resumen del estado del arte de las herramientas que automatizan de alguna manera este tipo de actividad. Al pasar al capítulo 5, el lector se encontrará con una descripción de Fastest, incluyendo detalles de su arquitectura, diseño y tecnología empleada en su desarrollo. El capítulo 6 presenta la distribución del prototipo que acompaña a este informe y un manual de usuario para la correcta utilización del mismo. Un caso de estudio es mostrado en el capítulo 7 y, por último, el capítulo 8 contiene la conclusión de esta tesina y sugiere trabajos futuros a realizar en el área.

Capítulo 2

La técnica de testing

El presente capítulo presentará un repaso breve y general sobre algunos aspectos del testing de programas. Se hará un recorrido sobre su significado, sus características más salientes, cualidades esperadas y otras cuestiones relacionadas.

2.1. ¿Qué se entiende por testing?

Una forma muy natural de verificar que algo funciona es simplemente ponerlo en operación en algunas situaciones representativas y comprobar si su comportamiento es el esperado [1]. En general, será imposible realizar esto en todas las condiciones de funcionamiento posibles[1] con lo cual es necesario encontrar *casos de prueba* o *casos de test* que aporten evidencia suficiente para suponer, en forma razonable, que el comportamiento en todas las demás condiciones de funcionamiento será el requerido. Entonces, se puede considerar ahora la siguiente definición, dada en [9]:

> El testing es la verificación *dinámica* del comportamiento de un programa contra el comportamiento *esperado* de ese mismo programa, utilizando un conjunto *finito* de casos de prueba, seleccionados apropiadamente del dominio de ejecución, usualmente infinito.

Como ya se vio en la introducción de esta tesina, la característica dinámica del testing significa que al sistema bajo prueba hay que analizarlo al ponerlo en funcionamiento, al ejecutarlo con argumentos o datos de entrada específicos que permitan encontrar fallas en su comportamiento.

El procedimiento de testing, a grandes rasgos, consistirá en tomar elementos del dominio de entrada de un programa, ejecutar el programa en estos casos de prueba y comparar la salida o estado final con la salida o estado final esperados. En la literatura sobre testing se presupone la existencia de lo que se llama un *oráculo*, que es algún tipo de método para determinar que una salida dada es la salida esperada. Éste oráculo puede ser incluso un humano; por ejemplo, el futuro usuario del sistema. En general, el oráculo más confiable lo constituirá una especificación formal de sistema a testear, como se verá en el capítulo 3.

[1] A esta tarea de realizar pruebas en todos los casos posibles se la denomina *testing exhaustivo*.

Dado que usualmente se tiene un conjunto muy grande, o incluso infinito, de posibles casos de prueba, pero sólo se puede ejecutar una fracción muy pequeña de ellos, una cuestión fundamental del testing radica en la elección de los casos que tengan más posibilidades de exponer las fallas del sistema. Es aquí donde se pone de manifiesto la importancia que tienen la experiencia y las habilidades del encargado de testing: en el aprovechamiento de lo que conoce sobre el sistema para identificar los conjuntos de casos de prueba que producen el mismo comportamiento y los que producen distinto comportamiento.

Antes de continuar, vale la pena aclarar algunos términos básicos:

- Una *falla (failure)* es una situación que manifiesta un comportamiento no deseado y, típicamente, ocurre cuando el sistema a testear se pone en ejecución.
- Un *defecto (fault)* es una porción de software incorrecto[2] que causa una falla. Si el sistema no puede fallar, entonces no tiene defectos. Por el contrario, si uno observa una falla, lo que debe hacer es intentar encontrar el defecto que produjo la falla y tratar de corregirlo. De todas maneras, nunca se puede probar que un sistema no puede fallar, sólo se puede probar que contiene defectos.
- Un *error (error, mistake)* es una acción humana que resulta en un software con algún defecto. Un error puede llevar a incluir un defecto en el sistema, haciéndolo fallar.

Entonces, el *testing* es la actividad de ejecutar un sistema con el fin de encontrar defectos. Es por esto, porque intenta mostrar que algo es incorrecto, que es un enfoque apropiado el considerar al testing como un proceso destructivo. El encargado de testing debe adoptar una actitud destructiva hacia el programa, debe querer que falle, debe esperar que falle y debe concentrarse en encontrar casos de prueba que muestren sus fallas.

2.2. Objetivos del testing

Un aspecto que hay que tener muy en cuenta a la hora de hablar de esta tarea es que, como bien reza la frase de Dijkstra, **el testing puede ser usado para mostrar la presencia de defectos en el software, pero nunca para mostrar su ausencia**. Esta idea busca sintetizar los objetivos de la actividad: así como se puede asegurar que un sistema no es correcto sólo con encontrar un caso en que el resultado no sea el esperado, no basta con tener un correcto funcionamiento del sistema en un número finito de casos para afirmar su corrección en cualquier situación. De esta manera, es que hay que apreciar al testing como sólo uno de los medios que existen (aunque sea el más popular, y de hecho uno muy útil) para verificar software, pero nunca como un proceso que ofrezca garantía absoluta de detección de errores.

Además de mostrar la presencia de defectos, el testing debe ayudar a *localizarlos*. No bastará con testear una pieza de software y obtener la respuesta binaria que indica si la pieza contiene un error o no. Es decir, ante el hallazgo de un error, el proceso de testing debe brindar información útil sobre

[2]Puede encontrarse en la especificación, en el diseño, o el código del sistema.

la localización del mismo, para que después pueda ser utilizada por el proceso de *debugging*, el que intentará corregir el software. En definitiva, el testing tendrá verdadero sentido sólo si después de haberse realizado y de haber detectado un error, hay alguna posibilidad de reparar la situación [1].

2.3. Testing no es debugging

Como se vio en la sección anterior, el testing de sofware debe interactuar con el *debugging* del mismo. Sin embargo, aunque suelen considerarse parte del mismo proceso, vale remarcar que el testing y el debugging son actividades bien distintas. Mientras que el testing establece la existencia de defectos, el debugging está centrado en identificar y corregir esos defectos. Es decir, el testing y el debugging no son lo mismo, pero es claro que están íntimamente relacionados.

Otra diferencia entre el testing y el debugging es que es conveniente que la primera actividad sea realizada por una persona, o grupo, externo a la organización de desarrollo, mientras que la segunda debe ser realizada necesariamente por alguien interno, que conozca la arquitectura, el diseño y la implementación con mayor profundidad. Que se recomiende que el testing sea llevado a cabo por personas de otra organización se debe principalmente a cuestiones psicológicas y tiene relación con la característica destructiva del testing, ya mencionada anteriormente. Más precisamente, el autor de un programa o sistema tiende a cometer los mismos errores al ponerlo a prueba, es decir, debido a que es SU programa, inconcientemente tiende a hacer casos de prueba que no hagan fallar al mismo. Además, puede llegar a comparar mal el resultado esperado con el resultado obtenido debido al deseo de que el programa pase las pruebas.

En base a esto, se pueden considerar cuatro niveles de independencia entre el testing y el desarrollo de un sistema:

- Tests diseñados por las personas que codificaron el software a testear.
- Tests diseñados por personas distintas pero del equipo de desarrollo.
- Tests diseñados por personas de otro grupo de la organización (área de testing)
- Tests diseñados por personas de otra organización (tercerización)

2.4. Testing estructural vs. testing funcional

Estos son los dos enfoques que se pueden encontrar si se intenta clasificar a la actividad de testing de acuerdo al tipo de información utilizada para guiarla. Por un lado, el *testing funcional* o también llamado *de caja negra (black-box testing)* realiza el testeo de una pieza de software ignorando por completo cómo está construida internamente y sólo mirando su especificación. Se puede pensar que ve al software bajo prueba como si fuera una función, pasándole argumentos o datos de entrada y observando los valores devueltos, sin detenerse a analizar cómo computa dichos valores (de allí la denominación de funcional).

Por otro lado, el *testing estructural* o *de caja blanca (white-box testing)* sí utiliza información acerca de la estructura interna del programa (de su código fuente), pudiendo incluso ignorar su especificación. Se dice que el testing estructural comprueba lo que el programa *hace*, mientras que el testing funcional comprueba lo que *se supone que hace* (lo que debería hacer de acuerdo a la especificación) [1].

Algunas características del testing estructural

- Requiere que haya finalizado la fase de codificación para que pueda empezarse a testear.
- Si se cambia la estructura del código, deben recalcularse los casos de prueba.

Algunas características del testing funcional

- No hace falta que haya comenzado la fase de implementación para poder empezar con la fase de testing.
- Si el programa es modificado, gran parte del esfuerzo de testing sigue siendo útil.

Aunque parezca que poner en práctica el testing funcional sea más conveniente que hacerlo con el testing estructural, lo cierto es que ambos enfoques son útiles y complementarios. Es bastante común que se utilicen técnicas de testing funcional para encontrar pruebas funcionales y de robustez. Luego se pueden utilizar métricas del testing estructural para chequear qué partes de la implementación no han sido evaluadas correctamente y así obtener nuevas pruebas a partir de esos casos.

Este trabajo se enfocará en un tipo particular de testing funcional, que es el testing funcional basado en especificaciones formales. Como su nombre lo indica, esta actividad se basa en utilizar una especificación formal del sistema bajo prueba, y es el próximo capítulo donde se describe más en detalle.

2.5. *Testing in the Large*

En la sección anterior se mencionaron dos grandes enfoques que existen para llevar a cabo el proceso de testing. Sin embargo, no puede dejar de mencionarse que los dos son de aplicación práctica sólo en el testing de módulos individuales, es decir, son aplicables para hacer lo que se denomina *Testing in the Small*. Pero al someter a prueba a un sistema de software completo, se requerirá realizar el proceso de testing de todo un programa o de una serie de programas que deben verificar, en conjunto, un cierto comportamiento esperado. Es lo que se llama *Testing in the Large*. En ese caso, las técnicas mencionadas podrían sufrir de una explosión combinatoria que las volverían inaplicables [1].

De todas maneras, los mecanismos de testing estructural y de testing funcional continúan siendo útiles en sistemas complejos. La clave es organizar la actividad de testing de la misma manera que se organiza la actividad de diseño del software en cuestión: dividiendo el problema a tratar en problemas más pequeños. La posibilidad más natural de poner en práctica esta idea es guiar el proceso de testing a partir de la estructura modular del sistema. De esta forma, se podrían ir testeando los módulos separadamente, uno por uno, hasta eventualmente poder testear el sistema completo [1].

En general, el proceso de testing se separa en tres etapas: el *testing de unidad*, el *testing de integración* y el *testing de sistema*. A medida que se van encontrando errores en una etapa, se va a requerir modificar el programa para corregir estos errores, lo que probablemente va a necesitar que se repitan algunas de las etapas ya realizadas. Así, puede apreciarse que el proceso de testing raramente es secuencial; lo normal es que sea iterativo.

Testing de unidad. El testing de unidad consiste en poner a prueba las menores unidades (módulos) separables de un sistema. Cuando se dice que sea separable, se entenderá que esa unidad pueda ser ejecutada en forma aislada del resto del sistema. Esto no significa que la unidad no interactúe con otras, sino que ésta pueda ser testeada, siempre que sus interfaces estén correctamente definidas, aún cuando aquellas no estén implementadas completamente. Si bien parece imposible hacer uso de módulos para los cuales no se ha terminado su codificación, el hecho de que tengan sus secretos bien ocultos detrás de una interfaz, permite utilizarlos con una implementación básica, que provea un comportamiento altamente simplificado. A este tipo de módulos se los llama normalmente *stubs*.

Testing de integración Esta fase consiste en el testeo de colecciones de módulos que han sido integrados en subsistemas. Para llevarla a cabo, hay distintas estrategias, como por ejemplo, *top-down* y *bottom-up*. La primera de ellas testea los componentes más abstractos antes de testear los más específicos mientras que la segunda es la contraria, primero se comienza testeando los módulos del nivel más bajo de la jerarquía y se va trabajando hacia arriba por esta jerarquía hasta testear el módulo más general. Sin embargo, sea cual sea la estrategia que se adopte, lo más conveniente es utilizar un enfoque incremental, donde se vayan realizando las pruebas agregando los módulos de a uno, siempre testeando cada incremento antes que se agregue un nuevo incremento al sistema. Esto hace posible encontrar errores con mayor facilidad y no tener que esperar a que todos los módulos hayan sido implementados para comenzar a testear el sistema.

Testing de sistema Pretende verificar si el conjunto completo de módulos de un sistema consigue el comportamiento adecuado del mismo, posiblemente asumiendo que sus módulos constituyentes proveen la funcionalidad esperada.

El presente trabajo no hace un aporte directo a las técnicas de testing de integración y de sistema, y solo se concentra en el testing de unidad de programas. Sin embargo, es cierto que podría interpretarse como un aporte indirecto a esas áreas. Esto es en un sentido mencionado anteriormente, cuando se lleve a cabo el *Testing in the Large* mediante una división en módulos y una subsecuente aplicación, en esos módulos, de las técnicas aquí desarrolladas.

Capítulo 3

El testing basado en especificaciones formales Z

Como se vio en el capítulo anterior, un programa es correcto si verifica su especificación. La especificación puede ser formal, semi-formal, informal, o incluso puede no estar escrita y que sólo la conozca el programador o el usuario. El resto de este trabajo asumirá que existe una especificación formal Z del sistema a ser verificado y el presente capítulo explicará las características del testing funcional bajo tal hipótesis, no sin antes hacer una breve introducción al lenguaje de especificación Z.

3.1. El lenguaje de especificación Z

En esta tesina se utilizará la notación Z, un lenguaje de especificación formal usado para describir y modelar sistemas de software [10]. Z fue desarrollado en 1974 por J. R. Abrial y el Programming Research Group de la Universidad de Oxford. Está basado en la teoría de conjuntos, la lógica de primer orden y el cálculo lambda. Todas las expresiones en Z son tipadas, por lo que se evitan algunas paradojas de la teoría de conjuntos simple. El lenguaje contiene un catálogo estandarizado (llamado *mathematical toolkit*) con las funciones matemáticas y predicados usados frecuentemente.

Un modelo Z permite especificar máquinas de estados, las cuales son representaciones idénticas a las máquinas de estados finitos, con la salvedad de que la cantidad de estados en cada una de ellas puede ser infinita. Para lograr definir una de estas máquinas, Z debe especificar su conjunto de estados y sus operaciones, las cuales determinan las transiciones posibles entre los distintos estados. La descripción de los estados se realiza a través de la definición de los llamados *esquemas de estado*, los cuales contienen las declaraciones de las variables de estado de la máquina. Por otro lado, con el objeto de definir las operaciones existentes, deben darse uno o varios *esquemas de operación* para cada una de ellas. En estos esquemas se especifican las transiciones de estados posibles en la máquina, a través de predicados que relacionan las variables de estado con variables de entrada y de salida propias de la operación. [11]

En el resto de la tesina se asumirá que el lector está familiarizado con los conceptos de conjuntos, relaciones y cálculo de predicados. En el Apéndice A se hace un repaso de las características más

importantes del lenguaje Z. Allí también se remarcan aquellos elementos sintácticos que aún no están soportadas en la versión de la herramienta que se presenta en esta tesina.

3.2. Formalizando el testing funcional basado en especificaciones formales

Para explicar algunos conceptos generales del testing funcional basado en especificaciones formales, en esta sección se formalizarán algunas ideas relacionadas a él. Sencillamente, se tomará parte de la formalización dada en [1] y en [12].

En lo sucesivo, se verá a un programa o subrutina como una función

$$P : ID \rightarrow OD$$

donde a *ID* se lo llamará *dominio de entrada* del programa y a *OD dominio de salida*. El dominio de entrada estará dado por el producto cartesiano de los tipos[1] de las variables de entrada del programa y el dominio de salida por lo análogo para las variables de salida. Vale la pena observar que la definición podría no ser la más adecuada para tratar con cierto tipo de software. Por ejemplo, en el testing de interfaces gráficas de usuario, *GUI testing*, el concepto de variables de entrada no resulta del todo claro.

De todas formas, en el caso general, con "variables de entrada del programa" se entenderá a todas aquellas entradas que aparecen explícitamente en el código del mismo (dejando de lado otra información abstracta que pueda estar relacionada), lo que incluye a archivos, parámetros recibidos, datos leídos desde el entorno, etc. Análogamente, con "variables de salida" se hará referencia a todas aquellas salidas explícitas del programa, como ser salidas por cualquier dispositivo, parámetros o valores de retorno e incluso errores tales como no terminación, etc.

De esta manera, dado un programa P con dominio de entrada ID, un caso de prueba para P será un elemento $x \in ID$, y testear P con x se hará simplemente calculando $P(x)$ y comparando su valor con el resultado esperado. En el mismo sentido, un conjunto de prueba T para P será cualquier conjunto de casos de prueba que sea subconjunto finito de D. Testear P con T significa comprobar que $P(x)$ es el esperado para cada $x \in T$.

En particular, se dirá que un caso de prueba para P es *exitoso* si al testear P con x no se obtiene el resultado esperado, con lo cual el caso de prueba estaría descubriendo un error en el programa. Pero para poder juzgar si el comportamiento del programa es el adecuado o no, se deberá contar con una especificación del mismo, la cual lo describirá de una manera diferente. Es la especificación del programa la que tendrá que vincular los casos de prueba con los resultados obtenidos y esperados, para concluir si existe una correspondencia entre los primeros y los últimos. De esta forma, se verá a la especificación Z de un programa[2], también como una función, que va desde el espacio definido por

[1] Con "tipo" no se hace referencia a un tipo de un lenguaje de programación, ya que un programa a ser verificado podría estar escrito en un lenguaje no tipado y aún así poder ser testeado.

[2] En verdad, a un esquema de operación contenido en una especificación Z de un programa.

la declaración de las variables de entrada y de estado hasta el espacio dado por las variables de salida y de estado de la operación. Es decir, una especificación *Op* será vista como una función de *IS* en *OS*, donde *IS* se llama *espacio de entrada* y *OS* se llama *espacio de salida* y se definen de la siguiente manera:

$$IS == [v?_1 : T_1, ..., v?_a : T_a, s_1 : T_{a+1}, ..., s_b : T_{a+b}]$$
$$OS == [v!_1 : U_1, ..., v!_c : T_c, s_1 : T_{a+1}, ..., s_b : T_{a+b}]$$

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

Pero aunque el estado y cierta entrada sean parte del espacio de entrada de una operación, puede que para ellos no esté especificado el comportamiento de una operación. En otras palabras, una operación puede no relacionar cada elemento de su espacio de entrada con algún elemento de su espacio de salida. Por ejemplo, dado el siguiente esquema de operación:

Decrement

$$n : \mathbb{N}$$
$$n' : \mathbb{N}$$

$$n' = n - 1$$

y al verlo como una función de $IS == [n : \mathbb{N}]$ en $OS == [n : \mathbb{N}]$, se puede apreciar que la misma no está definida cuando $n = 0$.

Es por esto que se introduce la noción de *espacio válido de entrada* (*VIS*) como el subconjunto del espacio de entrada para el cual la operación está definida. Más formalmente, se puede definir al espacio válido de entrada de la especificación *Op*, VIS_{Op}, como el subconjunto de *IS* que satisface la precondición de *Op*:

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

y de esta manera ver a *Op* como la siguiente función total:

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

Para poder formalizar lo que se entiende por caso de prueba exitoso será necesario relacionar de alguna manera un programa con su especificación. Es decir, habrá que establecer una forma de vincular los elementos de *ID* con los *VIS* y los de *OD* con los de *OS*, Para tal fin se introducen las siguientes funciones:

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

donde *ref* y *abs* son las abreviaturas de *función de refinamiento* y *función de abstracción*, respectivamente. Se puede decir que la primera de ellas refina un elemento a nivel de especificación en un elemento a nivel de implementación mientras que la segunda hace la transformación inversa: dado un

elemento a nivel de implementación, permite obtener su correspondiente elemento a nivel de especificación. La razón por la que se formaliza la función de abstracción como una de tipo parcial es que no es siempre posible encontrar un elemento abstracto para cada salida de un programa. Este es el caso de una terminación abrupta e inesperada, donde por lo general no se puede realizar una abstracción de este resultado a nivel de especificación, ya que no es algo que se suela especificar. De cualquier manera, que en situaciones como esas no pueda aplicarse la función *abs* no resulta ser un inconveniente ya que, en definitiva, en ellas se ha alcanzado el objetivo principal del testing: encontrar errores en el programa.

Sea $P : ID \rightarrow OD$ un programa tal que su especificación Z es $Op : VIS_{Op} \rightarrow OS$ y sean $t \in VIS_{Op}$ y $x = ref_P^{Op}(t)$. Se dirá que x es un caso de prueba exitoso para P sí y sólo si $Op(t) \neq abs_P^{Op}(P(x))$, lo que intuitivamente describe lo expuesto anteriormente: que la salida obtenida de ejecutar el caso de prueba no coincide con lo que la especificación esperaba.

3.3. Las etapas del testing funcional

Para poner a prueba un programa P, teniendo su especificación Op, se deberá entonces aplicar iterativamente el proceso de testing que consta de las siguientes etapas:

- Generación de un caso de prueba (*gen*)
- Refinamiento del caso de prueba (*ref*)
- Ejecución del caso de prueba (*ejec*)
- Abstracción de la salida (*abs*)
- Comprobación (*compr*)

y que se ilustra en la Figura 3.1. En el diagrama se puede observar que partiendo de un programa P y de la especificación Op de P, y mediante las etapas antes mencionadas, se puede alcanzar un resultado, *res*, que indica si el proceso en cuestión fue exitoso o no, es decir, si encontró un error en P o no. La primer etapa, *gen*, es la que genera, a partir de la especificación Op, un caso de prueba a nivel de especificación (que será llamado *caso de prueba abstracto*), t. La segunda etapa, *ref*, es la que se encarga de transformar ese caso de prueba abstracto en un caso de prueba a nivel de implementación (que será llamado *caso de prueba concreto*), x. A continuación es necesario ejecutar x en P, con lo cual el paso *ejec*, representado en la figura por dos flechas, permite obtener la salida $P(x)$ a nivel de implementación, llamada *salida concreta*. La etapa siguiente, *abs*, es la que hace posible abstraer esta salida al nivel de especificación (que como es de esperar, se llamará *salida abstracta*), resultando en y. Por último, a través de *compr* y utilizando Op, el caso de prueba abstracto t y la salida abstracta y, se comprueba si y se corresponde con t según Op, obteniendo el resultado *res*. *res* es una respuesta binaria que indica si existe o no tal correspondencia. Si no existe, significa que se ha encontrado un error en P.

La mayoría de las etapas del proceso de testing recién mostradas son automatizables en gran medida. El objetivo de esta tesina es, justamente, mostrar cómo es posible hacer esto principalmente

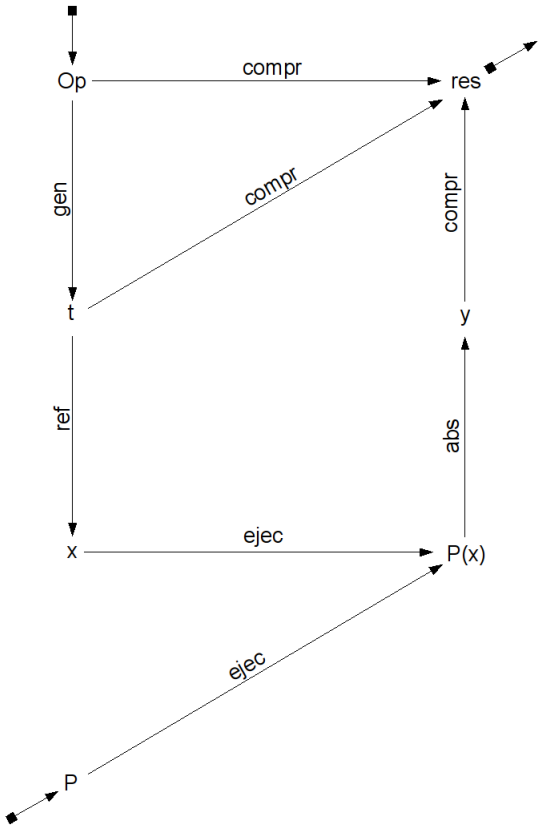


Figura 3.1: El proceso de testing funcional basado en especificaciones formales.

con la etapa denominada *gen*, es decir, cómo pueden generarse de forma automática casos de prueba abstractos partiendo de una especificación Z.

3.4. Generando casos de prueba abstractos

En esta sección, y en las que siguen en este capítulo, se describirá con más detalle cómo puede llevarse a cabo la generación de casos de prueba abstractos. Para esto, se tendrán en cuenta los trabajos de Hall [3] [4], Stocks [6], Hörcher y Peleska [7] y Stocks y Carrington [8], que mostraron que las especificaciones formales Z pueden ser usadas para derivar sistemáticamente casos de prueba abstractos. Conceptualmente, la idea se basa en que una especificación define precisamente los aspectos fundamentales del software que describe, omitiendo información más detallada, estructural, y de implementación. Estos aspectos fundamentales son, más precisamente, las alternativas funcionales del software que se tuvieron, o mejor dicho, debieron tenerse en cuenta a la hora de implementarlo. En general, el diseñar casos de prueba para testing, lo que puede considerarse un arte, es justamente identificar estas alternativas funcionales para después poder testear el programa sobre ellas.

Las alternativas funcionales de un sistema pueden expresarse como restricciones sobre las variables de entrada y de estado definidas en su especificación. Las técnicas propuestas por los autores anteriormente mencionados, entonces, sugieren modelar estas alternativas como esquemas Z a los que se les denominará *clases de prueba*. Por ejemplo,

$$ClaseDePrueba == [a, b : \mathbb{Z} \mid a < b]$$

define una clase de prueba con dos valores, a y b, tales que a es menor que b. En este contexto, puede apreciarse que *ClaseDePrueba* no es más que un conjunto (infinito) de casos de prueba definido por comprensión. $[a, b : \mathbb{Z} \mid a = 4 \land b = 10]$ y $[a, b : \mathbb{Z} \mid a = -10 \land b = 5]$ son ejemplos de casos de prueba pertenecientes a *ClaseDePrueba*.

El proceso a través del cual se generan casos de prueba abstractos, partiendo de la especificación Z de una operación *Op*, se dividirá en los dos siguientes pasos:

- Generación de un árbol de pruebas para *Op*
- Selección de casos de prueba

El primer paso consiste en construir una jerarquía entre clases de prueba, que relacionará a las mismas y que se llamará *árbol de pruebas*. Para tal fin, la propuesta es dividir iterativamente a las clases de prueba en clases de prueba menores usando mecanismos que se denominarán *tácticas de testing*[3]. El segundo paso, por su parte, consiste en elegir uno o más casos de prueba para cada una de las clases de prueba que se hayan generado en el paso anterior, que no contengan a ninguna de las otras clases de prueba, y que no tengan su predicado equivalente a *false*. Se explicará la generación de árboles de pruebas en la Sección 3.5 y la selección de casos de prueba en la Sección 3.6.

[3]En la bibliografía donde se expone esta metodología se utiliza el término estrategia en lugar de táctica. En este trabajo, al igual que se hace en [12], se empleará el segundo de ellos.

3.5. Construcción de árboles de pruebas

Dado que todos los casos de prueba de una operación deben ser tomados del espacio válido de entrada de la misma, éste es un buen punto de partida para construir el árbol de pruebas. Una vez determinado el *VIS*, hay que dividirlo, aplicando alguna táctica de testing, en una cantidad de nuevas clases de prueba, que posiblemente constituyan una partición de él[4]. Estas clases conformarán el primer nivel de nodos del árbol de pruebas. El objetivo es derivar conjuntos de casos de prueba que sean clases de equivalencia respecto a la posibilidad de encontrar un error en la unidad funcional analizada, y que además cubran el *VIS*. En otras palabras, se pretende encontrar clases tales que cada elemento en ellas, tenga la misma posibilidad de encontrar un error que los demás elementos de la misma clase. Es por esto que la aplicación de tácticas de testing debe realizarse repetidamente, obteniendo nuevas y nuevas clases de prueba que completen el árbol, hasta que se considere que las clases de prueba "hoja" (es decir, aquellas que no tengan nodos hijos en el árbol) representen todas las alternativas funcionales importantes de la operación en cuestión.

El árbol de pruebas asociado a una operación queda entonces representado como un grafo, donde los nodos son las clases de prueba (en particular, la raíz es el espacio válido de entrada de la operación) y las aristas son las tácticas de testing aplicadas en las derivaciones de nuevas clases de prueba. Típicamente, un árbol de pruebas se ve como el mostrado en la Figura 3.2. En ella se muestra cómo, en primer lugar, a partir de la aplicación de una táctica $Tac1$ al *VIS* de la operación Op, VIS_{Op}, se obtienen las n clases de prueba Op_i^{Tac1}, para i entre 1 y n. Luego se utiliza la táctica $Tac2$ sobre la clase de prueba Op_1^{Tac1} para obtener las r clases de prueba Op_i^{Tac2}, para i entre 1 y r. De la misma forma la táctica $Tac3$ permite dividir a la clase de prueba Op_n^{Tac1} en Op_i^{Tac3}, para i entre 1 y q. Por último, la táctica $Tac4$ genera las s clases de prueba Op_i^{Tac4} con i entre 1 y s, cuando es aplicada a Op_r^{Tac2}.

Es importante remarcar que el proceso de generación de clases de prueba no es exclusivo del lenguaje Z. El mismo puede aplicarse sobre cualquier lenguaje de especificación basado en lógica, como TLA o B, y puede modificarse apropiadamente, según [13] y [14], para ser usado en Statecharts y CSP, respectivamente. A continuación se describirá una serie de tácticas de testing y se presentará la manera en que estas pueden aplicarse sucesivamente para poder construir árboles de pruebas.

3.5.1. Tácticas de testing

Esta subsección muestra a través de un ejemplo algunas de las tácticas de testing que se conocen y que fueron propuestas en [6] y en [8]. En particular, se explicarán aquellas que fueron implementadas por el prototipo de Fastest que se describe en esta tesina. Justamente, es notable que tanto la aplicación de éstas como de otras tácticas puede ser automatizada en gran medida, no dependiendo en absoluto de la intervención manual de un encargado de testing.

Para realizar la presentación de las tácticas de testing se introduce la siguiente especificación Z, la cual describe una versión simplificada de la lectura de archivos en UNIX. El contenido de los archivos

[4]La división de una clase de prueba en otras clases de prueba, a través de la aplicación de una táctica de testing, podría resultar en clases con algunas de ellas solapadas entre sí, por lo que no formarían una partición de la original.

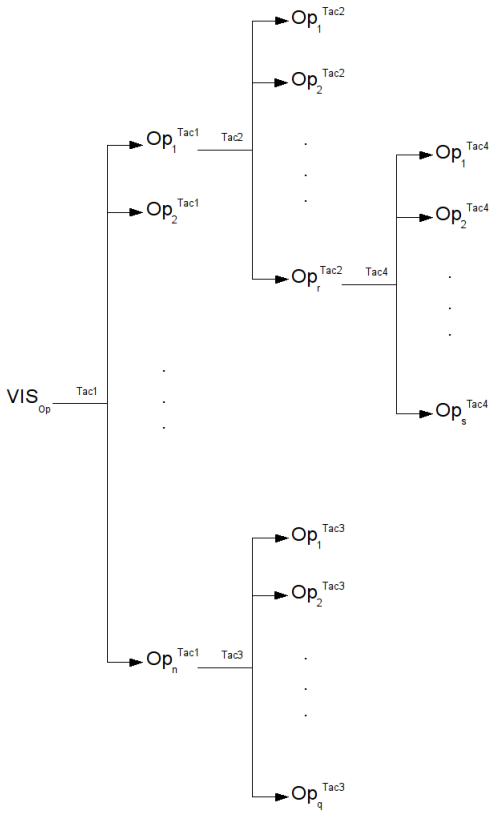


Figura 3.2: Estructura típica de un árbol de pruebas.

se modelará como una secuencia de bytes. A su vez, los bytes serán introducidos con el siguiente tipo básico, sin mayores detalles internos:

$[BYTE]$

Cada archivo tendrá asociado un permiso de acceso, que podrá ser prohibido (si es un archivo sobre el que no se puede leer ni escribir), de sólo lectura, o de lectura y escritura. Los mismos se representan a través del siguiente tipo enumerado:

$PERMISO ::= prohibido \mid solo\_lectura \mid lectura\_escritura$

Archivo

$$datos : \operatorname{seq} BYTE$$
$$permiso : PERMISO$$

El invariante de estado de un archivo se describe en el siguiente esquema:

InvarianteArchivo

$Archivo$

$$\#datos \leq 255$$

La operación de lectura toma el valor de una longitud como entrada y lee esa cantidad de caracteres desde el comienzo del archivo. Se distinguirán cuatro posibles situaciones respecto a la operación de lectura, *Leer*:

- Una lectura exitosa, donde también se incluye el caso, teniendo o no permiso de lectura, de solicitar la lectura de 0 bytes del archivo. En el caso general es necesario tener al menos permiso de lectura.
- Un intento de lectura fallido por querer hacerse sobre un archivo vacío.
- Un intento de lectura fallido por querer hacerse hasta más allá del final del archivo.
- Un intento de lectura fallido por querer hacerse sobre un archivo cuyo permiso de acceso es prohibido.

las que al ejecutar la operación serán reportadas, según corresponda, con los siguientes mensajes:

$REPORTE ::= ok \mid error\_vacio \mid error\_fin\_archivo \mid error\_acceso$

En un primer esquema se captura la situación de una lectura con éxito. Puede notarse que el predicado de *LeerOk* contiene al operador $\Rightarrow$, el cual no es muy habitual a la hora de escribir predicados de esquemas Z. Sin embargo, se lo ha incluido por cuestiones didácticas, para favorecer más adelante la presentación de una de las tácticas de testing.

LeerOk

$\Xi Archivo$
$longitud? : \mathbb{N}$
$lectura! : \text{seq}\, BYTE$
$rep! : REPORTE$

$\#datos > 0$
$longitud? \leq \#datos$
$longitud \neq 0 \Rightarrow permiso \neq prohibido$
$lectura! = (1..longitud?) \lhd datos$
$rep! = ok$

Los tres esquemas que siguen definen las situaciones erróneas anteriormente mencionadas.

ArchivoVacio

$\Xi Archivo$
$lectura! : \text{seq}\, BYTE$
$rep! : REPORTE$

$\#datos = 0$
$lectura! = \langle\rangle$
$rep! = error\_vacio$

FinDeArchivoEncontrado

$\Xi Archivo$
$longitud? : \mathbb{N}$
$lectura! : \text{seq}\, BYTE$
$rep! : REPORTE$

$longitud? > \#datos$
$lectura! = \langle\rangle$
$rep! = error\_fin\_archivo$

ArchivoProhibido

$\Xi Archivo$
$longitud? : \mathbb{N}$
$lectura! : \text{seq}\, BYTE$
$rep! : REPORTE$

$\neg\, (longitud? \neq 0 \Rightarrow permiso \neq prohibido)$
$lectura! = \langle\rangle$
$rep! = error\_acceso$

La operación cuya implementación se quiere testear, *Leer*, no altera el contenido de los archivos, lo que se ve expresado por la inclusión de $\Xi Archivo$ en los cuatro esquemas presentados. Finalmente, se define *Leer* como la disyunción de tales esquemas:

$$Leer == LeerOk \lor ArchivoVacio \lor FinDeArchivoEncontrado \lor ArchivoProhibido$$

Para ilustrar el significado de esta combinación de esquemas, se muestra el esquema *Leer* expandido completamente. Esta expansión no es más que una manipulación sintáctica de los esquemas que conforman a *Leer*. Si bien el predicado resultante podría contener información redundante o bien podría simplificarse, no es algo que resulte de interés en este momento.

La versión completamente expandida de *Leer* es:

Leer

$$
\begin{array}{l}
datos, datos' : \text{seq}\, BYTE \\
permiso, permiso' : PERMISO \\
longitud? : \mathbb{N} \\
lectura! : \text{seq}\, BYTE \\
rep! : REPORTE \\
\hline
datos' = datos \\
permiso' = permiso \\
(\#datos > 0 \land longitud? \leq \#datos \land (longitud \neq 0 \Rightarrow permiso \neq prohibido) \land \\
lectura! = (1..longitud?) \lhd datos \land rep! = ok) \lor \\
(\#datos = 0 \land lectura! = \langle\rangle \land rep! = error\_vacio) \lor \\
(longitud? > \#datos \land lectura! = \langle\rangle \land rep! = error\_fin\_archivo) \lor \\
(\neg\, (longitud? \neq 0 \Rightarrow permiso \neq prohibido) \land lectura! = \langle\rangle \land rep! = error\_acceso)
\end{array}
$$

3.5.1.1. Obteniendo el *VIS* de la operación

Como ya se vio al comienzo de la Sección 3.5, el primer paso para generar casos de prueba para una operación es identificar su espacio válido de entrada. En el caso de *Leer*, puede probarse que su precondición es equivalente a *true*, por lo que su *VIS* define el mismo espacio que su *IS*, el que a su vez está dado por:

$$IS_{Leer} == [datos : \text{seq}\, BYTE;\ permiso : PERMISO;\ longitud? : \mathbb{N}]$$

En consecuencia, VIS_{Leer} es simplemente el esquema formado por la declaración de variables de entrada y de estado iniciales que aparecen en el esquema de operación *Leer*.

Como ejemplo de una operación donde su *IS* y su *VIS* no coinciden, se tiene a *LeerOk*. En tal caso se tendrían:

$$IS_{LeerOk} == [datos : \text{seq}\, BYTE;\ permiso : PERMISO;\ longitud? : \mathbb{N}]$$

VIS_{LeerOk}

$datos : \text{seq}\, BYTE$
$permiso : PERMISO$
$longitud? : \mathbb{N}$

$\#datos > 0$
$longitud? \leq \#datos$
$(longitud \neq 0 \Rightarrow permiso \neq prohibido)$

Volviendo a la operación *Leer*, y habiendo obtenido su *VIS*, ya puede comenzarse con la aplicación de tácticas de testing. Las mismas serán explicadas, una por una, a continuación.

3.5.1.2. Forma Normal Disyuntiva (DNF)

La táctica de Forma Normal Disyuntiva (cuya abreviatura será DNF, por Disjunctive Normal Form), es la más conocida de las tácticas de testing y por lo general es la primera que se aplica. La misma se basa en el concepto que proviene de la lógica, con el mismo nombre, y que asegura que un predicado está escrito en DNF si es una disyunción de una o más conjunciones de uno o más literales. Un *literal* es un predicado atómico (indivisible) o la negación de un predicado atómico. Para llevar cualquier predicado a DNF se pueden aplicar los siguientes cuatro pasos:

1. Se transforman todas las equivalencias $a \Leftrightarrow b$ contenidas en el predicado en $(a \Rightarrow b) \wedge (b \Rightarrow a)$.
2. Se transforman todas las implicaciones $a \Rightarrow b$ contenidas en el predicado en $\neg\, a \vee b$.
3. Se distribuyen todos los conectivos de negación ($\neg$) de forma tal que siempre queden junto a predicados atómicos. Para esto se deben utilizar las siguientes reglas:
   - $\neg\, (a \vee b) \Leftrightarrow \neg\, a \wedge \neg\, b$
   - $\neg\, (a \wedge b) \Leftrightarrow \neg\, a \vee \neg\, b$
   - $\neg \neg\, a \Leftrightarrow a$
4. Se distribuyen todas las conjunciones sobre las disyunciones.

Ahora bien, para poder aplicar la táctica DNF a una clase de prueba *ClaseDePrueba* del árbol de una operación *Op*, se deben realizar los siguientes dos pasos:

- Expresar el esquema *Op* como una disyunción de uno o más esquemas, Op_1, Op_2, ..., Op_n, donde cada Op_i verifique tener como predicado una conjunción de literales.
- Dividir *ClaseDePrueba* utilizando las precondiciones de los esquemas Op_i, para i de 1 a n. Esto significa, por un lado, que hay una nueva clase de prueba por cada uno de estos esquemas. Y por otro lado, que cada clase de prueba tiene como predicado la conjunción del predicado de la clase de prueba original, *ClaseDePrueba*, con la precondición del esquema que le corresponde entre Op_1, Op_2, ..., Op_n.

Para mostrar el procedimiento en la práctica, se lo aplicará al *VIS* de la operación *Leer* del ejemplo de lectura de archivos. El primer paso es entonces expresar a la operación como una disyunción de esquemas que cumplan con la condición de tener como predicado una conjunción de literales. Como *Leer* ya está definida como una disyunción de esquemas, habrá que analizar cuáles de estos esquemas tienen que subdividirse en más esquemas, a través de más disyunciones, de tal manera que cada uno cumpla con la recién mencionada condición.

Por empezar, observando que el esquema *LeerOk* no tiene esta estructura por contener un $\Rightarrow$ en su predicado, es evidente que habrá que transformarlo, y quizás subdividirlo. La manera de realizar esta subdivisión es reescribiendo su predicado en Forma Normal Disyuntiva y derivando un esquema por cada término de la disyunción, de tal forma que cada término sea el predicado del esquema correspondiente. En el caso de *LeerOk*, teniendo en cuenta que $a \Rightarrow b$ es equivalente a $\neg\ a \lor b$ y que la conjunción puede distribuirse respecto a la disyunción, es posible dividir al esquema *Leer* en los dos siguientes esquemas de operación:

*LeerOk*1

$\Xi Archivo$
$longitud? : \mathbb{N}$
$lectura! : \text{seq}\, BYTE$
$rep! : REPORTE$

$\#datos > 0$
$longitud? \leq \#datos$
$longitud = 0$
$lectura! = (1..longitud?) \lhd datos$
$rep! = ok$

*LeerOk*2

$\Xi Archivo$
$longitud? : \mathbb{N}$
$lectura! : \text{seq}\, BYTE$
$rep! : REPORTE$

$\#datos > 0$
$longitud? \leq \#datos$
$permiso \neq prohibido$
$lectura! = (1..longitud?) \lhd datos$
$rep! = ok$

los cuales verifican tener predicados que son conjunciones de literales, como se pretendía. En el caso de los esquemas de operación *ArchivoVacio* y *FinDeArchivoEncontrado*, puede verse que cumplen con esto desde un principio, por lo que no será necesario modificarlos ni subdividirlos.

Por último, habrá que transformar el predicado de *ArchivoProhibido*, que al igual que el de *LeerOk*, también contiene un $\Rightarrow$. Como en este caso la implicación está negada, el esquema se reescribe pero no es necesario subdividirlo.

ArchivoProhibido

$\Xi Archivo$
$longitud? : \mathbb{N}$
$lectura! : \text{seq}\, BYTE$
$rep! : REPORTE$

$longitud? \neq 0$
$permiso = prohibido$
$lectura! = \langle\rangle$
$rep! = error\_acceso$

Después de todo esto, la operación *Leer* queda expresada de la siguiente forma:

$$Leer == LeerOk1 \lor LeerOk2 \lor ArchivoVacio \lor FinDeArchivoEncontrado \lor ArchivoProhibido$$

Para finalizar la aplicación de la táctica DNF, se debe dividir VIS_{Leer} con la precondición de cada uno de los esquemas que conforman *Leer*, lo que resulta en el siguiente conjunto de esquemas:

$$Leer_1^{DNF} == [VIS_{Leer} \mid \#datos > 0 \land longitud \leq \#datos \land longitud = 0]$$

$$Leer_2^{DNF} == [VIS_{Leer} \mid \#datos > 0 \land longitud? \leq \#datos \land permiso \neq prohibido]$$

$$Leer_3^{DNF} == [VIS_{Leer} \mid \#datos = 0]$$

$$Leer_4^{DNF} == [VIS_{Leer} \mid longitud? > \#datos]$$

$$Leer_5^{DNF} == [VIS_{Leer} \mid longitud? \neq 0 \land permiso = prohibido]$$

En primer lugar, es importante notar que no se ha obtenido una partición de VIS_{Leer} pues no se cumple con que todas las particiones sean disjuntas entre sí. Por ejemplo, $Leer_1^{DNF}$ y $Leer_2^{DNF}$ no son disjuntas. Sin embargo, esta división en clases de prueba sí realiza un cubrimiento del *VIS* de *Leer*. Se puede notar que a partir de estas clases pueden elegirse casos de prueba que hagan posible testear al sistema en las situaciones más importantes, las que se listan a continuación:

1. Se quiere leer una cantidad nula de bytes (y que no sobrepasa el tamaño) del archivo, el cual no está vacío.
2. Se quiere leer una cantidad de bytes que no sobrepasa el tamaño del archivo, el cual no está vacío y no tiene permiso de acceso prohibido.
3. Se quiere leer un archivo vacío.
4. Se quiere leer del archivo una cantidad de bytes que sobrepasa el tamaño del mismo.
5. Se quiere leer una cantidad no nula de bytes del archivo, el cual tiene permiso de acceso prohibido.

Como aparte de éstas hay otras pruebas que también puede ser interesante realizar, habría que encontrar la manera de generarlas a partir de alguna o algunas otras tácticas de testing. Así, se podría combinar las pruebas nuevas con las anteriores a través del árbol de pruebas de la operación. La siguiente táctica muestra como agregar nuevas clases de prueba al árbol actual, el cual contiene al *VIS* de *Leer* como raíz y a las cinco clases recién listadas como nodos hijos de la raíz.

3.5.1.3. Particiones Estándar (SP)

La táctica de Particiones Estándar (SP, por *Standard Partitions*) fue desarrollada por Stocks y Carrington [6, 8] y consiste en utilizar los operadores matemáticos que aparecen en la operación para generar nuevas clases de prueba. Una metodología tradicional en el testing basado en especificaciones es reducir una especificación a Forma Normal Disyuntiva, como se vio en la sección anterior, y elegir entradas que satisfagan las precondiciones de cada término de la disyunción obtenida. Sin embargo, esto tiende a ser muy simplista porque, por lo general, los lenguajes de especificación tienen operadores matemáticos poderosos donde la complejidad de sus dominios de entrada no se hace evidente en una transformación a DNF. Es esta complejidad a nivel de especificación la que normalmente se ve reflejada en la implementación de los propios operadores, y por lo que omitir su análisis puede impedir que se revelen algunos errores existentes en el programa. Esta táctica, por lo tanto, divide los dominios de estos operadores de acuerdo a la *partición estándar* de cada uno ellos. Una partición estándar es una partición del dominio del operador en conjuntos llamados *sub-dominios*, los cuales pueden definirse como condiciones expresadas en función de los operandos del operador. Es la condición que define cada subdominio la que determinará cada nueva clase de prueba.

A modo de ilustración, se aplicará esta táctica a las clases de prueba obtenidas en la sección anterior. Para esto, en primer lugar, hay que seleccionar una ocurrencia de un operador en el esquema de operación *Leer*. Se eligirá el operador $\lhd$ de la expresión:

$$(1..longitud?) \lhd datos$$

y se tendrá en cuenta que para la restricción de dominio de un conjunto S y una relación R, $S \lhd R$, una partición estándar podría ser:

1. $R = \{\}$
2. $R \neq \{\} \land S = \{\}$
3. $R \neq \{\} \land S = \mathrm{dom}\, R$
4. $R \neq \{\} \land S \neq \{\} \land S \subset \mathrm{dom}\, R$
5. $R \neq \{\} \land S \neq \{\} \land S \cap \mathrm{dom}\, R = \{\}$
6. $R \neq \{\} \land S \cap \mathrm{dom}\, R \neq \{\} \land \neg\, (S \subseteq \mathrm{dom}\, R)$

Así, lo que sigue es reemplazar los parámetros formales que aparecen en el listado de sub-dominios de la partición (S y R) por los parámetros reales que aparecen en la expresión original ($1..longitud?$ y $datos$), para de esta forma obtener la partición en términos de éstos últimos. El último paso es considerar la clase de prueba sobre la que se quiere aplicar la táctica y generar otras nuevas a partir de los sub-dominios de la partición resultante. En este caso se aplicará sobre $Leer_2^{DNF}$ y $Leer_4^{DNF}$.

De esta forma, se obtienen nuevas clases de prueba, que están contenidas en las anteriores y que se representan en el árbol de pruebas como las primeras "colgando" de las últimas. En los esquemas

correspondientes, deben incluirse unas dentro de otras, como se verá en el siguiente listado, que presenta todas las clases de prueba obtenidas hasta el momento:

$$
\begin{aligned}
&Leer_1^{SP} == [Leer_2^{DNF} \mid datos = \{\}] \\
&Leer_2^{SP} == [Leer_2^{DNF} \mid datos \neq \{\} \land 1..longitud? = \{\}] \\
&Leer_3^{SP} == [Leer_2^{DNF} \mid datos \neq \{\} \land 1..longitud? = \mathrm{dom}\ datos] \\
&Leer_4^{SP} == [Leer_2^{DNF} \mid datos \neq \{\} \land 1..longitud? \neq \{\} \land 1..longitud? \subset \mathrm{dom}\ datos] \\
&Leer_5^{SP} == [Leer_2^{DNF} \mid datos \neq \{\} \land 1..longitud? \neq \{\} \land 1..longitud? \cap \mathrm{dom}\ datos = \{\}] \\
&Leer_6^{SP} == [Leer_2^{DNF} \mid datos \neq \{\} \land 1..longitud? \cap \mathrm{dom}\ datos \neq \{\} \\
&\qquad \land \neg\ (1..longitud? \subseteq \mathrm{dom}\ datos)]
\end{aligned}
$$

$$
\begin{aligned}
&Leer_7^{SP} == [Leer_4^{DNF} \mid datos = \{\}] \\
&Leer_8^{SP} == [Leer_4^{DNF} \mid datos \neq \{\} \land 1..longitud? = \{\}] \\
&Leer_9^{SP} == [Leer_4^{DNF} \mid datos \neq \{\} \land 1..longitud? = \mathrm{dom}\ datos] \\
&Leer_{10}^{SP} == [Leer_4^{DNF} \mid datos \neq \{\} \land 1..longitud? \neq \{\} \land 1..longitud? \subset \mathrm{dom}\ datos] \\
&Leer_{11}^{SP} == [Leer_4^{DNF} \mid datos \neq \{\} \land 1..longitud? \neq \{\} \land 1..longitud? \cap \mathrm{dom}\ datos = \{\}] \\
&Leer_{12}^{SP} == [Leer_4^{DNF} \mid datos \neq \{\} \land 1..longitud? \cap \mathrm{dom}\ datos \neq \{\} \\
&\qquad \land \neg\ (1..longitud? \subseteq \mathrm{dom}\ datos)]
\end{aligned}
$$

Análogamente, se considerará el operador $\leq$ de la expresión:

$$longitud? \leq \#datos$$

con la siguiente partición estándar para $a \leq b$ (que también vale para $a < b$, $a > b$, $a \geq b$ y $a = b$):

1. $a < 0 \land b < 0$
2. $a < 0 \land b = 0$
3. $a < 0 \land b > 0$
4. $a = 0 \land b < 0$
5. $a = 0 \land b = 0$
6. $a = 0 \land b > 0$
7. $a > 0 \land b < 0$
8. $a > 0 \land b = 0$
9. $a > 0 \land b > 0$

que se aplicará a $Leer_5^{DNF}$, obteniendo:

$$
\begin{aligned}
&Leer_{13}^{SP} == [Leer_5^{DNF} \mid longitud? < 0 \land \#datos < 0] \\
&Leer_{14}^{SP} == [Leer_5^{DNF} \mid longitud? < 0 \land \#datos = 0] \\
&Leer_{15}^{SP} == [Leer_5^{DNF} \mid longitud? < 0 \land \#datos > 0] \\
&Leer_{16}^{SP} == [Leer_5^{DNF} \mid longitud? = 0 \land \#datos < 0] \\
&Leer_{17}^{SP} == [Leer_5^{DNF} \mid longitud? = 0 \land \#datos = 0] \\
&Leer_{18}^{SP} == [Leer_5^{DNF} \mid longitud? = 0 \land \#datos > 0] \\
&Leer_{19}^{SP} == [Leer_5^{DNF} \mid longitud? > 0 \land \#datos < 0] \\
&Leer_{20}^{SP} == [Leer_5^{DNF} \mid longitud? > 0 \land \#datos = 0] \\
&Leer_{21}^{SP} == [Leer_5^{DNF} \mid longitud? > 0 \land \#datos > 0]
\end{aligned}
$$

Como para ninguna táctica de testing existe la necesidad de aplicarse a todas las clases de prueba, en este caso se eligirá no hacerlo con la clase $Leer_1^{DNF}$ y $Leer_3^{DNF}$.

Es importante remarcar que cada clase de prueba está restringida tanto por la condición que aparece explícitamente como por la dada con la inclusión de esquemas. Por ejemplo, el predicado de $Leer_7^{SP}$ es en realidad $longitud? > \#datos \land datos = \{\}$. Teniendo en cuenta esto, se puede notar que varias de las clases recién listadas tienen predicados contradictorios (son conjunciones equivalentes a *false*), como es el caso de $Leer_1^{SP}$, $Leer_5^{SP}$, etc. Dado que éstas clases de prueba no están más que definiendo conjuntos vacíos de casos de prueba, no tienen ninguna utilidad, y se las puede descartar por completo[5], quedando sólo las que se listan a continuación:

$$Leer_2^{SP} == [Leer_2^{DNF} \mid 1..longitud? = \{\}]$$
$$Leer_3^{SP} == [Leer_2^{DNF} \mid 1..longitud? = \text{dom } datos]$$
$$Leer_4^{SP} == [Leer_2^{DNF} \mid 1..longitud? \neq \{\} \land 1..longitud? \subset \text{dom } datos]$$

$$Leer_7^{SP} == [Leer_4^{DNF} \mid datos = \{\}]$$
$$Leer_{12}^{SP} == [Leer_4^{DNF} \mid datos \neq \{\}]$$

$$Leer_{20}^{SP} == [Leer_5^{DNF} \mid longitud? > 0 \land \#datos = 0]$$
$$Leer_{21}^{SP} == [Leer_5^{DNF} \mid longitud? > 0 \land \#datos > 0]$$

En esta nueva presentación de las clases de prueba se han simplificado los predicados, omitido aquellas condiciones, generadas por la partición estándar, que puedan deducirse de las que ya introducen los esquemas incluidos en las clases. Por ejemplo, no se ha hecho explícito que $datos \neq \{\}$ en $Leer_2^{SP}$, $Leer_3^{SP}$ y $Leer_4^{SP}$ porque esto se deduce de la condición $\#datos > 0$ proveniente del predicado de $Leer_2^{DNF}$. Se puede observar aquí que las clases de prueba que se obtuvieron con la aplicación de la táctica de Particiones Estándar especifican formas más particulares de ejecutar el programa. Como muestra de esto, vale considerar que la alternativa funcional que se especifica en $Leer_2^{DNF}$ se dividió en:

- Se quiere leer una cantidad de bytes que no sobrepasa el tamaño del archivo, el cual no está vacío y no tiene permiso de acceso prohibido... y esa cantidad de bytes a leer es nula.
- Se quiere leer una cantidad de bytes que no sobrepasa el tamaño del archivo, el cual no está vacío y no tiene permiso de acceso prohibido... y esa cantidad de bytes es igual al tamaño del archivo. Es decir, se quiere leer completamente un archivo no vacío y que no tiene permiso de acceso prohibido.
- Se quiere leer una cantidad de bytes que no sobrepasa el tamaño del archivo, el cual no está vacío y no tiene permiso de acceso prohibido... y esa cantidad de bytes es menor al tamaño del archivo. Es decir, se quiere leer una porción de un archivo no vacío y que no tiene permiso de acceso prohibido.

Es evidente que de esta manera se pondrá a prueba el programa en las tres situaciones, mientras que con el enfoque anterior algunas de ellas podrían no haberse tenido en cuenta.

[5] A esta acción de eliminar las clases de prueba vacías se la llama *poda del árbol de pruebas*, y no está soportada por el prototipo actual de Fastest.

Si bien esta táctica de testing podría utilizarse con cualquier operador matemático, incluso con los que pueden agregarse como extensión al lenguaje, es cierto que Particiones Estándar depende de que esté definida la partición estándar del operador analizado. Como una alternativa, los mismos autores de esta técnica proponen solamente definir las particiones estándar de los operadores más simples y, expresando los operadores más complejos en función de éstos, derivar también sus particiones estándar. A esta otra táctica, aún no implementada en Fastest, la denominaron *Propagación de Sub-dominios* y pueden encontrarse más detalles de ella en [6, 8].

3.5.1.4. Tipos Libres (FT)

La táctica de Tipos Libres (FT, por Free Types) busca capturar todas las formas posibles en que puede construirse un término cuyo tipo sea un tipo libre. De esta manera, si el *VIS* de una operación *Op* declara una variable de tipo libre T, y se quiere aplicar esta táctica sobre una clase de prueba *ClaseDePrueba* de *Op*, deberá dividirse *ClaseDePrueba* en las clases que correspondan a todas las formas en que pueden construirse elementos de T. Por ejemplo, si se considera el siguiente tipo libre[6], que define el tipo de los árboles binarios:

$$ArbolBin ::= hoja \mid nodoInterno\langle\langle ArbolBin \times ArbolBin\rangle\rangle$$

y si el *VIS* de una cierta operación *CalculaAltura*, declara una variable *arbol* de tipo *ArbolBin*, la táctica FT genera la siguiente partición de $VIS_{CalculaAltura}$:

$$CalculaAltura_1^{FT} == [VIS_{CalculaAltura} \mid arbol = hoja]$$
$$CalculaAltura_2^{FT} == [VIS_{CalculaAltura} \mid \exists\, arbol_1, arbol_2 : ArbolBin \bullet arbol = nodoInterno(arbol_1, arbol_2)]$$

Para mostrar como se podría aplicar esta táctica a las clases de prueba del ejemplo guía, hay que considerar la variable *permiso* de tipo *PERMISO*. En este caso, el tipo *PERMISO* no define un tipo inductivo, sino que sólo un simple tipo enumerado. Entonces, si se aplica FT a las clases de prueba $Leer_3^{SP}$ y $Leer_3^{DNF}$, se obtienen las clases de prueba:

$$Leer_1^{FT} == [Leer_3^{SP} \mid permiso = prohibido]$$
$$Leer_2^{FT} == [Leer_3^{SP} \mid permiso = solo\_lectura]$$
$$Leer_3^{FT} == [Leer_3^{SP} \mid permiso = lectura\_escritura]$$

$$Leer_4^{FT} == [Leer_3^{DNF} \mid permiso = prohibido]$$
$$Leer_5^{FT} == [Leer_3^{DNF} \mid permiso = solo\_lectura]$$
$$Leer_6^{FT} == [Leer_3^{DNF} \mid permiso = lectura\_escritura]$$

pero como $Leer_1^{FT}$ resulta ser contradictoria (pues $Leer_3^{SP}$ introduce la condición $permiso \neq prohibido$), se la puede descartar, y por lo tanto sobreviven sólo las clases:

$$Leer_2^{FT} == [Leer_3^{SP} \mid permiso = solo\_lectura]$$
$$Leer_3^{FT} == [Leer_3^{SP} \mid permiso = lectura\_escritura]$$

[6] Este ejemplo muestra un tipo libre inductivo; Fastest sólo soporta tipos libres que sean enumerados.

$$Leer_4^{FT} == [Leer_3^{DNF} \mid permiso = prohibido]$$
$$Leer_5^{FT} == [Leer_3^{DNF} \mid permiso = solo\_lectura]$$
$$Leer_6^{FT} == [Leer_3^{DNF} \mid permiso = lectura\_escritura]$$

las cuales representan, respectivamente, las siguientes alternativas funcionales:

- Se quiere leer completamente un archivo no vacío... que tiene acceso de sólo lectura.
- Se quiere leer completamente un archivo no vacío... que tiene acceso de lectura y escritura.
- Se quiere leer un archivo vacío... que tiene acceso prohibido.
- Se quiere leer un archivo vacío... que tiene acceso de sólo lectura.
- Se quiere leer un archivo vacío... que tiene acceso de lectura y escritura.

Como se vio en la Sección 3.5, las clases de prueba se relacionan unas con otras a través de un árbol de pruebas, donde se ve reflejado a partir de qué clases se fueron obteniendo las demás. La Figura 3.3 muestra entonces el árbol de pruebas de la operación *Leer* del ejemplo guía, que fue generado por la aplicación de las tácticas desarrolladas en este capítulo. Las clases de prueba que están recuadradas son las hojas del árbol, y sólo desde ellas se deberán extraer casos de prueba abstractos, lo cual se discutirá en la próxima sección.

3.6. Selección de casos de prueba abstractos

Una vez que se considera que las clases de prueba representan suficientes alternativas funcionales en el programa, es necesario seleccionar uno o más casos de prueba abstractos para cada una de ellas. Para expresar los casos de prueba se utilizarán esquemas Z, de manera similar a como se viene haciendo con las clases de prueba. En particular, un caso de prueba abstracto es una tupla de constantes, y se describe a través de un esquema, que declara las mismas variables que la clase de prueba que contiene al caso de prueba, y cuyo predicado contiene una conjunción de igualdades entre variables y constantes, con una igualdad por cada variable declarada. Si bien en algunos casos se utilizan constantes que ya están provistas por el lenguaje Z (como las pertenecientes a $\mathbb{Z}$, el conjunto vacío, etc), en general es necesario que el encargado de testing introduzca las propias para los tipos que se definan en la especificación correspondiente. Para llevar a cabo esta tarea, se deberá hacer uso de las definiciones axiomáticas, asumiendo que si allí se definen dos variables con distintos nombres, éstas corresponden a distintas constantes. Por ejemplo, si en el ejemplo guía se quisiera introducir tres constantes distintas para el tipo $BYTE$, se podría hacer a través de la siguiente definición axiomática:

$$byte_1, byte_2, byte_3 : BYTE$$

Teniendo constantes para todos los tipos que participan en las declaraciones de variables de las clases de prueba de interés, se está en condiciones de escoger casos de prueba abstractos. De la misma manera que para expresar una nueva clase de prueba se incluía en su esquema el de la clase de prueba que la contenía, en este punto se documentará cada caso de prueba como un esquema que incluya al de

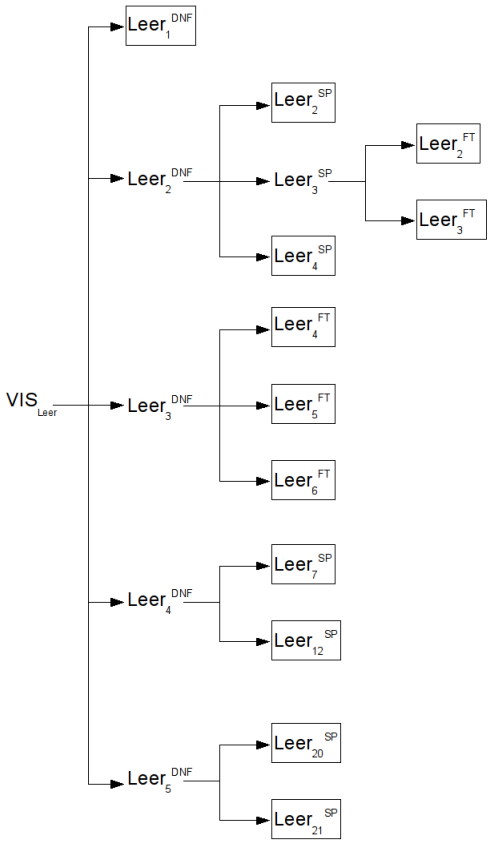


Figura 3.3: El árbol de pruebas generado para la operación *Leer* del ejemplo guía. Las clases de prueba recuadradas son aquellas a partir de las cuales se seleccionan casos de prueba abstractos.

la clase de prueba desde donde se seleccionó. Por ejemplo, considerando las clases de prueba $Leer_2^{FT}$, $Leer_6^{FT}$ y $Leer_{12}^{SP}$, se pueden elegir los siguientes casos de prueba abstractos:

$$Leer_2^{TC} == [Leer_2^{FT} \mid datos = \langle byte_1, byte_2, byte_3 \rangle \land longitud? = 3 \land permiso = solo\_lectura]$$
$$Leer_6^{TC} == [Leer_6^{FT} \mid datos = \langle\rangle \land longitud? = 2 \land permiso = lectura\_escritura]$$
$$Leer_{12}^{TC} == [Leer_{12}^{SP} \mid datos = \langle byte1 \rangle \land longitud? = 2 \land permiso = solo\_lectura]$$

donde *TC* hace referencia a Test Case (caso de prueba, en inglés). Es imperativo que las constantes seleccionadas para definir los casos verifiquen todas las condiciones impuestas por las clases de prueba que les corresponden. De otra forma, se estaría testeando el programa con entradas que no ponen a prueba las alternativas funcionales que se pretendían.

Aunque la teoría sugiere que la selección de casos de prueba abstractos a partir de clases de prueba es una tarea sencilla, su automatización es uno de los pasos más complicados del proceso de testing funcional. En la Sección 5.4.3 se mostrará cómo el prototipo que esta tesina describe aborda esta cuestión.

Capítulo 4

Estudio del estado del arte

En este capítulo se presentará un estudio del estado del arte de las propuestas similares a la aquí desarrollada, aquellas que involucran al software cuya función sea la de automatizar el proceso de testing funcional basado en especificaciones.

La herramienta *Conformiq Test Generator (CTQ)*, desarrollada por la empresa Conformiq en 2002, permite calcular, a partir de un modelo UML Statecharts (Unified Modelling Language, [15]) del sistema a testear, una serie de casos de prueba, y ejecutarlos sobre dicho sistema. Según la empresa, CTQ representa la segunda generación de su tecnología de derivación de casos de prueba, y fue precedida por Swifest, que es considerada (también por ellos) como la primera generación del rubro.

A principios de 2007, Conformiq lanzó al mercado la herramienta *Qtronic*, que representa la tercera generación de la compañía en lo que respecta a la automatización de testing basado en especificaciones. La diferencia fundamental entre Conformiq Test Generator y Qtronic radica en lo que el modelo o especificación usado para la generación de tests describe. Con CTQ es necesario definir un modelo que describa el entorno del sistema a testear. Por ejemplo, si se desea testear un servidor, los modelos deben definir la funcionalidad del cliente que interactúa con el servidor. Qtronic, por otra parte, funciona a partir de un modelo de comportamiento del propio sistema a testear, implementando una generación de tests guíadas por modelo real. Además, esta herramienta soporta modelos concurrentes, modelos multi-hilo, restricciones de tiempo y testing de sistemas no-determinísticos, y se integra con una cantidad considerable de herramientas CASE (Computer-aided Software Engineering) y editores UML. Para más detalles respecto a ambas herramientas, referirse a [16] y a [17].

UniTESK Lab., parte del Institute for System Programming (ISP) de la Russian Academy of Sciences (RAS) ha realizado investigación y desarrollo y ha provisto servicios de testing y verificación de sistemas de software desde hace aproximadamente 12 años. En particular, esta agrupación se dedica a aplicar una tecnología de testing basado en especificaciones formales, donde las especificaciones se escriben usando extensiones especializadas de los lenguajes de programación con que se codifican los sistemas a testear. UniTesk ha desarrollado extensiones para los lenguajes C y Java, soportadas por sus herramientas comerciales *CTESK* y *JavaTESK*, respectivamente. El grupo fue fundado en 1994 para llevar a cabo un proyecto para Nortel Networks que consistía en el testing de las API de un núcleo

de sistema operativo en tiempo real como paso previo a la integración del núcleo con una plataforma multiprocesador. Actualmente, las herramientas de UniTESK son usadas por el Linux Verification Project [18]. Se ofrece una descripción más minuciosa de la tecnología en [19].

Existe otra herramienta comercial, llamada *Test Designer*, que fue lanzada al mercado muy recientemente por la empresa LEIRIOS (renombrada a Smartesting en mayo de 2008) y que permite generar automáticamente casos de prueba y scripts ejecutables de testing a partir de un modelo funcional del sistema a testear. Utilizando éste modelo, que debe estar dado en el lenguaje UML, Test Designer también provee facilidades para la simulación del comportamiento esperado del sistema especificado. La herramienta es una versión renovada de su predecesora, *LEIRIOS Test Generator / UML*. La empresa también distribuía una versión de la misma para el lenguaje de especificaciones B, llamada *LEIRIOS Test Generator / B*. Puede buscarse más información en [20].

Rave (T-VEC's Requirements-based Automated Verification) es una herramienta comercial desarrollada por T-VEC que brinda la posibilidad de generar casos de prueba a partir de un modelo tabular del sistema a testear. La propia herramienta permite construir dichos modelos tabulares, utilizando un entorno gráfico que corre bajo Windows. El enfoque de modelado tabular fue derivado del método *Software Cost Reduction (SCR)* desarrollado por el U.S Naval Research Laboratory (NRL) y por David Parnas a principios de los 70. Rave también hace posible la transformación de casos de prueba en *scripts* de testing y en *test drivers* codificados en cualquier lenguaje de programación, para poder así llevar a cabo la ejecución de los tests. Pueden verse más detalles de la herramienta en [21].

Como una alternativa a Rave, la misma empresa ha desarrollado otro producto comercial llamado *T-VEC Tester for Simulink and Stateflow*. Éste automatiza gran parte del proceso de testing, analizando un modelo Simulink del sistema a testear con el objeto de obtener casos de prueba y así poder validar el modelo y verificar implementaciones del mismo. Al igual que Rave, esta herramienta ofrece la posibilidad de producir *tests drivers* para ejecutar vectores de testing contra el código fuente, que puede ser generado automáticamente a partir del modelo. Se pueden ver todas las características de T-VEC Tester for Simulink and Stateflow en [22].

Reactis, de Reactive Systems, es un paquete de software comercial ideado para la automatización del diseño de software y que consiste en tres componentes: Reactis Tester, Reactis Simulator y Reactis Validator. La parte que más nos interesa, Reactis Tester, genera tests a partir de modelos Simulink y Stateflow. Por su parte, Reactis Simulator y Reactis Validator, son utilidades pensadas para la simulación de modelos y la validación de requerimientos especificados por el usuario, respectivamente.
Reactis se ejecuta sólo sobre la plataforma Windows y requiere las aplicaciones MATLAB, Simulink, and Stateflow instaladas. Para obtener más detalles acerca del producto, dirigirse a [23].

Otra utilidad para la generación de casos de prueba es el módulo *Automatic Test Generator*, que es un componente de los productos comerciales *Telelogic Statemate* y *Telelogic Rhapsody*, desarrollados por Telelogic, de IBM. Ambas herramientas facilitan el prototipado y la simulación y son útiles en el desarrollo de sistemas embebidos complejos. Las mismas permiten capturar gráficamente requerimientos de sistemas, especificaciones y diseños utilizando UML como lenguaje base. Para más información sobre Telelogic Statemate y sobre Telelogic Rhapsody, consultar [24].

Si bien en el ambiente GNU no se ha encontrado gran cantidad de software relacionado al que éste trabajo describe, se pueden mencionar algunos trabajos existentes. ModelJUnit, por ejemplo, es una librería de Java que extiende JUnit para soportar testing basado en modelos. Tales modelos son máquinas de estados finitas que pueden escribirse en Java y a partir de ellos pueden obtenerse casos de prueba y medir una serie de métricas de cubrimiento. ModelJUnit es una herramienta de código abierto, distribuida bajo la licencia GNU GPL y fue desarrollada en 2007 por Mark Utting, profesor del Departamento de Ciencias de la Computación de la Universidad de Waikato, Nueva Zelanda. Pueden encontrarse más detalles sobre ModelJUnit, junto con binarios descargables de la librería, código fuente y documentación en [25].

Por otro lado, en la comunidad del software libre se cuenta con org.tigris.mbt, una implementación en Java del testing basado en especificaciones formales. Ésta ofrece la posibilidad de generar secuencias de tests a partir de máquinas de estado finitas dadas en forma de grafos. org.tigris.mbt toma un archivo graphml como entrada, por lo que el modelado del sistema a testear debe ser realizado en una herramienta aparte, como por ejemplo yED de yWorks[1] . La utilidad no soporta la ejecución de las pruebas generadas, que de necesitarse, podría hacerse utilizando alguna aplicación externa como Quick Test Professional, Functional Tester o Win32::GuiTest.

Como conclusión de este resumen, puede apreciarse que la mayoría del software existente para automatizar el testing funcional basado en especificaciones es comercial, de código cerrado y que ninguno de éstos productos trabaja con el lenguaje de interés en éste trabajo, el lenguaje de especificación Z.

[1]Disponible gratuitamente, sin ningún tipo de restricción de funcionalidad, en `http://www.yworks.com/en/products_yed_about.htm`.

Capítulo 5

Descripción de Fastest

El objetivo de este capítulo es presentar la versión actual de Fastest, una herramienta cuyo desarrollo fue impulsado para intentar automatizar, lo más posible, el proceso de testing funcional basado en especificaciones Z, el cual fue explicado en el capítulo 3. En esta parte de la tesina se irán describiendo las características funcionales de Fastest y explicando detalladamente su arquitectura, diseño e implementación. Para realizar este trabajo fueron aplicadas técnicas de Ingeniería de Software, se utilizó Java como lenguaje de programación y se recurrió al framework Community Z Tools (CZT) para contar con utilidades relacionadas al lenguaje de especificación Z. En lo que sigue se explicará con precisión cada uno de estos aspectos.

5.1. Visión general

En la Figura 5.1, se muestra el proceso de testing funcional desde la perspectiva de la herramienta aquí presentada, la cual interactúa con el usuario a través de un intérprete de comandos. Inicialmente, se toma como entrada una especificación Z[1] en formato LaTeX (que comúnmente viene en archivos .tex o .zed), la cual en una primera etapa se traduce a una representación interna en la herramienta, más adecuada para su manipulación (mayores detalles luego). A continuación, el usuario debe ingresar una lista con los nombres de operaciones a testear, y las tácticas de testing que se aplicarán a cada una de ellas. A partir de estos elementos, en una segunda etapa, es posible generar un árbol de pruebas para cada una de las operaciones. El paso que sigue es el de generación de casos de prueba abstractos, el cual aunque lo intenta, no garantiza que efectivamente se obtengan casos de prueba, debido a algunos factores que también serán descriptos más adelante.

Como puede verse, el diagrama está dividido en dos partes por medio de una línea de puntos horizontal. Esta línea separa las etapas que recién fueron mencionadas, y que están implementadas en la versión actual de la herramienta, de aquellas que si bien fueron en parte investigadas, aún no resultaron consideradas en el diseño ni volcadas a la implementación. De todas maneras, vale la pena hacer una breve presentación de ellas como parte de esta introducción, ya que no dejan de formar parte de las ideas que motivaron el desarrollo de Fastest.

[1] Más precisamente, la especificación Z debe respetar el estándar ISO [35].

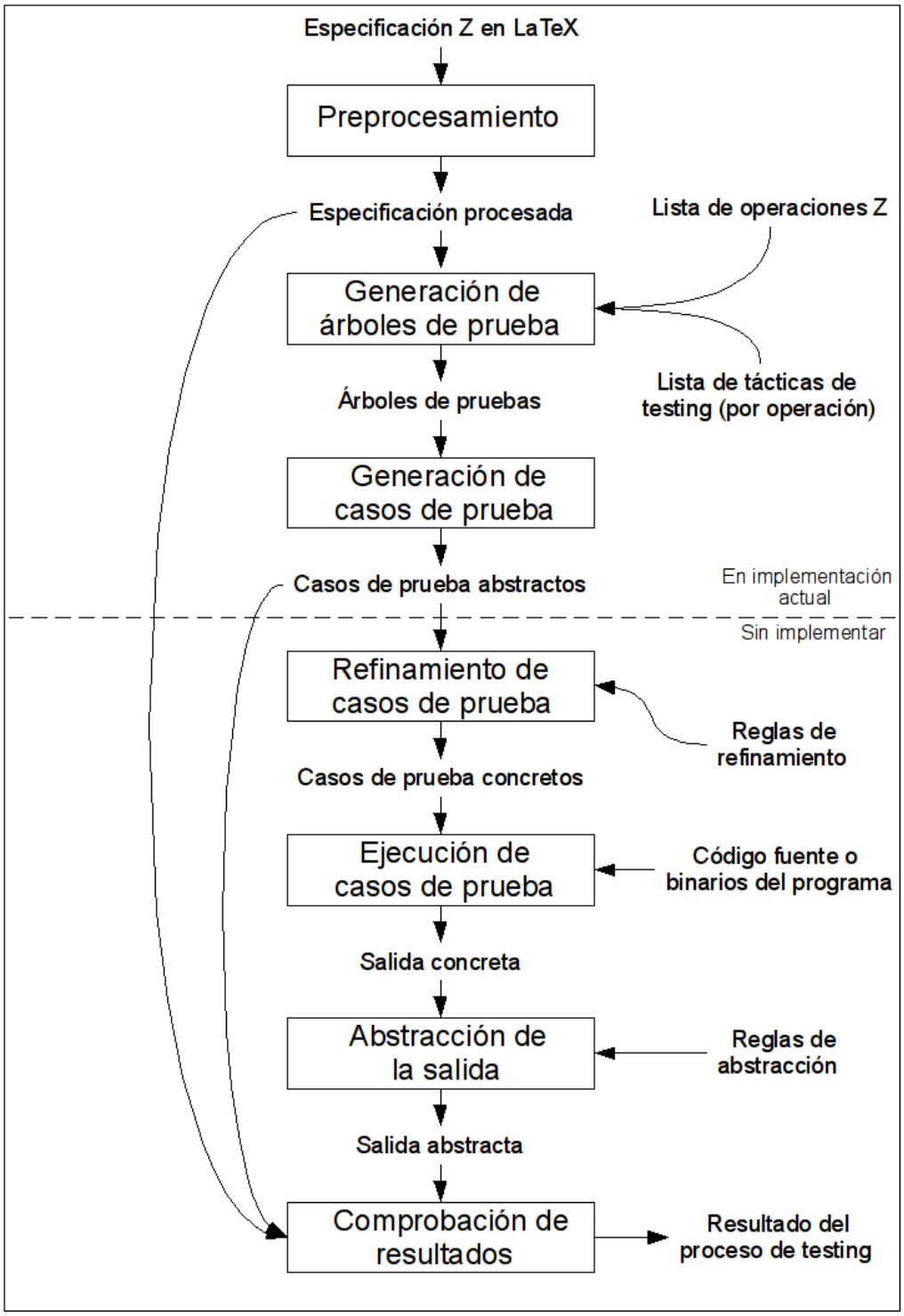


Figura 5.1: El proceso de testing en Fastest. La línea de puntos separa las etapas del proceso implementadas en la versión actual, de aquellas aun pendientes de desarrollo.

Entonces, luego de obtener casos de prueba abstractos para las operaciones Z seleccionadas, es necesario escribir estos casos en términos del lenguaje de programación (casos de prueba concretos), con el objeto de que puedan ser ejecutados. En este paso, sería necesario que el usuario provea a Fastest con una serie de reglas o funciones de refinamiento, las cuales deberían establecer una relación entre elementos sintácticos abstractos y elementos concretos. Llegado a este punto, y teniendo en cuenta también las etapas que siguen, sería conveniente diseñar este paso de forma independiente del lenguaje de programación utilizado, de tal forma de que la herramienta pueda ir soportando nuevos lenguajes a medida que vaya evolucionando.

A continuación, considerando los casos de prueba obtenidos, expresados en términos de implementación, y el código fuente o los ejecutables del programa a testear (elección que dará lugar a trabajos de investigación), será necesario ejecutar los casos de prueba en éste último. La salida obtenida deberá ser capturada por la herramienta y a través de una nueva etapa, transformarse a términos abstractos, es decir, otra vez expresada en función de construcciones del lenguaje Z. Para esto, al igual que para el refinamiento, será preciso contar con la intervención del usuario para proveer a Fastest de un conjunto de funciones de abstracción.

Por último, la etapa final consiste en utilizar la especificación original, los casos de prueba abstractos, y las salidas abstractas de la etapa anterior, para evaluar si la salida de cada caso de prueba corresponde con lo que indica la especificación. El resultado obtenido permitirá determinar si el programa testeado contiene errores, y en caso afirmativo, en qué partes de la implementación del programa éstos se encuentran.

5.2. Arquitectura

En esta sección se mostrarán las decisiones arquitectónicas que se tuvieron en cuenta a la hora de concebir a la herramienta desde un enfoque de alto nivel de abstracción. Se hará una introducción a la arquitectura de software y a los estilos arquitectónicos, para después pasar a discutir estas cuestiones en el caso particular de la herramienta presentada en esta tesina.

5.2.1. Introducción

La *arquitectura* de un sistema de software define al sistema en términos de componentes y de conectores entre tales componentes. Los componentes son elementos tales como clientes y servidores, bases de datos, filtros, y capas en un sistema estratificado. Las interacciones entre componentes a este nivel de diseño pueden ser simples y familiares, tales como llamadas a procedimiento o acceso a variables compartidas. Pero pueden también ser complejas y semánticamente ricas, como lo son los protocolos cliente-servidor, protocolos de acceso a bases de datos, difusión de eventos y flujos de datos por tubos [26].

Un concepto importante en relación a las arquitecturas de software es el de *estilo arquitectónico*. Un estilo arquitectónico define una familia de sistemas en términos de un patrón de organización estructural. Más precisamente, un estilo arquitectónico define un vocabulario de tipos de componentes y conectores, y un conjunto de restricciones de cómo pueden éstos combinarse entre sí. Para muchos

estilos también puede existir uno o más modelos semánticos que especifiquen cómo determinar las propiedades generales del sistema a partir de las propiedades de sus partes [26].

Algunos estilos arquitectónicos popularmente utilizados son:

- Tubos y filtros
- Cliente/Servidor
- Sistemas estratificados
- Invocación implícita basada en eventos
- Blackboard Systems
- Control de Procesos

5.2.2. Arquitectura de Fastest

A continuación se hará un descripción de las decisiones arquitectónicas que se consideraron para el caso particular de Fastest.

5.2.2.1. Estilos arquitectónicos empleados

Fastest fue desarrollado siguiendo una arquitectura Cliente/Servidor [27]. En este tipo de sistemas, un *servidor* representa un proceso que provee servicios a otros procesos llamados *clientes*. Es usual que los servidores no conozcan, en principio, la identidad de los clientes que tendrán acceso a ellos en tiempo de ejecución. Por otro lado, son los clientes quienes conocen la identidad de un servidor. En el caso de Fastest, habrá un servidor particular, el *servidor de datos*, donde entre otra información se encontrará el registro de todos los demás servidores, los *servidores de cómputo*.

La razón principal por la cual se eligió el mencionado estilo arquitectónico fue la de aprovechar eficientemente los recursos que provee una red de computadoras. Principalmente, se ha buscado obtener mayor desempeño ejecutando varias tareas en paralelo, pero además, tener la posibilidad de correr más de un cliente de Fastest en distintas terminales. Todos los clientes deben consultar a un único repositorio de datos, ubicado en el llamado servidor de datos. Por otra parte, estos solicitan a los servidores de cálculo la generación de casos de prueba abstractos y la comprobación de correspondencia entre casos de prueba abstractos, especificación y salida abstracta resultante de la ejecución de tales casos.

Para que todos los componentes de una arquitectura Cliente/Servidor cooperen entre sí, se deben emplear ciertas técnicas de comunicación intercomponentes. En teoría, hay tres tipos básicos de técnicas de comunicación de procesamiento cooperativo que una arquitectura Cliente/Servidor puede usar:

- Tubos
- Llamadas a procedimiento remotos
- Interacciones Cliente/Servidor SQL

Los *tubos* son mecanismos orientados a la conexión que pasan datos de un proceso a otro. En principio, los procesos pueden estar en diferentes máquinas, incluso corriendo sobre distintos sistemas operativos. Los detalles de los mecanismos de transporte soportados están ocultos a los usuarios de un tubo y los tubos imponen restricciones mínimas de protocolo y formato a los usuarios.

Una *llamada a procedimiento remoto* es un mecanismo por el cual un proceso puede ejecutar otro proceso (subrutina) que reside en un sistema diferente, y usualmente remoto, posiblemente corriendo en un sistema operativo diferente. Cualquier parámetro necesario para la subrutina es pasado entre el proceso original y el de la subrutina. Al igual que con los tubos, los detalles del mecanismo de transporte se ocultan al usuario de la llamada a procedimiento.

Una *interacción Cliente/Servidor SQL* es un mecanismo que permite pasar consultas Structured Query Language (SQL) y los datos asociados a ellas de un proceso (usualmente un cliente) a otro proceso (servidor). Esta es un caso especial de interacción Cliente/Servidor, que se utiliza en el contexto de aplicaciones de base de datos relacionales distribuidas. En este caso, el servidor es el servidor de la base de datos relacional. Este puede residir en un sistema diferente y posiblemente corriendo en un sistema operativo distinto. Mientras los detalles de los mecanismos de transporte están ocultos por los desarrolladores de aplicaciones, las interacciones Cliente/Servidor de SQL imponen restricciones severas de protocolos y formato a los usuarios.

En el caso de Fastest, para realizar las interacciones entre clientes y servidores de cómputo se utilizaron unos tubos particulares llamados *sockets*. Los sockets establecen una abstracción a través de la cual dos programas (posiblemente situados en computadoras distintas) pueden intercambiar flujos de datos, generalmente de manera fiable y ordenada. Un socket queda definido por una dirección IP, un protocolo y un número de puerto.

Por otro lado, cada cliente de la arquitectura Cliente/Servidor es organizado de acuerdo a una arquitectura de Invocación Implícita. La idea detrás de la Invocación Implícita, la cual se ilustra en la Figura 5.2, es que los componentes invoquen los servicios provistos por otros componentes a través de un mecanismo indirecto. En efecto, un componente puede anunciar uno o más eventos. Los componentes del sistema pueden registrar interés en un evento, asociando a él un procedimiento. Cuando el evento es anunciado por algún componente, el propio sistema (a través de un administrador de eventos) invoca a todos los procedimientos que anteriormente registraron interés en tal evento. Entonces, el anuncio de un evento "implícitamente" causa la invocación de procedimientos en otros módulos. En términos arquitectónicos, los componentes en un estilo de Invocación Implícita son módulos cuyas interfaces pueden proveer tanto una colección de procedimientos como un conjunto de eventos. Los procedimientos pueden ser invocados de la forma usual, pero un componente puede también registrar algunos de sus procedimientos con eventos del sistema. Esto causará que tales procedimientos sean invocados cuando los eventos de interés sean anunciados en tiempo de ejecución [26].

El principal invariante del estilo es que los anunciantes de eventos no conocen qué componentes serán afectados por los eventos que pueden lanzar. Por esta razón, los componentes no tienen la posibilidad de hacer suposiciones acerca del orden de procesamiento, ni sobre si el procesamiento ocurrirá o no como resultado del anuncio de determinado evento [28]. El beneficio más importante de este as-

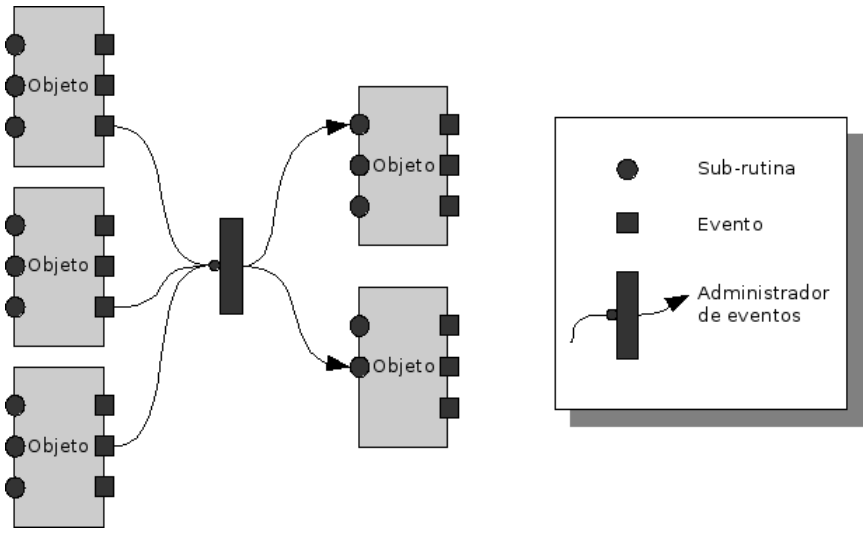


Figura 5.2: El estilo arquitectónico de Invocación Implícita.

pecto, y el principal motivo que llevó a elegir este estilo para organizar los clientes de Fastest, es que facilita enormemente la evolución del sistema: los componentes pueden ser reemplazados por otros componentes o pueden agregarse otros nuevos sin afectar las interfaces de los ya existentes.

En este sistema, la capacidad dinámica de poder agregar y quitar componentes del sistema se controla a través de un archivo de configuración donde se indica qué procedimientos de qué componentes están interesados en qué eventos. Simplemente lo que se hace es leer este archivo de configuración al momento de iniciar el cliente del sistema, almacenando en una tabla dentro del administrador de eventos las relaciones entre eventos, componentes y procedimientos de componentes. Por esto, aunque la declaración de eventos es estática (dado que el conjunto posible de eventos queda fijado en tiempo de compilación), el binding entre eventos y procedimientos es dinámico: los componentes registran su interés en eventos en tiempo de ejecución. La declaración de eventos es centralizada: nos pareció suficiente contar con un único administrador de eventos. En cuanto a la estructura de los eventos, se decidió que los mismos tengan una lista de parámetros por tipo de evento, es decir, cada evento tiene una lista fija de parámetros pero la cantidad y tipo puede ser diferente para cada evento.

Si bien en términos de arquitectura se habla de Invocación Implícita, la implementación de este mecanismo se realizó a través de llamadas a procedimientos tradicionales. Esto es, el anuncio de un evento se lleva a cabo realizando la invocación explícita de una subrutina particular del administrador de eventos (módulo llamado EventAdmin), pasando como parámetro la instancia del módulo que representa al evento. El administrador simplemente consulta su tabla de eventos en busca de el o los procedimiento/s interesado/s en el evento, para luego realizar cada una de las invocaciones en un hilo de proceso diferente. Luego, es responsabilidad del componente "invocado" (esto es, del cual se invocó un procedimiento) obtener los parámetros relevantes del evento para llevar a cabo la tarea que le corresponde.

5.2.2.2. Diagrama canónico

Los componentes del estilo Cliente/Servidor se organizan en capas lógicas. Una de las topologías más comunes consta de las cuatro capas lógicas que se describen a continuación:

Lógica de presentación (LP) Esta es la capa que se encarga de interactuar en forma directa con el usuario del sistema. Contiene el código responsable de dar disposición a los elementos gráficos de la pantalla así como de escribir información en la misma. En el caso particular de Fastest, esta porción del sistema se ubica exclusivamente en los clientes, ya que son los únicos que brindan una interfaz al usuario.

Lógica de Negocio (LN) Es la parte del sistema que usa los datos de entrada y realiza tareas de negocio. En el caso de Fastest, la misma está dividida entre los clientes y los servidores de cómputo. En los primeros se encontrarán aquellas tareas relativas a:

- la validación de los datos de entrada, esto es: especificación y código fuente (o binarios) del programa a testear, listado de operaciones a testear, listado de tácticas a aplicar y, eventualmente, nuevas tácticas, funciones de refinamiento y abstracción y reglas de simplificación de predicados.
- la generación del árbol de pruebas partiendo de la especificación y de las tácticas indicadas.
- la poda de los árboles de pruebas generados, eliminando clases de prueba cuyos predicados equivalgan a *false*
- el refinamiento de casos de prueba abstractos en casos de prueba concretos.
- la abstracción de casos de prueba concretos en casos de prueba abstractos.
- la ejecución del programa usando los casos de prueba concretos.
- el almacenamiento de casos de prueba abstractos y concretos, así como registro de los tests fallidos y exitosos.

Por el lado de los servidores de cómputo, se encontrará la parte de la lógica de negocio vinculada a:

- la generación de casos de prueba abstractos a partir de casos de prueba concretos.
- la comprobación de la correspondencia entre los casos de prueba abstractos, la especificación y la salida abstracta que se obtenga de ejecutar los casos de prueba.

Esta división de funcionalidad se justifica en que las dos tareas asignadas a los servidores de cómputo son bastante costosas, y por lo tanto, se asume que se puede necesitar dividir su procesamiento en varios equipos.

Lógica del Procesamiento de los Datos (LPD) En esta capa se oculta la forma en que se consultan o almacenan los datos persistentes. Dado que tanto clientes como servidores de cómputo hacen uso de tales datos persistentes, hay partes de la LPD en ambos tipos de aplicaciones.

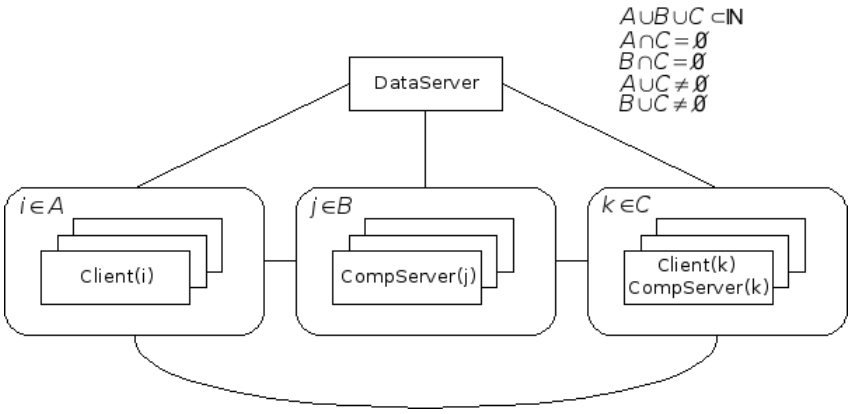


Figura 5.3: Distribución de procesos en hardware, donde el proceso correspondiente al servidor de datos (DataServer) corre de forma solitaria en una computadora.

DBMS Aquí es donde se almacenan los datos persistentes. En particular, estos corresponden a:

- las definiciones de las tácticas de testing
- las definiciones de las funciones de refinamiento y abstracción
- las reglas de simplificación de predicados
- información sobre los servidores de cómputo del sistema
- historial de pruebas realizadas previamente, para evitar calcular casos de prueba nuevamente si se modifica la implementación del programa a verificar

5.2.2.3. Estructura de Hardware y Estructura Física

Los clientes y servidores de Fastest pueden dividirse en capas físicas de acuerdo a la topología física de la red de la corporación donde el sistema funcione. Habrá que tener en cuenta que debe haber un único servidor de datos, uno o más servidores de cómputo o procesamiento y uno o más clientes. Es importante destacar que aunque el sistema fue pensado para ser distribuido, tanto servidores como clientes pueden, eventualmente, correr en una misma computadora.

Dado que clientes y servidores del sistema requieren hardware con prestaciones similares, la red estará compuesta solo por PC's de aproximadamente el mismo potencial. Por lo tanto, se considera que la Estructura de Hardware consta de una única capa.

La Estructura Física, también llamada Estructura de Despliegue, permite relacionar elementos de software con plataformas de ejecución. La relación que se usará para documentar esta vista es $ALOJADO\_EN$, que en este caso vinculará procesos con procesadores o computadoras. En las Figuras 5.3, 5.4, 5.5 y 5.6 se ven representadas estas relaciones, teniendo en cuenta las referencias de la Figura

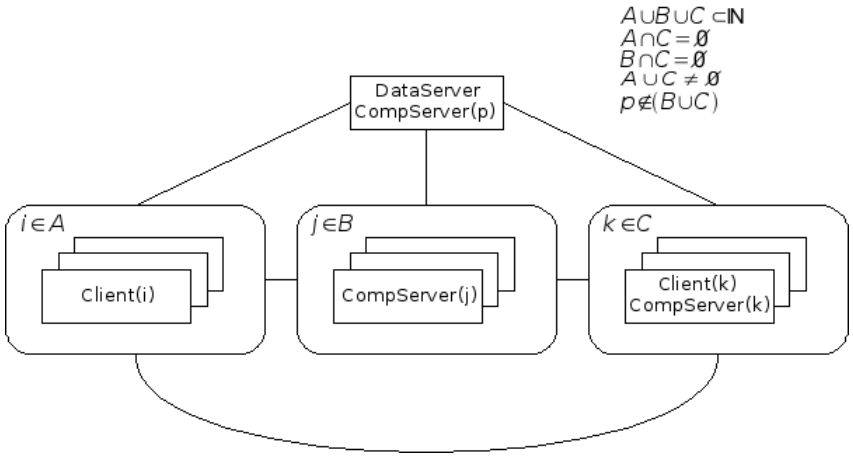


Figura 5.4: Distribución de procesos en hardware, donde el proceso correspondiente al servidor de datos (DataServer) corre junto a un servidor de cómputo (CompServer) en una computadora.

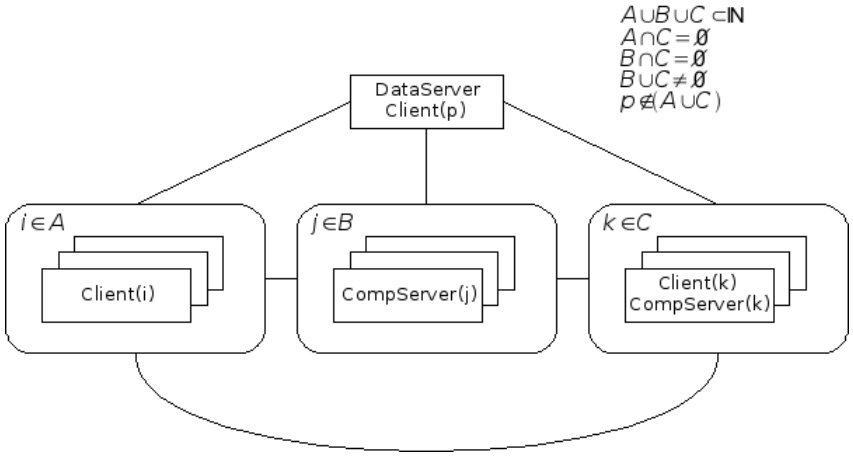


Figura 5.5: Distribución de procesos en hardware, donde el proceso correspondiente al servidor de datos (DataServer) corre junto a un cliente (Client) en una computadora.

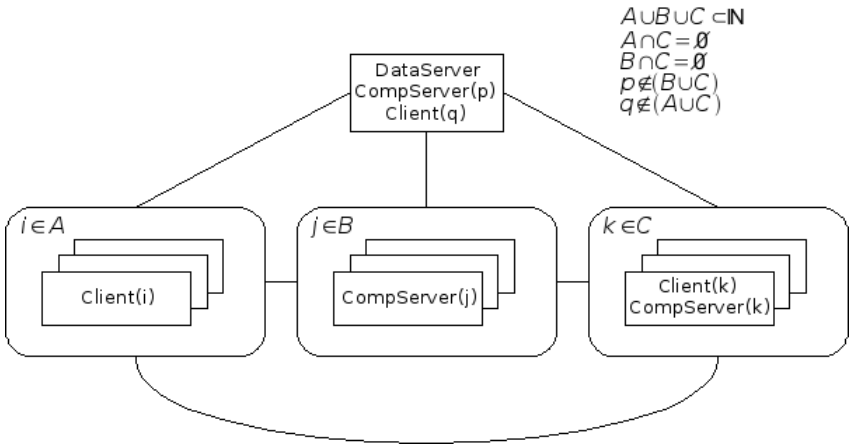


Figura 5.6: Distribución de procesos en hardware, donde el proceso correspondiente al servidor de datos (DataServer) corre junto a un servidor de cómputo (CompServer) y a un cliente (Client) en una computadora.

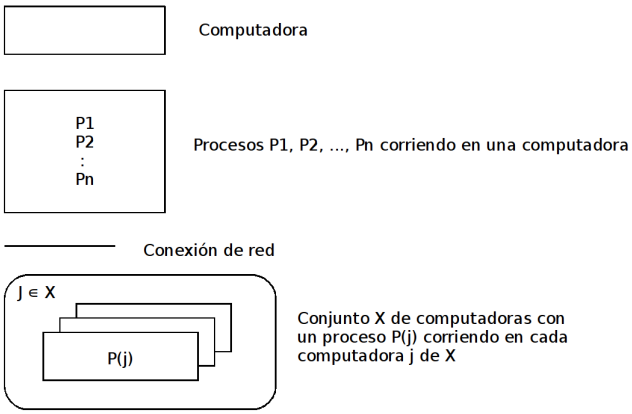


Figura 5.7: Referencias.

5.7, la cual indica que los nodos (denotados con rectángulos) son computadoras, los arcos conexiones de red y en cada nodo se dibujan los procesos alojados en esa computadora. Además, una agrupación de computadoras se grafica a través de rectángulos apilados dentro de un rectángulo redondeado. Las restricciones matemáticas en cada caso indican la necesidad de tener al menos un servidor de cómputo y al menos un cliente corriendo en alguna computadora, así como la imposibilidad de tener un mismo cliente o un mismo servidor de cómputo corriendo en más de una computadora. Además, las conexiones de red entre los equipos (líneas continuas en los esquemas) deben asegurar que el servidor de datos pueda comunicarse con al menos un servidor de cómputo y un cliente, y que a su vez éstos estén conectados entre sí.

5.2.3. Implementación actual de la arquitectura de Fastest

Como se adelantó en la Sección 5.1, la versión de Fastest que este trabajo presenta no implementa completamente el proceso de testing allí descripto. Solamente está soportada la etapa de generación de casos de prueba, dejando pendientes las que corresponden al refinamiento de casos de prueba, la ejecución de casos de prueba, la abstracción de la salida de las ejecuciones y la comprobación entre casos de prueba, especificación y salidas obtenidas. En consecuencia, debe quedar claro en el resto del trabajo que cualquier mención de éstas últimas etapas del proceso de testing solamente está relacionada a cuestiones de arquitectura y de diseño pero no de implementación.

Por otra parte, el prototipo de Fastest que existe actualmente no cuenta con el servidor de datos mencionado en las últimas secciones, aunque está previsto agregarlo en un futuro cercano. La mayoría de la información que anteriormente se indicó como parte de este repositorio de datos se encuentra en los clientes de la herramienta. Por lo tanto, ésta no se encuentra centralizada, sino que la misma es replicada en cada uno de los clientes. Este no es el caso de los historiales y resúmenes de tests, ya que los mismos directamente no están implementados en esta versión de Fastesta.

5.3. Tecnología utilizada

5.3.1. Java

Java [30] es un lenguaje de programación, de propósito general y orientado a objetos, que fue introducido por Sun Microsystem en 1995. Se distingue por proveer un entorno para la ejecución de programas, llamado máquina virtual de Java. Este entorno permite que las aplicaciones desarrolladas en Java puedan ejecutarse en cualquier máquina, independientemente del sistema operativo y la configuración de hardware utilizados. Sun describe al lenguaje Java como “simple, orientado a objetos, distribuido, interpretado, robusto, seguro, de arquitectura neutra, portable, de alta prestaciones, multitarea y dinámico”. La plataforma Java proporciona:

- *Java Virtual Machine* (JVM)
- *Java Application Programming Interface* (API)

donde JVM es la máquina virtual de Java antes mencionada y donde API es un conjunto de componentes de software que provee diferentes utilidades. Estos últimos están agrupados en librerías de clases

e interfaces relacionadas, a las cuales se las denomina *paquetes*, y son practicamente imprescindibles para el desarrollo de nuevas aplicaciones Java.

Se eligió Java como lenguaje de desarrollo por su expresividad, la posibilidad de generar documentación directa a través de Javadoc, y su integración con utilidades ya existentes vinculadas al lenguaje Z, las que serán explicadas en la próxima sección.

5.3.2. CZT

5.3.2.1. Introducción a CZT

El proyecto *Community Z Tools* (CZT)[2] [31] es un framework de código abierto creado con el objetivo de construir un conjunto de herramientas, basadas en métodos formales, para la notación Z y otros dialectos de Z. Estas herramientas incluyen soporte para editar, parsear, hacer chequeo de tipos (*typechecking*) e imprimir especificaciones Z en LaTeX, Unicode o formatos XML. Estas utilidades son desarrolladas bajo el lenguaje de programación Java.

CZT fue propuesto por Andrew Martin en 2001, con el objetivo de "formar una comunidad en Internet cuyo objetivo sea el construir un framework para integrar herramientas Z y agregados (plug-ins) para estas herramientas". En 2003, Petra Malik y Mark Utting crearon el proyecto CZT en *Sourceforge*, uno de los más grandes sitios web del mundo relacionados al desarrollo de software de código abierto.

El lenguaje de especificación Z fue adoptado como un estándar ISO en 2002 [35]. Una de las principales barreras para la difusión de este estándar, la falta de herramientas de soporte, fue uno de los impulsores más importantes del proyecto CZT. Si bien existían herramientas Z disponibles para realizar parseo y chequeo de tipos y algunas que ofrecían verificación formal de programas, la mayoría de ellas no soportaba el Estándar ISO para Z. Más aún, estas herramientas usaban diferentes versiones de Z, requerían diferentes macros de LaTeX, y había poca integración entre ellas.

A continuación se describirán las facilidades provistas por CZT que han sido usadas en este trabajo.

5.3.2.2. CZT Parser y Árboles de Sintaxis Abstracta (AST)

El CZT Parser puede transformar especificaciones Z en varios formatos (LaTeX, Unicode, XML, etc.) en una representación en forma de *árbol de sintaxis abstracta* (AST, por Annotated Syntax Tree). AST permite representar en forma de árbol, una especificación Z que haya sido parseada, usando una serie de clases e interfaces Java[3]. Esto facilita el acceso desde aplicaciones Java a elementos sintácticos del lenguaje Z, como esquemas, predicados y expresiones. Un usuario del AST debe siempre hacer uso de referencias a interfaces en lugar de referencias a clases concretas. De esta manera, se protege al usuario de cambios en la implementación subyacente y permite la utilización de diferentes implementaciones de las interfaces AST.

[2] Ver `http://czt.sourceforge.net`.

[3] Esta colección de clases e interfaces que permiten construir AST's conforman el llamado proyecto Corejava de CZT.

5.3.2.3. CZT Typechecker

Una especificación parseada puede ser sometida a un chequeo de tipos usando CZT Typechecker. Las reglas de inferencia de tipo que utiliza esta utilidad están definidas en el Estándar ISO de Z. CZT Typechecker está implementado como un visitante de una estructura AST. Este visita cada término en el árbol para determinar su tipo (si lo tiene) y detectar errores de tipo. Si no hay errores de tipo, el término es modificado para mantener un registro de su tipo. Si en cambio, hay errores, el término es modificado registrando las posiciones de éstos, y manteniendo una lista de referencias a todos ellos para luego poder fácilmente imprimir mensajes de error.

5.3.2.4. CZT ZLive

ZLive es un subproyecto de CZT que provee un animador para evaluar expresiones y predicados Z. En general, un animador es una utilidad que, dada la entrada para un sistema especificado formalmente, calcula la salida que se obtiene al ejecutarlo con tal entrada. De esta forma, permite ver cómo se comporta un sistema sin necesidad de codificarlo, por lo que es útil para validar especificaciones. En particular, ZLive permite utilizar una interfaz en modo texto que soporta Z en los formatos LaTeX y Unicode, aunque como fue en el caso de Fastest, las clases que lo implementan pueden servir para el desarrollo de otras herramientas. ZLive no cubre actualmente todo el lenguaje Z por completo. Por ejemplo, las definiciones axiomáticas no son soportadas. Por otra parte, es relativamente poco eficiente la evaluación de cada expresión o predicado que se le suministra, lo que lleva a no desestimar la utilización de otro animador en el futuro.

5.4. Implementando el testing funcional en Fastest

En esta sección se hará una descripción de las decisiones que se han tomado en el desarrollo de Fastest para poder implementar las etapas del proceso de testing explicado en la Sección 5.1. Como allí se mencionó, la versión actual de la herramienta tiene implementada hasta la etapa de generación de casos de prueba, inclusive, de las que aparecen en la Figura 5.1.

5.4.1. Preprocesando la especificación

Una vez que el usuario ha decidido cargar una especificación Z al contexto de Fastest, indicando el archivo LaTeX que la contiene, la herramienta se encarga de parsear la especificación para transformarla en una estructura de objetos Java, a partir de la cual pueda extraerse y modificarse más fácilmente la información de sus partes. Para tal fin, Fastest hace uso de las clases provistas por el CZT Parser (ver Sección 5.3.2.2), el cual al parsear el archivo LaTeX, genera una estructura de objetos AST (también descripto en 5.3.2.2) que representa la especificación contenida en tal archivo. Tener en cuenta que si bien una especificación está representada como un AST, cada una de sus partes, incluídos sus esquemas, también se representan con AST's. Obviamente, el AST de cualquier término de una especificación es subárbol del AST de la especificación.

El proceso de parseo, en el cual se realiza una verificación sintáctica de la especificación ingresada, continúa con un chequeo de los tipos de la misma. Esta revisión se realiza a través de la funcionalidad

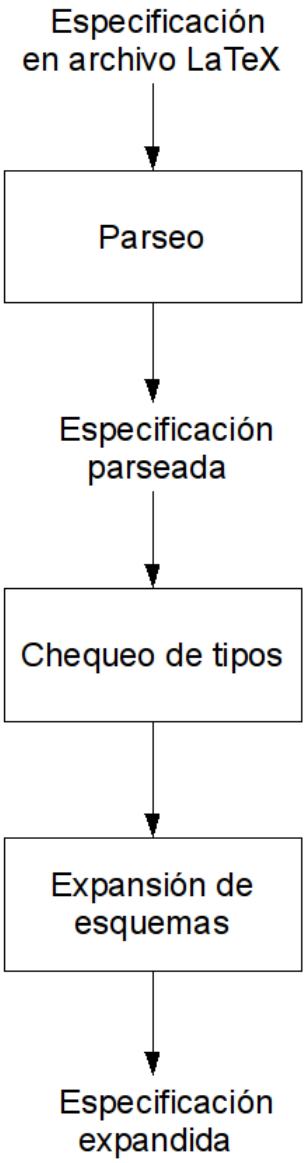


Figura 5.8: Preprocesamiento de una especificación. El archivo LaTeX con la especificación se parsea, y al resultado se le hace un chequeo de tipos. En caso de superar este control, se obtiene una versión modificada de la especificación (sin eliminar la original), con todos sus esquemas completamente expandidos.

provista por las clases de otra utilidad ya mencionada del framework CZT, el CZT Typechecker (ver Sección 5.3.2.3). Si la especificación no supera esta etapa por contener algún error de tipo, se reportan los errores apropiados; en caso contrario, el preprocesamiento continúa.

El preprocesamiento de la especificación finaliza con la obtención de una versión alternativa de ella, también expresada como un AST, y la cual se construye tomando la especificación original y expandiendo completamente todos sus esquemas Z[4]. Esta funcionalidad de expansión se implementa en un módulo que será presentado en la sección "Diseño de Fastest", y consiste en hacer una copia del AST de la especificación y modificar apropiadamente los AST's de los esquemas de la copia.

La especificación expandida es fundamental para las demás etapas del proceso de testing que aquí se implementa, pues permite considerar cada esquema Z en forma independiente de los demás. Esto significa que al querer acceder al contenido de un esquema, a través de un recorrido por los objetos que conforman su AST, no será necesario buscar información en el AST de ningún otro esquema. Lamentablemente, es usual que se necesite información de algún tipo definido globalmente, como un tipo básico, un tipo libre o una abreviatura. En esos casos no queda más opción que recorrer el AST de la especificación.

5.4.2. Generando los árboles de prueba

Cuando ya se ha ingresado la especificación (y ésta ha sido preprocesada por Fastest), la lista de operaciones que se desea testear, y la lista de tácticas que se aplicarán a cada una de éstas, el usuario puede ordenarle a la herramienta la generación de los árboles de prueba. Dados una operación y su lista de tácticas, el proceso comienza con la obtención del *VIS* de la operación. Como se indicó en la Sección 3.2, el *VIS* de una operación *Op* está dado por el subconjunto del espacio de entrada *IS* que satisface la precondición de *Op*. Entonces, en primer lugar, se obtiene el *IS* de *Op*, formando un nuevo esquema a partir de las declaraciones de variables de estado no primadas y de entrada de *Op*. Luego, dado que la precondición de una operación se obtiene ocultando sus variables de estado primadas y sus variables de salida, se decidió implementar su cálculo tomando el (AST del) predicado de *Op*, expresándolo en forma normal disyuntiva, y eliminando cada (AST que corresponda a un) predicado literal en el cual aparezca alguna variable primada o de salida. Combinando las declaraciones de variables de *IS* con el predicado obtenido se tiene el *VIS* de *Op*.

Una vez hecho esto, es necesario comenzar con la aplicación de tácticas de testing para generar el árbol de pruebas. En la Sección 3.5 se mostró que al aplicar una táctica a una clase de prueba es posible obtener nuevas clases de prueba y que la práctica usual es empezar desde el *VIS* de la operación a testear. Por otra parte, se comentó la posibilidad de aplicar las tácticas de testing de forma independiente en las clases de prueba, pudiendo aplicar una táctica particular en algunas clases pero no en otras. En esta versión de Fastest, el mecanismo no es tan flexible. En su lugar, como un primer enfoque, se ha adoptado una estrategia de aplicación de tácticas en la cual, en un primer paso, dados la lista de tácticas y el *VIS* de la operación, se itera sobre las tácticas de la lista hasta que alguna de ellas permita generar nuevas clases de prueba. Luego, se repite el proceso recursivamente

[4]Se puede ver el concepto de expansión e inclusión de esquemas en el Apéndice A.

en cada nueva clase de prueba, pero considerando en cada caso la lista sin la última táctica aplicada, de forma de no intentar utilizar dos veces la misma táctica. Este proceso termina cuando en cada clase de prueba que sea hoja del árbol no puedan aplicarse más tácticas, o bien se hayan aplicado todas.

En la sección correspondiente al Diseño de Fastest se mencionará como en el mismo fue concebida la posibilidad de cambiar fácilmente la estrategia de aplicación de tácticas de testing para una operación. Por ejemplo, una variante sería considerar la lista de tácticas de forma tal de aplicar la táctica que se encuentra en la posición i-ésima de la lista al i-ésimo nivel del árbol: la táctica 0 se aplica al *VIS*, la táctica 1 a las clases de prueba del primer nivel de nodos del árbol, la táctica 2 a las del segundo, etc.

5.4.3. Generación de casos de prueba abstractos

A partir de la obtención de los árboles de pruebas ya es posible comenzar con el proceso de generación de casos de prueba abstractos. Fastest considera la estructura de cada árbol de pruebas y la recorre hasta llegar a las hojas del árbol, las que contienen a las clases de prueba más apropiadas para la búsqueda de casos de prueba. Cada vez que se alcanza una de éstas clases de prueba, se inicia la búsqueda de un caso de prueba contenido en la clase.

En Fastest, la idea detrás de la generación de casos de prueba consiste, básicamente, en construir conjuntos finitos de valores para las variables de la clase de prueba de interés, y verificar si el predicado de la clase de prueba se satisface para alguna combinación de esos valores. Dada una variable x de tipo T, se dirá que un conjunto finito de elementos de tipo T es un *modelo finito del tipo* T y se lo notará MF_T. Por otra parte, dada una clase de prueba *ClaseDePrueba*, un *modelo finito de ClaseDePueba*, $MF_{ClaseDePueba}$, será el producto cartesiano de los modelos finitos de las variables declaradas en la clase de prueba. Evidentemente, al considerar sólo algunos elementos del tipo de cada una de las variables de una clase de prueba, es posible no encontrar ninguna combinación de valores que satisfaga el predicado de la clase, por lo cual el proceso de generación puede fracasar. A esto se suma que, como la versión actual de la herramienta no soporta la poda de los árboles de prueba, es bastante común que se intente encontrar casos de prueba para clases de prueba vacías. Sin embargo, dado que en la práctica las especificaciones Z suelen ser similares entre sí, los modelos finitos pueden construirse de forma tal que contengan los valores más usuales de cada tipo, por lo que las posibilidades de que Fastest encuentre casos de prueba, en clases no vacías, aumenta considerablemente.

Los siguientes son ejemplos de modelos finitos para algunos tipos dados:

- $\{-1, 0, 1\}$ es un modelo finito del tipo $\mathbb{N}$.
- Dado un tipo básico *CHAR* y las variables $char1, char2, char3 : CHAR$, definidas a través de una definición axiomática, $\{char1, char2, char3\}$ es un modelo finito del tipo *CHAR*.
- Dado un tipo libre $COLOR ::= blanco \mid negro \mid rojo \mid azul \mid amarillo$, $\{blanco, rojo, amarillo\}$ es un modelo finito de *COLOR*.
- Considerando el tipo *COLOR* del ítem anterior, $\{(-1, rojo), (1, azul), (1, blanco)\}$ es un modelo finito del tipo $\mathbb{N} \times COLOR$.

En relación a la evaluación de predicados, es necesario mencionar que la herramienta hace uso de la funcionalidad provista por la utilidad CZT ZLive (ver Sección 5.3.2.4). Para cada predicado que resulta de reemplazar variables por valores tomados del modelo finito de una clase de prueba, ZLive permite determinar si tal predicado equivale a *true* o a *false*. Cada evaluación que realiza ZLive es una operación costosa. Por otro lado, puede requerirse una gran cantidad de evaluaciones de predicados, para encontrar un caso de prueba en una clase de prueba, o bien para determinar el fracaso de la generación, cuando no queden evaluaciones por realizar. Estos aspectos hacen que todo el proceso de generación de casos de prueba suela requerir un tiempo considerable, aún cuando no sea posible encontrar ningún caso. En consecuencia, es probable que en un futuro cercano se desarrolle un propio evaluador de expresiones y predicados más eficiente que el utilizado actualmente. De esta manera se tendría una mejor desempeño en todo el proceso de testing implementado por Fastest.

Como una forma de disminuir la cantidad de evaluaciones de predicados a realizar en las búsquedas de casos de prueba, Fastest tiene en cuenta lo siguiente:

- Si una variable declarada en la clase de prueba no aparece en el predicado de la clase, el correspondiente modelo finito se construye con un sólo elemento. Esto se debe a que sea cual sea el valor que tome la variable, éste no tendrá influencia en el resultado de la evaluación del predicado que se obtiene de reemplazar las variables de tal predicado por los valores que les corresponden. Y al tener una variable con un modelo finito de menor tamaño, el modelo finito de toda la clase de prueba es de menor tamaño, por lo que la cantidad máxima de evaluaciones que eventualmente se harán también disminuye.
- Si una variable aparece igualada a un valor constante en el predicado de la clase analizada, el modelo finito que se asociará a la variable sólo contendrá tal valor. Como en el ítem anterior, esto reduce el número de evaluaciones, lo que hace más eficiente el proceso de generación de casos de prueba.

A continuación se hará una descripción de la manera en que, en Fastest, se construyen los modelos finitos para los tipos con los que pueden declararse variables. La construcción en cada caso dependerá de la forma que tenga el tipo correspondiente. En el código de la herramienta se define un valor, $TamanioMF$, que indica una sugerencia para el tamaño de los modelos finitos, y que en esta versión de Fastest no puede modificarse desde la interfaz de usuario. Hay que tener en cuenta que este parámetro no se utiliza en la construcción de los modelos finitos para todos los tipos, sino sólo para los más simples, como se verá en lo que sigue.

5.4.3.1. Tipos básicos

El modelo finito de un tipo básico A, el cuál se define a través de:

$$[A]$$

está dado por el conjunto de constantes $\{a_1, a_2, ..., a_{TamanioMF}\}$.

5.4.3.2. Tipos libres

El modelo finito de un tipo libre enumerado B, definido por:

$$B ::= b_1 \mid b_2 \mid ... \mid b_n$$

está dado por el conjunto de n elementos $\{b_1, b_2, ..., b_n\}$, tomados de la definición del tipo. En este caso el valor de *TamanioMF* no es utilizado por Fastest en la determinación del modelo finito. Por el momento, la herramienta sólo soporta tipos libres que sean enumerados, es decir, aquellos que no definan tipos inductivos.

5.4.3.3. Números naturales ($\mathbb{N}$)

El modelo finito para el tipo $\mathbb{N}$, $MF_{\mathbb{N}}$, se define como el conjunto $\{0, 1, 2, ..., \mathit{TamanioMF} - 1\}$ de *TamanioMF* elementos.

5.4.3.4. Números enteros ($\mathbb{Z}$)

El modelo finito para el tipo $\mathbb{Z}$, $MF_{\mathbb{Z}}$ se define como el conjunto:

- $\{0, -1, 1, -2, 2, ..., \lfloor \mathit{TamanioMF}/2 \rfloor\}$, si *TamanioMF* es un número impar.
- $\{0, -1, 1, -2, 2, ..., -(\mathit{TamanioMF}/2)\}$, si *TamanioMF* es un número par.

de *TamanioMF* elementos.

5.4.3.5. Conjuntos por extensión

El modelo finito para el tipo dado por el conjunto por extensión $A = \{a_0, a_1, ..., a_{n-1}\}$, MF_A, se define simplemente como el mismo conjunto de n elementos. En este caso el valor de *TamanioMF* tampoco es utilizado por Fastest en la determinación de este modelo finito.

5.4.3.6. Rango de valores

El modelo finito para el tipo $a..b$, con $a, b \in \mathbb{Z}$, $MF_{a..b}$, está dado por el conjunto $\{a, a + 1, ..., b\}$ de $b - a + 1$ elementos. Fastest tampoco utiliza el valor de *TamanioMF* para construir este modelo finito.

5.4.3.7. Conjunto de partes

Dado un tipo A, el modelo finito para el tipo $\mathbb{P}\, A$, $MF_{\mathbb{P}\, A}$, se define como el conjunto de partes del modelo finito de A. Por ejemplo, si el modelo finito de A es $\{a_1, a_2, a_3\}$ entonces el modelo finito de $\mathbb{P}\, A$ será $\{\{\}, \{a_1\}, \{a_2\}, \{a_3\}, \{a_1, a_2\}, \{a_1, a_3\}, \{a_2, a_3\}, \{a_1, a_2, a_3\}\}$. Fastest no utiliza el valor de *TamanioMF* para construir este modelo finito.

5.4.3.8. Producto cartesiano

Dados los tipos $A_1, A_2, ..., A_n$, el modelo finito para el tipo $A_1 \times A_2 \times ... \times A_n$, $MF_{A_1 \times A_2 \times ... \times A_n}$, se calcula recursivamente como el producto cartesiano de los modelos finitos de $A_1, A_2, ..., A_n$. Fastest no utiliza el valor de *TamanioMF* para construir este modelo finito.

5.4.3.9. Relaciones binarias

Dados los tipos A y B, el modelo finito del tipo $A \leftrightarrow B$ (relaciones entre A y B), $MF_{A \leftrightarrow B}$, se calcula recursivamente como el modelo finito de $\mathbb{P}(A \times B)$, $MF_{\mathbb{P}(A \times B)}$. Fastest no utiliza el valor de *TamanioMF* para construir este modelo finito.

5.4.3.10. Funciones totales

Dados los tipos A y B, el modelo finito del tipo $A \rightarrow B$ (funciones totales entre A y B), $MF_{A \rightarrow B}$, se calcula recursivamente como el tipo de las funciones totales entre el modelo finito de A y el modelo finito de B. En símbolos, si el modelo finito de A, MF_A, es $\{a_0, a_1, ...a_{n-1}\}$ y el modelo finito de B, MF_B, es $\{b_0, b_1, ...b_{m-1}\}$, entonces el modelo finito de $A \rightarrow B$, $MF_{A \rightarrow B}$, se define como el conjunto $\{\{(a_0, x_0), (a_1, x_1), ..., (a_{n-1}, x_{n-1})\} \mid x_0, x_1, ..., x_{n-1} \in MF_B\}$. Puede probarse que se cumple que $\#MF_{A \rightarrow B} = m^n$. Fastest no utiliza el valor de *TamanioMF* para construir este modelo finito.

5.4.3.11. Funciones parciales

Dados los tipos A y B, el modelo finito del tipo $A \nrightarrow B$ (funciones parciales entre A y B), $MF_{A \nrightarrow B}$, se calcula recursivamente como el tipo de las funciones parciales entre el modelo finito de A y el modelo finito de B. En símbolos, si el modelo finito de A, MF_A, es $\{a_0, a_1, ...a_{n-1}\}$ y el modelo finito de B, MF_B, es $\{b_0, b_1, ...b_{m-1}\}$, entonces el modelo finito de $A \nrightarrow B$ se define como el conjunto $\bigcup_{A' \in \mathbb{P}\, MF_A} \{A' \rightarrow MF_B\}$. Puede probarse que se cumple que $\#MF_{A \rightarrow B} = \sum_{i=0}^{n} m^i$. Fastest no utiliza el valor de *TamanioMF* para construir este modelo finito.

5.4.3.12. Secuencias

Dado un tipo A, el modelo finito para el tipo seq A (secuencias de elementos de tipo A), $MF_{\text{seq}\, A}$, se calcula recursivamente en función del modelo finito de A, como se ilustra a través del siguiente ejemplo. Si el modelo finito de A, MF_A, es $\{a_0, a_1, a_2\}$, entonces:

$$
\begin{aligned}
&MF_{\text{seq}\, A} = \{\langle\rangle\ , \langle a_0\rangle, \langle a_1\rangle, \langle a_2\rangle, \langle a_0, a_1\rangle, \langle a_0, a_2\rangle, \langle a_1, a_2\rangle, \langle a_0, a_1, a_2\rangle,\\
&\langle a_0, a_0\rangle\ , \langle a_0, a_0, a_0\rangle\ ,\\
&\langle a_1, a_1\rangle\ , \langle a_1, a_1, a_1\rangle\ ,\\
&\langle a_2, a_2\rangle\ , \langle a_2, a_2, a_2\rangle\}
\end{aligned}
$$

donde, una vez más, Fastest no utiliza el valor de *TamanioMF* para construir este modelo finito.

5.4.3.13. Tipos no soportados en la versión actual

La versión actual de Fastest sólo permite que la clase de prueba para la que quiere buscarse un caso de prueba tenga declaraciones de variables cuyos tipos se encuentren entre los recién presentados.

Por ejemplo, una variable no puede ser de tipo A, si A es un esquema definido en la especificación. Algunos de los tipos aún no soportados, dados los tipos A y B son $A \nrightarrow B$, $A \rightarrowtail B$, $A \rightarrowtail\!\!\!\!\!\rightarrow B$, $\mathbb{P}_1 A$ y $\mathbb{N}_1$.

5.5. Diseño de Fastest

En esta sección se presentarán los módulos más importantes de los que conforman el diseño de la herramienta, el cual es orientado a objetos. Se mostrarán las decisiones que se han tomado para elaborarlo y se hará una breve explicación de los patrones de diseño utilizados. A través de un diseño de esta naturaleza es posible aislar detalles de implementación para favorecer la introducción de futuras modificaciones.

5.5.1. Patrones de diseño

Durante el desarrollo de Fastest se emplearon patrones de diseño con el objetivo de obtener una codificación más sencilla de comprender y de ampliar. Un *patrón de diseño* describe, por un lado, un problema de diseño que ocurre regularmente al construir sistemas de software, y por otro, una solución al problema, la cual es efectiva (ya se resolvió el problema en ocasiones anteriores) y reusable (se puede aplicar a diferentes problemas de diseño en distintas circunstancias) [32].

Los patrones de diseño que se aplicaron en la construcción de la herramienta fueron:

- *Singleton*, para asegurar la existencia de únicas instancias de la clase que almacena la información necesaria para conectarse a los servidores de cómputo; de la clase que implementa, en cada cliente, el administrador de eventos del estilo arquitectónico de Invocación Implícita; y de la clase que agrupa las particiones estándar de los operadores matemáticos.
- *Command*, para encapsular las solicitudes de Invocación Implícita (dadas por un componente, un procedimiento del componente, y los parámetros que deben pasarse al procedimiento), como objetos.
- *Composite*, para representar la composición de nodos que forman los árboles de prueba.
- *Visitor*, para agregar operaciones sobre la estructuras de los árboles de pruebas fácilmente, sin tener que modificar las clases que definen a éstos árboles. También se utiliza para agregar nueva funcionalidad a las estructuras de objetos AST (ver Sección 5.3.2.2) que representan elementos sintácticos del lenguaje Z.
- *Strategy*, a fin de permitir la existencia de varios algoritmos para la aplicación de tácticas de testing y para la generación de casos de prueba abstractos.
- *Iterator*, para recorrer colecciones de elementos sin exponer las estructuras internas de éstas.

cuyas relaciones con los módulos del diseño pueden encontrarse en la Guía de Módulos y la Especificación de Interfaces adjunta.

5.5.2. El subsistema de Invocación Implícita

En la Sección 5.2.2.1 se vio que el estilo de Invocación implícita se implementa a través de un administrador de eventos que captura los eventos anunciados y realiza llamadas a los procedimientos interesados en tales eventos. En Fastest, cada uno de los eventos se representa con una instancia del módulo `Event`, cuya interfaz se presenta a continuación:

Module	**Event**
exportsproc	setEventName(**i** String)
	getEventName(): String

Cada tipo de evento deberá estar representado por una clase que herede de `Event` y que agregue los procedimientos para consultar y establecer los parámetros particulares que acompañan al evento. Por ejemplo, la interfaz del módulo `TTreeGenerated`, el cual es el tipo de evento que debe anunciarse después de generarse el árbol de pruebas para una operación, es:

Module	**TTreeGenerated inherits from** Event
imports	TClassNode
exportsproc	setOpName(**i** String)
	getOpName(): String
	setTTree(**i** TClassNode)
	getTTree(): TClassNode

en la que no se especifican los procedimientos `setEventName` y `getEventName`, por heredarlos de Event, pero agrega procedimientos para consultar y establecer los parámetros propios de `TTreeGenerated`, que son un árbol de pruebas y el nombre de la operación para la que se generó un árbol. Es fundamental que el constructor del módulo establezca, correctamente, el valor de la variable de estado que almacena el nombre del evento que le corresponda. La importancia de este nombre se debe a que es el que se utiliza para identificar de qué tipo de evento se trata cuando es anunciado, y que permita invocar a los procedimientos interesados en el evento.

Precisamente, al momento de ser anunciado el evento, es el administrador de eventos (implementado en una instancia única de `EventAdmin`) el que debe consultar su tabla de eventos (instancia de `EventTable`) para invocar los procedimientos interesados en el evento anunciado. Como se describió anteriormente, `EventAdmin` construye su tabla de eventos leyendo un archivo de configuración que se describirá más adelante. La interfaz del módulo `EventAdmin` es la siguiente:

Module	**EventAdmin**
imports	ClientUI, Event, IIComponent, File, EventTable
exportsproc	getInstance(): EventAdmin
	announceEvent(**i** Event)
	readFile()
	setFile(**i** File)
	getFile(**i** File)

La subrutina `getInstance` es necesaria para implementar el patrón de diseño *Singleton* en relación al administrador de eventos, el que permite forzar la existencia de una única instancia de `EventAdmin` en el sistema. `setFile` y `getFile` tienen como objetivo, respectivamente, establecer y obtener una referencia al archivo de configuración de eventos. La subrutina `readFile` parsea tal archivo y construye la tabla de eventos en el estado del módulo `EventAdmin`. En esta tabla se listan las ternas (evento, componente interesado en el evento, subrutina del componente a ser llamada al lanzarse el evento). Además de completar la tabla de eventos, la subrutina `readFile` tiene la responsabilidad de crear cada uno de los componentes indicados en la tabla. La subrutina `announceEvent` debe invocar a cada método interesado en el evento que se pasa como argumento, es decir, a cada método indicado por los subscriptores listados en la tabla de eventos. En la implementación, se decidió hacer cada invocación en un thread diferente, para no forzar un orden entre las distintas invocaciones.

Respecto a los componentes que pueden interesarse en eventos, es importante destacar que cada uno de ellos se representa como una instancia de un módulo que herede de `IIComponent`, cuya interfaz se muestra a continuación:

Module	**IIComponent**
imports	ClientUI, Event
exportsproc	manageEvent(**i** Event)
	setMyClientUI(**i** ClientUI)
	getMyClientUI(): ClientUI

El módulo `ClientUI` que aquí se menciona es el que implementa la interfaz de usuario de Fastest. Como los componentes interesados en eventos pueden necesitar acceder a la interfaz para solicitar o presentar información al usuario, es necesario que `IIComponent` tenga una referencia a ella. A través de los procedimientos `setMyClientUI` y `getMyClientUI` es posible establecer y obtener esta referencia. Por su parte, el procedimiento `manageEvent` es el que cada componente puede designar como interesado en algún evento, y es el que será llamado por el administrador de eventos en caso de anunciarse tal evento.

En este prototipo de Fastest, los eventos utilizados son los siguientes:

- `AllTCasesGenerated`, que debe ser lanzado cuando se haya concluido el intento de generación de todos los casos de prueba solicitados.
- `AllTTreesGenerated`, que debe ser lanzado cuando se haya concluido el intento de generación de todos los árboles de pruebas solicitados.
- `AllTCasesRequested`, que debe ser lanzado al solicitar el cálculo de todos los casos de prueba para las operaciones seleccionadas (a través del comando `genalltca`, a explicarse en la Sección 6.3.7). Se lanza este evento por cada árbol de pruebas previamente generado.
- `SpecLoaded`, que debe ser lanzado cuando el usuario haya cargado, correctamente, la especificación del programa a testear.

- `TCaseGenerated`, que debe ser lanzado después de haberse terminado el intento de generación del caso de prueba abstracto de cierta clase de prueba de una operación particular.
- `TCaseRequested`, que debe ser lanzado al solicitar el cálculo de un caso de prueba para cierta clase de prueba.
- `TTreeGenerated`, que debe ser lanzado después de generarse el árbol de pruebas de cierta operación.
- `TTreeRequested`, que debe ser lanzado cuando se haya cargado la información correspondiente a una operación a testear, esto es, su nombre y lista de tácticas a aplicar.

Por otra parte, actualmente los componentes que se interesan en eventos son los que se muestran a continuación:

- `TTreeGen`. Este módulo es el encargado de organizar la generación de árboles de prueba. Se interesa en los eventos de tipo `SpecLoaded` y `TTreeRequested`.
- `TClassExtractor`. Se encarga de procesar eventos de tipo `AllTCasesRequested` lanzando, para cada uno, sucesivos eventos de tipo `TCaseRequested`, uno para cada clase de prueba que sea hoja del árbol correspondiente. Esto permite, dados la solicitud de generación de los casos de prueba para una operación y el árbol de pruebas de la operación, solicitar a través de eventos de tipo `TCaseRequested` la generación de casos de pruebas para las clases de prueba que sean hojas del árbol.
- `TCaseGenClient`. Se encarga de manejar solicitudes para la generación de casos de prueba abstractos, las cuales se indican en instancias de `TCaseRequested`. Por cada solicitud, este módulo inicia el proceso de generación de un caso de prueba en un thread diferente, para favorecer cuestiones de desempeño.
- `Controller`. Este módulo tiene la responsabilidad de mantener referencias a árboles de prueba, casos de prueba y demás resultados del proceso testing. También mantiene referencias a la especificación y a la lista de operaciones a ser testeadas junto con la lista de tácticas a aplicar. Se interesa en los eventos `AllTCaseGenerated`, `AllTTreeGenerated`, `TCaseGenerated` y `TTreeGenerated`.

5.5.3. Aplicando el patrón de diseño Visitor a los AST

En la Sección 5.3.2.2 se describieron las facilidades provistas por el proyecto CZT respecto a los árboles de sintaxis abstracta, AST. A través de ellos es posible representar las distintas construcciones del lenguaje de especificación Z como estructuras de objetos Java relacionados entre sí. Una característica muy importante de las clases e interfaces de las que éstos objetos son instancias es que ofrecen la posibilidad de agregar nuevas operaciones fácilmente. Esto se logra a través del patrón de diseño *Visitor*.

El patrón de diseño *Visitor* [32] provee una forma de separar la estructura de un conjunto de objetos de las operaciones que se realizan sobre tales objetos. Esto permite que pueda definirse una

nueva operación sin modificar las clases AST. Para definir una nueva operación, todo lo que se necesita es implementar un visitante en una nueva clase. Por ejemplo, a continuación se presenta el código de un visitante que permite comprobar si el predicado sobre el que se aplica es una disyunción tal que el predicado que se pasa como argumento en el constructor de la clase es igual a uno de sus operandos:

```
package common.z.czt.visitors;

import net.sourceforge.czt.z.ast.Pred;
import net.sourceforge.czt.z.ast.OrPred;
import net.sourceforge.czt.z.visitor.PredVisitor;
import net.sourceforge.czt.z.visitor.OrPredVisitor;

public class PredInOrVerifier
    implements  OrPredVisitor<Boolean>, PredVisitor<Boolean>{

    private Pred myPred;

    public PredInOrVerifier(Pred pred){
        myPred = pred;
    }

    public Boolean visitOrPred(OrPred orPred){
        Pred leftPred = orPred.getLeftPred();
        Pred rightPred = orPred.getRightPred();
        boolean leftResult = false;
        boolean rightResult = false;
        if(leftPred instanceof OrPred)
            leftResult = ((OrPred)leftPred).accept(this);
        else
            leftResult = SpecUtils.areEqualTerms(myPred, leftPred);

        if(leftResult)
            return new Boolean(true);

        if(rightPred instanceof OrPred)
            rightResult = ((OrPred)rightPred).accept(this);
        else
            rightResult = SpecUtils.areEqualTerms(myPred, rightPred);

        return new Boolean(rightResult);
    }

    public Boolean visitPred(Pred pred){
        return new Boolean(false);
```

```
    }
}
```

El patrón de diseño *Visitor* estándar [32] provee un mecanismo de solicitud doble. El conjunto de objetos sobre los que se definen nuevas operaciones deben proveer un método `accept(Visitor v)`. `Visitor` es una interfaz que define métodos de visita para la clase de cada objeto que puede ser visitado (como `visitOrPred` en el ejemplo anterior). Al invocar el método `accept` de un objeto, pasándole como argumento un visitante particular, se llama al método de visita, del visitante, que corresponde a la clase de ese objeto. Esto permite que se definan diferentes implementaciones de la interfaz Visitor para realizar distintas operaciones sobre los objetos.

La desventaja de este enfoque es que es difícil agregar nuevas clases AST, porque cada nueva clase AST implica que se debe agregar un nuevo método a la interfaz `Visitor`, lo que a su vez requiere modificar todas las implementaciones existentes. Como en el proyecto CZT se trabaja con la idea de ir agregando extensiones al lenguaje Z, ellos consideran que es necesario que la adición de nuevas clases AST sea una tarea sencilla [31]. Otra desventaja es que cada clase visitante necesita implementar un conjunto fijo de métodos, uno para cada clase AST, y hay un número grande de clases AST definidas en CZT. Para solucionar estos inconvenientes, CZT ha trabajado en el desarrollo de variantes del patrón de diseño *Visitor* estándar. En particular, el *CZT Visitor* incorpora las ventajas de los patrones de diseño *Acyclic Visitor* [33] y *Default Visitor* [34], junto con otras pequeñas modificaciones. Para más información acerca del patrón de diseño *CZT Visitor*, referirse a [31].

A continuación se listan los módulos que están basados en el patrón de diseño *CZT Visitor* y que permiten realizar una operación sobre todos o algunos de los términos que pueden formar parte de un AST.

AndOrPredDistributor Permite aplicar a predicados Z la propiedad distributiva de la conjunción respecto a la disyunción. Este módulo puede ser utilizado para realizar uno de los pasos a través de los cuales se lleva un predicado a Forma Normal Disyuntiva (ver Sección 3.5.1.2).

AndPredClausesExtractor Permite obtener, de una conjunción de predicados, una colección con los predicados que la conforman.

AndPredRemover Permite remover de un predicado que sea una conjunción, todas las clausulas iguales a cierto predicado p. p es un predicado que se toma como argumento en el constructor del módulo.

ContainsTermVerifier Permite verificar si un término del lenguaje Z contiene a otro.

CZTReplacer Dados dos términos Z, permite reemplazar en un tercero (al que se aplica el visitante que el módulo implemente) todas las ocurrencias del primero por ocurrencias del segundo. El término original y el que reemplazará al original se deben pasar como parámetros al constructor del módulo.

ImpliesPredRemover Permite transformar todos los predicados de la forma $p_1 \Rightarrow p_2$ en predicados de la forma $\neg\ p_1 \lor p_2$. Este módulo puede ser utilizado para realizar uno de los pasos a través de los cuales se lleva un predicado a Forma Normal Disyuntiva (ver Sección 3.5.1.2).

NegPredDistributor Permite aplicar a predicados dados las Leyes de Morgan ($\neg\ (p1 \lor p2) \Leftrightarrow \neg\ p1 \land \neg\ p2$ y $\neg\ (p1 \land p2) \Leftrightarrow \neg\ p1 \lor \neg\ p2$) y la eliminación de la doble negación ($\neg\ \neg\ p \Leftrightarrow p$). Este módulo puede ser utilizado para realizar uno de los pasos a través de los cuales se lleva un predicado a Forma Normal Disyuntiva (ver Sección 3.5.1.2).

OrPredRemover Permite remover de un predicado que sea una conjunción, todas las clausulas que sean disyunciones tales que alguno de los operandos que las conforman sea igual a cierto predicado p que se toma como argumento en el constructor del módulo.

ParamExtractor Permite, dado un término del lenguaje Z, obtener una lista con sus parámetros. Por ejemplo, al término $a = b$ lo conforman a y b.

PredInOrVerifier Permite determinar si el predicado al que se le aplica la funcionalidad es una disyunción donde uno de sus operandos es el predicado que se pasa al constructor del módulo.

PredRemover Permite remover de un predicado, que esté en Forma Normal Disyuntiva, todos los literales que contengan la ocurrencia de alguno de los nombres de variables contenidos en la lista que el constructor del módulo toma como argumento.

PrimeVarsMaker Permite primar todas las variables (es decir, concatenarles el caracter $'$) en la expresión o esquema Z visitado.

WordsFinder Permite determinar si alguna de las variables contenidas en la lista de nombres de variables dada (que se pasa como argumento al constructor de este módulo) ocurre en la expresión o esquema Z visitado.

5.5.4. Los módulos del lado del servidor

Los dos módulos que siguen tienen sus instancias sólo en los servidores de cómputo de Fastest, y tienen relación con la gestión de pedidos provenientes de los clientes de la herramienta.

ServiceManager Está encargado de iniciar el funcionamiento del servidor de cómputo correspondiente, esperando y atendiendo pedidos de servicio de parte de los clientes. Que se puedan atender varios servicios en simultáneo se debe a que cada uno se corre en un thread diferente, invocando al procedimiento `run` del módulo `ServerThread`.

ServerThread Módulo sobre el que se atiende una solicitud de servicio particular, proveniente de un cliente de Fastest. El procedimiento principal de este módulo es `run`, el que implementa el diálogo con el cliente para obtener todos los parámetros necesarios para llevar a cabo la tarea solicitada, se interrumpe mientras se procesa la actividad, y se reanuda para poder regresar al cliente el resultado del servicio.

A continuación se muestra un listado con los módulos, también propios de los servidores de cómputo, que permiten implementar la funcionalidad de generación de casos de prueba abstractos.

SchemeEvaluator Permite evaluar si el predicado de una clase de prueba es satisfecho por un conjunto dado de asignaciones de variables a expresiones.

TCaseStrategy Interfaz que abstrae las estructuras de datos y los algoritmos que implementan una estrategia de generación de casos de prueba. Este módulo está basado en el patrón de diseño *Strategy*. Es necesario definir un módulo que implemente esta interfaz para agregar a Fastest una nueva estrategia de generación de casos de prueba.

IterativeTCaseStrategy Módulo físico que oculta las estructuras de datos y los algoritmos que implementan una estrategia particular de generación de casos de prueba. La estrategia consiste en ir generando asignaciones de valores a las variables de la clase de prueba para la que se quiere obtener un caso de prueba hasta que alguna de ellas satisgafa el predicado de dicha clase de prueba. Para hacer la evaluación de cada asignación de valores se llama al procedimiento `evalSchemeSat`, del módulo `SchemeEvaluator`. `IterativeTCaseStrategy` no necesita generar el modelo finito completo para cada tipo de variable de la clase de prueba ni para cada tipo definido globalmente en la especificación. El módulo hereda de `TCaseStrategy`, dado que implementa una estrategia particular de generación de casos de prueba.

FiniteModel Interfaz que abstrae un modelo finito. Un modelo finito está asociado a un tipo particular del lenguaje Z y, haciendo las veces de un iterador, puede solicitársele, de a uno, los distintos elementos que lo conforman. Este módulo se utiliza en conjunto con la estrategia `IterativeTCaseStrategy` de generación de casos de prueba, y como ésta sugiere, las instancias de `FiniteModel` no almacenan en su estado el modelo finito completo del tipo asociado, sino que sólo mantienen la mínima información necesaria para saber qué elemento del modelo debe devolverse la próxima vez que uno sea solicitado. Tiene que haber un módulo heredero de `FiniteModel` por cada tipo diferente del lenguaje Z.

TClassFiniteModel Módulo físico que oculta las estructuras de datos y los algoritmos que implementan un modelo finito asociado a una clase de prueba para la cual se quiere obtener un caso de prueba. `TClassFiniteModel` permite ir obteniendo, uno a uno, los distintos elementos del modelo finito de la clase. Para obtener estos valores, el módulo utiliza los modelos finitos que están asociados a los tipos de las variables. Este módulo no hereda de `FiniteModel` ya que tiene una interfaz ligeramente diferente.

FiniteModelCreator Módulo físico basado en el patrón de diseño *Visitor* que permite recorrer una especificación de forma tal de ir creando, y agregando a una estructura que forma parte del estado

del módulo, un modelo finito (instancia que hereda de `FiniteModel`) por cada tipo que se defina globalmente en la especificación.

Los siguientes son los módulos que heredan de `FiniteModel` para implementar un modelo finito particular.

GivenFiniteModel Implementa un modelo finito asociado a un tipo básico del lenguaje Z. La implementación actual responde a lo explicado en la Sección 5.4.3.1.

FreeTypeFiniteModel Implementa un modelo finito asociado a un tipo libre del lenguaje Z. La implementación actual responde a lo explicado en la Sección 5.4.3.2.

NatFiniteModel Implementa un modelo finito asociado al tipo del lenguaje Z que corresponde a los números naturales. La implementación actual responde a lo explicado en la Sección 5.4.3.3.

IntFiniteModel Implementa un modelo finito asociado al tipo del lenguaje Z que corresponde a los números enteros. El módulo es heredero de FiniteModel ya que implementa un modelo finito particular. La implementación actual responde a lo explicado en la Sección 5.4.3.4.

SetFiniteModel Implementa un modelo finito asociado al tipo de los conjuntos enumerados del lenguaje Z. El módulo es heredero de FiniteModel ya que implementa un modelo finito particular. La implementación actual responde a lo explicado en la Sección 5.4.3.5.

RangeFiniteModel Implementa un modelo finito asociado al tipo del lenguaje Z que corresponde a un rango de valores y que se indica de la forma $a \,..\, b$. El módulo es heredero de FiniteModel ya que implementa un modelo finito particular. La implementación actual responde a lo explicado en la Sección 5.4.3.6.

PowerFiniteModel Implementa un modelo finito asociado al tipo potencia del lenguaje Z. La implementación actual responde a lo explicado en la Sección 5.4.3.7.

ProdFiniteModel Implementa un modelo finito asociado al tipo producto cartesiano del lenguaje Z. La implementación actual responde a lo explicado en la Sección 5.4.3.8.

RelFiniteModel Implementa un modelo finito asociado al tipo de las relaciones del lenguaje Z. La implementación actual responde a lo explicado en la Sección 5.4.3.9.

TFuncFiniteModel Implementa un modelo finito asociado al tipo de las funciones totales del lenguaje Z. La implementación actual responde a lo explicado en la Sección 5.4.3.10.

PFuncFiniteModel Implementa un modelo finito asociado al tipo de las funciones parciales del lenguaje Z. La implementación actual responde a lo explicado en la Sección 5.4.3.11.

SeqFiniteModel Implementa un modelo finito asociado al tipo de las secuencias del lenguaje Z. La implementación actual responde a lo explicado en la Sección 5.4.3.12.

5.6. Guía para introducir cambios en la herramienta

Con esta información se pretende nutrir al desarrollador de los conocimientos necesarios para que éste pueda sumar nuevas funcionalidades a la herramienta o modificar las ya existentes.

5.6.1. Agregando nuevos eventos y componentes en los clientes

Para agregar un nuevo evento al diseño e implementación de un cliente de Fastest, se debe realizar lo siguiente:

- Crear una clase que herede de la clase `Event`.
- Si hay algún componente interesado en el evento, modificar el archivo de configuración cuyo nombre y ubicación, respecto al directorio raíz de la distribución de Fastest (ver Sección 6.2), es `lib/conf/eventtable.conf`.

En la versión actual de Fastest, este archivo está compuesto por las siguientes líneas:

```
specLoaded client.blogic.testing.ttree.TTreeGen manageEvent
tTreeRequested client.blogic.testing.ttree.TTreeGen manageEvent
tTreeGenerated client.blogic.management.Controller manageEvent
allTCasesRequested client.blogic.testing.tcasegen.TClassExtractor manageEvent
tCaseRequested client.blogic.testing.tcasegen.TCaseGenClient manageEvent
tCaseRequested client.blogic.management.Controller manageEvent
tCaseGenerated client.blogic.management.Controller manageEvent
allTCasesGenerated client.blogic.management.Controller manageEvent
tCaseStrategySelected client.blogic.testing.tcasegen.TCaseGenClient manageEvent
```

donde puede notarse que en cada una se indica el nombre del evento, el nombre de la clase del componente interesado en el evento (con la ruta completa del paquete Java que la contiene) y el nombre del método que debe invocarse al anunciarse el evento. El componente interesado en un evento debe ser necesariamente heredero de `IIComponent`.

Por lo tanto, para agregar un nuevo componente a los clientes de Fastest es necesario crear un heredero de `IIComponent` y agregar una línea apropiada al archivo de configuración de la tabla de eventos, una por cada evento de interés. El nuevo módulo deberá implementar cada uno de los métodos que estén interesados en algún evento de forma tal que pueda reaccionar apropiadamente al anunciarse el evento asociado.

5.6.2. Agregando nuevas tácticas de testing

Para lograr agregar una nueva táctica de testing el desarrollador debe simplemente crear una nueva clase que implemente la interfaz `Tactic`, la cual se describe a continuación:

Module	**Tactic**
imports	OpScheme, TClass
exportsproc	applyTactic(**i** TClass): List(TClass)
	setOriginalOp(**i** OpScheme)
	getOriginalOp(): OpScheme
	setOriginalOp(**i** OpScheme)
	getOriginalOp(): OpScheme
	parseArgs(**i** String): Bool

Para implementar esta interfaz hay que tener en cuenta lo siguiente:

- La subrutina `applyTactic` devuelve la lista de clases de prueba (instancias de `TClass`) que se obtienen al aplicar la táctica a la clase de prueba que se pasa como argumento. Si la táctica no genera nuevas clases de prueba para la clase de prueba dada, la subrutina devuelve la lista vacía.
- La subrutina `setOriginalOp` simplemente establece una referencia, en el estado del módulo, al esquema de la operación (instancia de `OpScheme`). `getOriginalOp` devuelve una referencia al esquema de operación.
- La subrutina `setSpec` establece una referencia, en el estado del módulo, a la especificación del sistema a testear (instancia de `Spec`, proveniente del framework CZT). `getSpec` devuelve una referencia a la especificación.
- La subrutina `parseArgs` permite parsear los parámetros que el usuario ingresó al agregar la táctica, utilizando el comando `addtactic` (ver Sección 6.3.5). `parseArgs` toma la cadena con todos los parámetros y si éstos son correctos, modifica el estado de la clase correspondiente y devuelve *true*. Si no son correctos, devuelve *false*.

5.6.3. Agregando comandos del lado del cliente

Para agregar un nuevo comando a la interfaz en modo texto de los clientes de Fastest, el desarrollador tiene que realizar lo siguiente:

- Crear una clase que implemente la interfaz `Command`.
- Agregar una línea al archivo de configuración de comandos cuyo nombre y ubicación, respecto al directorio raíz de los fuentes de Fastest, es `client/presentation/commands/commands.properties`.

La interfaz `Command` se especifica de la siguiente manera:

Module	**Command**
imports	ClientTextUI
exportsproc	run(**i** ClientTextUI, **i** String)

donde puede notarse que cuenta con un único procedimiento, y es el que debe implementar la funcionalidad del comando. El primer parámetro que se le pasa al procedimiento es una referencia al objeto que representa la interfaz de usuario utilizada (instancia de `ClientTextUI`). Mientras tanto, el segundo parámetro contiene la cadena que ingresó el usuario para ejecutar el comando, la cual indica los parámetros del mismo.

Con respecto al archivo de configuración de comandos mencionado, en la versión actual de Fastest, el mismo está compuesto por las siguientes líneas:

```
loadspec = client.presentation.commands.LoadSpecCommand
showspec = client.presentation.commands.ShowSpecCommand
showloadedops = client.presentation.commands.ShowLoadedOpsCommand
selop = client.presentation.commands.SelOpCommand
showselops = client.presentation.commands.ShowSelOpsCommand
unselop = client.presentation.commands.UnSelOpCommand
unselallops = client.presentation.commands.UnSelAllOpsCommand
addtactic = client.presentation.commands.AddTacticCommand
showsch = client.presentation.commands.ShowSchCommand
genalltt = client.presentation.commands.GenAllTTCommand
genalltca = client.presentation.commands.GenAllTCaseCommand
showtt = client.presentation.commands.ShowTTCommand
reset = client.presentation.commands.ResetCommand
help = client.presentation.commands.ShowHelpCommand
version = client.presentation.commands.ShowVersionCommand
```

donde es posible apreciar que cada una de ellas es una igualdad entre dos cadenas. El miembro de la izquierda de cada igualdad es la palabra que el usuario debe tipear para ejecutar el correspondiente comando; el miembro de la derecha es la clase (con la ruta completa del paquete Java que la contiene) que implementa la funcionalidad del comando.

Capítulo 6

Utilizando Fastest

En este capítulo se presentará un manual de usuario con el cual es posible adquirir el conocimiento necesario para utilizar correctamente la herramienta. Previamente, se indicarán los requerimientos de sistema y se dará un listado con las utilidades y documentación relacionados a la aplicación y que acompañan a este informe.

6.1. Requerimientos

Esta herramienta soporta los siguientes sistemas operativos:

- UNIX/LINUX: las distribuciones de Linux más utilizadas, Solaris y FreeBSD.
- Windows: en sus versiones 98, XP y Vista.

y para poder utilizarla es necesario tener instalado *Java Development Kit* (JDK) versión 1.6 o superior.

6.2. Distribución

La distribución del Sistema Fastest está compuesta por los siguientes ítems:

- Archivo `fastest.tar.gz`: en él se encuentran los archivos que conforman la herramienta y que permiten ejecutarla. Estos archivos pueden utilizarse en los sistemas operativos tanto UNIX/Linux como Windows.
- Binarios para Linux y Windows de *Java Development Kit* (JDK) versión 1.6.
- Manual de usuario, el cual es presentado en la siguiente sección de este informe.
- Suite de especificaciones Z. Consta de una serie de especificaciones Z de distinta complejidad, que tienen como finalidad introducir al usuario en la edición de especificaciones Z y en la práctica de la generación de casos de prueba abstractos.
- Documento PDF "Guía de Módulos del Sistema Fastest": en él se describen la funcionalidad y los secretos de todos los módulos de la herramienta. Se encuentra en el archivo `guia-modulos.pdf`.

- Documento PDF "Especificación de Interfaces del Sistema Fastest": en él se presentan las interfaces de los módulos de la guía antes mencionada. Se encuentra en el archivo `interfaces-modulos.pdf`.
- Documento PDF "ISO Standard Z": en él se presenta una descripción del Estándar ISO de Z. Se encuentra en el archivo `isoZ.pdf`.
- Archivo `simbolosZ.pdf`: en él se presenta una tabla con los comandos LaTeX que corresponden a los distintos símbolos que se utilizan en el lenguaje Z, de acuerdo al estándar ISO.
- Documentación *JavaDoc*: da información exhaustiva del código fuente, simple de utilizar y consultar debido a su carácter navegable. En la misma se describen las clases que componen al sistema, junto con sus interfaces. Se encuentra disponible sólo en inglés.

6.3. Manual de usuario

En esta sección se describen los pasos a seguir y los comandos disponibles para el usuario en la utilización del prototipo de Fastest. Si bien Fastest fue concebido como un sistema distribuido, con el procesamiento repartido entre clientes y servidores, también es posible utilizarlo de forma monolítica, ejecutando solamente una instancia de un cliente, en el cual ocurra todo el procesamiento. El manual en principio explicará cómo usarlo de esta manera, que es en la que los clientes de Fastest están configurados por defecto para trabajar. Al final del capítulo se detallará cómo hacer uso de la herramienta de forma distribuida.

6.3.1. Instalar y ejecutar Fastest

La distribución anexa a este informe incluye el archivo `fastest.tar.gz`, el cual contiene todo lo necesario para ejecutar la herramienta. En primer lugar, este deberá descomprimirse en alguna ubicación que el usuario desee. En el directorio `Fastest` que se obtiene al realizar la extracción se puede encontrar un archivo llamado `fastest.jar`. Este archivo está compuesto por todos los archivos de clases Java que conforman la herramienta, y es el que habrá que utilizar para poder correr Fastest, lo cual se logra ejecutando lo siguiente en la consola del sistema operativo utilizado:

```
$> java -jar fastest.jar
```

De no ocurrir ningún problema, debería aparecer en la pantalla el siguiente indicador:

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

con el cursor esperando por la entrada del usuario, en lo que es el intérprete de comandos de Fastest.

6.3.2. Cargar una especificación

Para cargar una especificación en el contexto de la herramienta se dispone del comando `loadspec`. Al utilizarlo, se lo debe seguir con el nombre del archivo LaTeX que contiene la especificación del sistema a testear. Considerando el ejemplo de la lectura de archivos de la Sección 3.5.1, y asumiendo que su especificación está codificada correctamente en LaTeX en el archivo `lectura.tex` del mismo directorio de `fastest.jar`,el comando debería utilizarse como se muestra a continuación:

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

a lo que sigue, como puede observarse, la impresión de un mensaje que indica que la especificación pudo cargarse exitosamente y el cursor en espera por nueva entrada del usuario.

Si la especificación tuviera algún error de sintaxis o de tipo, esto sería informado en este punto, de manera similar a como muestra el siguiente fragmento:

```
Fastest> loadspec lectura.tex
Loading specification..
Specification has not been loaded because it has type errors.

line 12 column 13 in "lectura.tex": Syntax error in variable declaration at
token BYTEpermiso; an expression is expected after token COLON.
line 13 column 1 in "lectura.tex": Possible missing hard space
Fastest>
```

donde se brinda información acerca del posible error en el archivo LaTeX, como una guía para que el usuario pueda corregirlo. Lógicamente, en casos de error como este, la especificación no se carga en el contexto de Fastest. Para intentar hacerlo de nuevo posteriormente, deberá utilizarse otra vez el comando `loadspec`.

6.3.3. Visualizar la especificación cargada

En algunos casos, el usuario puede querer presentar en pantalla la especificación que ya ha cargado en el contexto de Fastest. Para esto, la herramienta provee del comando `showspec`. Si el comando se utiliza sin argumentos, la especificación se imprime en pantalla de manera idéntica a como está codificada en el archivo a partir del cual fue cargada. No se imprime el contenido del archivo exactamente, pues el texto informal que éste puede contener no es tenido en cuenta; pero los fragmentos que corresponden a la especificación sí son presentados con exactitud.

El comando `showspec` puede utilizarse seguido de la opción `-u`, de la siguiente forma:

```
Fastest> showspec -u
```

en cuyo caso la especificación se muestra con todos sus esquemas completamente expandidos.

Además, al comando le puede seguir la opción `-o` junto con un nombre de archivo como argumento, para redirigir la salida al archivo indicado. Dado que la especificación ya está escrita en un archivo, esta opción puede parecer inútil. Sin embargo, es la combinación de las dos opciones `-u` y `-o` la que puede tener sentido considerar: para tener en un archivo una versión expandida de la especificación. Por ejemplo, podría enviarse la especificación al archivo `lectura2.tex`, con sus esquemas expandidos, con el siguiente comando:

```
Fastest> showspec -u -o lectura2.tex
```

Al utilizar esta opción, el contenido del archivo de destino no será idéntico a lo que se presenta en pantalla cuando no se especifica la opción. En este caso, además se incluyen porciones de código LaTeX adicionales, las cuales son necesarias para que el archivo pueda compilarse y generar un archivo en formato PDF.

Otro comando que vale la pena describir en este momento es `showloadedops`. A través de él es posible obtener un listado de las operaciones cargadas con la especificación, lo cual puede dar al usuario de Fastest conocimiento sobre las operaciones sobre las que puede aplicar el proceso de testing. Este comando se utiliza sin argumentos y no tiene opciones, como muestra el fragmento que sigue:

```
Fastest> showloadedops
* LeerOk
* ArchivoVacio
* FinDeArchivoEncontrado
* ArchivoProhibido
* Leer
```

6.3.4. Seleccionar operaciones para testear

Una vez que se ha cargado la especificación al contexto de Fastest y que se conocen cuales son sus operaciones, el siguiente paso es elegir aquellas para las cuales se intentará generar casos de prueba abstractos. Esto debe hacerse a través del comando `selop`, seguido del nombre de la operación que le corresponde. Se pueden seleccionar todas las operaciones que se deseen, entre las que hayan sido cargadas en el contexto de Fastest con la especificación, pero para cada una de ellas es necesario ejecutar el comando.

Para deseleccionar una de las operaciones previamente seleccionadas, Fastest dispone del comando `unselop`, que al utilizárselo debe seguirle el nombre de la operación correspondiente. Por comodidad, también existe el comando `unselallops`, que se utiliza sin argumentos ni opciones y permite quitar la elección de todas las operaciones seleccionadas para aplicar.

Para ver un listado de las operaciones seleccionadas, usar el comando `showselops`.

6.3.5. Agregar tácticas de testing

Además de ser necesario que el usuario de Fastest seleccione las operaciones a testear, éste debe indicar qué tácticas de testing van a intentar aplicarse a cada una de ellas para generar los árboles de prueba. El comando `addtactic` hace posible que se forme la lista de las tácticas que se aplicarán a determinada operación previamente seleccionada. La sintaxis de este comando es bastante compleja porque depende de la táctica que en particular se quiera aplicar. Sin embargo, la sintaxis base es:

```
addtactic <op_name> <tactic_name> <tactic_parameters>
```

donde `op_name` es el nombre de la operación seleccionada, `tactic_name` es el nombre de la táctica (soportada por Fastest) que se quiere aplicar y `tactic_parameters` es el listado de parámetros dependiente de la táctica indicada.

Como se vio en la Sección 3.5.1, las tácticas soportadas por la actual versión de Fastest son:

- Forma Normal Disyuntiva (ver Sección 3.5.1.2). Se aplica por defecto a toda operación y no es necesario agregarla.
- Particiones Estándar (ver Sección 3.5.1.3). Si se quiere agregar esta táctica, debe hacerse con el nombre `SP`. Los parámetros para `addtactic <op_name> SP` son `<operator> <expression>`, donde `operator` es un operador Z escrito en LaTeX, y `expression` es una expresión Z en LaTeX que contiene al operador y que está contenida en el predicado de la operación a la que se le quiere agregar la táctica. Por lo tanto, `SP` se puede aplicar a distintos operadores y distintas expresiones de la misma operación, siempre y cuando aparezcan en el predicado de la misma. Por otro lado, Fastest debe tener definida una partición estándar para el operador que el usuario especifique. Más adelante se verá cómo agregar definiciones de particiones estándar a las que ya posee la herramienta.
- Tipos Libres (ver Sección 3.5.1.4). Si se quiere agregar esta táctica, utilizar el nombre `FT`. Esta necesita sólo un parámetro: el nombre de la variable cuyo tipo es un tipo libre. La versión de Fastest aquí presentada sólo soporta la aplicación de FT considerando tipos libres que sean enumerados, es decir, que no definan un tipo inductivo.

Algunos ejemplos de uso del comando `addtactic` son:

1. `addtactic Leer SP \leq longitud? \leq \# datos`
2. `addtactic Leer SP \dres (1..longitud?) \dres datos`
3. `addtactic Leer FT permiso?`

los cuales, respectivamente, tienen los siguientes significados:

1. Se quiere agregar la táctica de Particiones Estándar al listado de tácticas de la operación *Leer*, considerando el operador $\leq$ de la expresión $longitud? \leq \# datos$, la cual está contenida en el predicado de *Leer*.

2. Se quiere agregar la táctica de Particiones Estándar al listado de tácticas de la operación *Leer*, considerando el operador $\lhd$ de la expresión $(1..longitud?) \lhd datos$, la cual está contenida en el predicado de *Leer*.

3. Se quiere agregar la táctica de Tipos Libres al listado de la operación *Leer*, considerando la variable *permiso*?, que está declarada en *Leer* como una variable cuyo tipo es el tipo libre *PERMISO*.

6.3.5.1. Configurar particiones estándar

Como se pudo observar, una de las tácticas soportadas por Fastest es Particiones Estándar. En la Sección 3.5.1.3 se mostró que al querer aplicar esta táctica a una clase de prueba de una operación *Op*, se debe considerar un operador que tenga alguna ocurrencia en el predicado de *Op* y para el que esté definida una partición estándar de su dominio. Con el objetivo de que el usuario pueda consultar las definiciones de las particiones estándar, editarlas o agregar otras nuevas fácilmente, Fastest las mantiene en un único archivo de configuración, escritas en un sencillo lenguaje especialmente diseñado para tal fin. El nombre y ubicación del archivo, respecto al directorio donde se encuentra `fastest.jar`, es `lib/conf/stdpartition.spf`, si el sistema operativo utilizado es UNIX/Linux o `lib\conf\stdpartition.spf`, si se trabaja bajo Windows. En caso de modificar el archivo, y de querer apreciar los cambios realizados al usar la herramienta, ésta debe ser reiniciada.

Cada partición estándar debe estar dada por exactamente una definición en el archivo mencionado. Cada línea de la definición debe terminar con un símbolo `;`(punto y coma). La primera de estas líneas debe tener el operador Z que corresponde a la partición, escrito en formato LaTeX, seguido por la cadena `: operator`, y finalizando con la lista de parámetros formales del operador, separados por comas y encerrando a todos juntos entre paréntesis. La última línea de la definición simplemente debe ser `end operator;` y cada línea intermedia debe definir un sub-dominio de la partición estándar. Cada sub-dominio es una conjunción de condiciones, las cuales deben separarse por comas. A su vez, cada condición debe ser una expresión escrita en LaTeX, en función de los parámetros formales listados en la primer línea de la definición. Por ejemplo, la partición estándar que suele utilizarse para el operador $<$, se define de la siguiente manera:

```
< : operator(A,B);
A < 0, B < 0;
A < 0, B = 0;
A < 0, B > 0;
A = 0, B < 0;
A = 0, B = 0;
A = 0, B > 0;
A > 0, B < 0;
A > 0, B = 0;
A > 0, B > 0;
end operator;
```

6.3.6. Generar el árbol de pruebas

Cuando el usuario de Fastest ha elegido las operaciones con las cuales trabajar y las tácticas de testing que se aplicarán a cada una de ellas, todo está preparado para iniciar la generación de árboles de pruebas para tales operaciones. El comando que debe utilizarse para ordenar este proceso se llama `genalltt`, cuyo nombre proviene de "*generate all test trees*" ("generar todos los árboles de prueba"). `genalltt` debe utilizarse sin argumentos y sin opciones, y de realizar su tarea exitosamente, Fastest debe presentar al usuario, nuevamente, el cursor en espera del ingreso de un nuevo comando.

6.3.7. Generar casos de prueba abstractos

La generación de casos de prueba se ordena a través del comando `genalltca`, sin utilizar argumentos ni opciones. El nombre del comando proviene de "*generate all test cases*" ("generar todos los casos de prueba"). Al ejecutar `genalltca`, Fastest intentará generar casos de prueba abstractos para todas las operaciones seleccionadas, distribuyendo el procesamiento entre todos los servidores registrados. Por cada operación seleccionada, se verificará si ésta tiene un árbol de pruebas previamente generado. En caso afirmativo, Fastest intentará obtener casos de prueba abstractos para las clases de prueba que sean hojas del árbol. Si a la operación no le corresponde ningún árbol de pruebas, la herramienta primero generará uno (teniendo en cuenta el listado de tácticas asociado a la operación) y después intentará encontrar casos de prueba a partir de él.

Al ejecutar el comando, se irán mostrando en pantalla los nombres de las clases de prueba para las que se realiza la búsqueda de casos de prueba, indicando para cada una el éxito o fracaso de la generación.

6.3.8. Presentar los resultados

Para presentar los resultados obtenidos en el proceso de testing, Fastest provee los comandos `showtt` y `showsch`. Sus nombres provienen de "*show test trees*" ("mostrar árboles de prueba") y "*show schema*' ("mostrar esquema"), respectivamente. `showtt` hace posible la presentación de todos los árboles de pruebas que hayan sido generados y no espera argumentos. Si ya se generaron casos de prueba, `showtt` además mostrará estos casos "colgando" de las clases que les corresponden. Por su parte, `showsch` muestra el esquema cuyo nombre se pasa como argumento, el cual puede ser un esquema que aparece en la especificación original, o bien una clase de prueba o un caso de prueba que hayan sido generados por la herramienta.

Los dos comandos cuentan con la opción `-o`, la cual como en 6.3.3, redirige la salida hacia el archivo cuyo nombre debe indicarse como argumento de la opción. El contenido del archivo incluirá código LaTeX adicional, necesario para que éste pueda compilarse y generar un archivo en PDF.

Además, el comando `showsch` permite utilizar la opción `-u`. Esta espera un número entero como argumento, `orden_de_expansion`. Si el esquema a presentar está en la especificación original y `orden_de_expansion` es cero, el esquema se muestra tal cual fue definido en esa especificación; si `orden_de_expansion` es distinto de cero, se muestra completamente expandido. Si el esquema a presentar es una clase de prueba o un caso de prueba, casos en los que será un nodo de algún árbol

de pruebas, el esquema se muestra expandiendo los esquemas incluídos tantas veces como indique `orden_de_expansion`, salvo que este valor sea negativo o mayor que la distancia del nodo a la raíz del árbol, situación en la que el esquema se muestra expandido completamente.

`showsch` puede usarse de una forma alternativa, sin indicar el nombre de un esquema a continuación del comando. Si el usuario ejecuta `showsch -tcl`, se muestran todos los esquemas correspondientes a clases de prueba. Por otro lado, al ejecutar `showsch -tca`, se presentan todos los esquemas que corresponden a casos de prueba. En estas dos variantes se puede utilizar la opción `-p` seguida por el nombre de una operación, en cuyo caso se presentan las clases de prueba o los casos de prueba (según sea `-tcl` o `-tca`, respectivamente) pero sólo para la operación indicada.

6.3.9. Otros comandos

Los siguientes son otros comandos ofrecidos por la herramienta, cuyos significados son más triviales, lo que no implica que sean menos necesarios en la utilización de Fastest. Ninguno de ellos espera argumentos ni acepta opciones.

`version`

Presenta en pantalla la versión actual de Fastest.

`help`

Presenta en pantalla la ayuda de Fastest, con un listado de los comandos disponibles. La ayuda se presenta en inglés, el idioma utilizado por la herramienta para interactuar con el usuario.

`reset`

Elimina toda la información del contexto de Fastest. Esto incluye, alguna especificación que se haya cargado, operaciones seleccionadas, tácticas de testing agregadas a una aplicación, árboles de pruebas y casos de prueba calculados, etc.

`quit / exit`

Cierra la herramienta.

6.3.10. Ejecutando Fastest como un sistema distribuido

Además de `fastest.jar` hay otros archivos que se extraen del archivo `fastest.tar.bz` hacia el directorio de nombre `Fastest`, entre los que se encuentra `fastest-server.jar`. Es a partir de este archivo que el usuario de la herramienta puede poner en funcionamiento un servidor para la generación de casos de prueba abstractos. El primer paso es realizar la misma extracción de `fastest.tar.bz` que se mostró en 6.3.1 para utilizar el cliente de Fastest, pero en el equipo donde quiere que funcione el servidor. A continuación, debe realizarse la ejecución en una consola, dentro del directorio `Fastest`, de la siguiente línea de comando:

```
$> java -jar fastest-server.jar
```

la cual inicia el correspondiente servidor y hará que éste espere por conexiones con uno o más clientes de Fastest. Éstas conexiones se establecerán, del lado del servidor, sobre un puerto que se especifica en el archivo `lib/conf/server-port.conf`, cuya ubicación es relativa al directorio donde se encuentra `fastest.jar`. Una vez iniciado, el servidor queda corriendo indefinidamente, lo cual se pretende mejorar en futuras versiones de Fastest.

Por el lado de los clientes, es necesario registrar en ellos los servidores a los cuales pueden conectarse para solicitar algún servicio de procesamiento. Para este fin, debe editarse el archivo `lib/conf/cserversinfo.conf`. Este archivo de configuración contiene un listado de los servidores asociados al cliente, indicando uno por línea y especificando el nombre del servidor, su dirección IP y el puerto de conexión, en ese orden. Un ejemplo de registro de servidores en un cliente de Fastest es el siguiente:

```
fastest1 127.0.0.1 5001
fastest2 192.168.131.1 4505
fastest3 192.168.131.2 7171
fastest4 192.168.131.3 6820
```

donde puede apreciarse que el cliente que utiliza esa configuración interactuará con 4 servidores de cómputo, teniendo al primero de ellos (el que tiene la dirección de loopback 127.0.0.1) en ejecución en la misma computadora. En verdad, en la versión actual de Fastest, si se indica un servidor con la dirección de loopback, todo el procesamiento ocurre en el mismo cliente, por lo que no es necesario poner a funcionar un servidor a través de `fastest-server.jar`. También en esta versión, el archivo de configuración `lib/conf/cserversinfo.conf`, por defecto, contiene una sóla línea y es como la primera del ejemplo. De esta manera, si el archivo no es modificado, todo el procesamiento de Fastest ocurre en el mismo cliente. En versiones futuras, puede considerarse modificar esta característica para que sea posible tener un cliente y un servidor funcionando en la misma computadora, sobretodo en situaciones donde el equipo tenga posibilidades de funcionar con más de un microprocesador.

Capítulo 7

Caso de Estudio: Protocolo de Comunicación

En este capítulo se presentará un caso de estudio en el cual se describirá el uso de Fastest sobre la especificación Z de un sistema particular, con la intención de mostrar cómo pueden generarse casos de prueba abstractos de manera automática. La especificación corresponde al protocolo de comunicación *EXP-OBDH Communication Protocol* (EOCP) para el microsatélite científico SACI-1.

7.1. La especificación Z

Esta sección describe el modelo formal del EXP-OBDH Communication Protocol (EOCP), el cual fue escrito a partir de su especificación informal [36]. A la hora de realizar la lectura del modelo, es importante notar lo siguiente:

1. De todos los componentes del protocolo, sólo uno, cuyo nombre es SLAVE, ha sido especificado. Esto se debe a que SLAVE es probablemente el módulo más complejo del sistema.

2. Dado que es difícil, sino imposible, especificar requisitos de tiempo en el lenguaje Z, y dado que la especificación informal de EOCP incluye tales requisitos, el modelo formal aquí presentado no es lo suficientemente preciso como debería ser. Sin embargo, en este caso, los requisitos de tiempo son muy simples.

3. Las prioridades son también difíciles de especificar en Z y EOCP requiere que el componente EXP asigne más prioridad para recibir interrupciones para adquirir datos que para reaccionar a comandos enviados por el componente OBDH. No se ha formalizado este requerimiento.

4. Dado que Fastest todavía no implementa completamente el lenguaje Z, la especificación ha sido forzada para utilizar sólo el subconjunto del lenguaje implementado en la herramienta. Esto puede conducir a una especificación innecesariamente compleja o ilegible.

5. El protocolo formalizado es principalmente un sistema reactivo, por lo que Z no es el lenguaje de especificación formal más apropiado.

6. Se asumirá que todas las operaciones son atómicas.

7. No se ha modelado la verificación de la integridad de un mensaje.

7.1.1. Tipos básicos y tipos libres

EXP captura información de su entorno, la cual, posiblemente, deba ser enviada hacia OBDH. Además, EXP almacena su propio estado en memoria. El siguiente tipo básico representa ambas clases de información:

$$[MDATA]$$

OBDH puede enviar distintos comandos a EXP para que éste realice determinada acción. Estos comandos se representan a través del tipo enumerado que sigue:

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

Los diferentes tipos de respuestas que envía EXP también se representan con un tipo enumerado.

$$RTYPE ::= MR \mid CD \mid EDP \mid ND \mid MDD$$

Los modos de configuración en los cuales EXP puede trabajar son representados como otro tipo enumerado.

$$CMODE ::= COF \mid CON$$

OBDH puede solicitarle a EXP actuar sobre sus diferentes dispositivos a través de un comando llamado Cargar Parámetros con un valor particular en el campo DATA. El tipo $PARAM$ se refiere a esos valores.

$$PARAM ::= LLDHiOff \mid LLDHiOn \mid TMOff \mid TMOn \mid HiV50 \mid HiV45 \mid HiV40 \mid HiVOff \mid RU \mid TS18 \mid TS13$$

En este modelo el tiempo se considera discreto. Teniendo en cuenta que el menor requisito de tiempo está dado en milisegundos, se asumirá que cada unidad de tiempo representa un milisegundo.

$$TIME == \mathbb{N}$$

BIN representa un tipo binario[1].

$$BIN ::= no \mid yes$$

[1]El lenguaje Z no trae definido un tipo para la representacion de tipo binario.

7.1.2. El espacio de estado del Sistema, sus estados iniciales y sus invariantes de estado

Las variables de estado del modelo se listan a continuación:

ctime ≈ El tiempo actual en EXP

memd ≈ La memoria en que EXP almacena registros de los experimentos realizados.

memp ≈ La memoria en que EXP almacena la imagen del programa y sus variables locales.

Se asume que EXP tiene espacio para 6 buffers de 43 elementos de tipo *MDATA*. Los primeros 5 buffers se reservan para el código del programa, sus variables, su pila, etc (*memp*), y el buffer restante es para información vinculada a experimentación (*memd*).

mdp ≈ Puntero de envío de memoria.

Es utilizado para mantener una indicación de cuántas celdas de memoria han sido enviadas al OBDH en respuesta a un comando de envío de memoria.

ped ≈ Puntero de memoria de datos de experimentación.

Es usada para mantener una indicación de cuántas celdas de memoria, para registros de experimentación, han sido enviadas al OBDH en respuesta a un comando de transmisión de datos.

mep ≈ Puntero que indica la última celda usada en la memoria para registros de experimentación.

Es usado para mantener una indicación de cuántas celdas de memoria están siendo usadas para almacenar registros de experimentación.

acquiring ≈ Si EXP está en modo de adquisición o no.

waiting ≈ Si EXP está esperando por el resto de un comando OBDH.

mode ≈ Modo de configuración en el que EXP está trabajando actualmente.

ccmd ≈ El próximo comando a ser procesado por EXP.

sending ≈ Si EXP está enviando datos a OBDH o no.

dumping ≈ Si EXP está enviando su memoria o no.

waitsignal ≈ Si EXP está esperando para enviar una señal al telescopio, que le indique que debe adquirir nuevos registros de experimentación.

Se han agrupado éstas variables de estado en tres esquemas Z: el primero, agrupa variables relacionadas a la memoria; el segundo, variables vinculadas al tiempo; y el tercero, todas las variables restantes.

Memory

$memp, memd : \text{seq}\, MDATA$
$mdp, mep, ped : \mathbb{N}$

$Time == [ctime : TIME]$

Status

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

El esquema de estado es la inclusión de los tres esquemas previos.

$ExpState == [Memory;\ Time;\ Status]$

El esquema de estado inicial, el cual representa un conjunto infinito de estados, establece valores razonables para algunas de las variables de estado.

ExpInitState

$ExpState$

$\#memp = 5 * 43$
$\#memd = 43$
$mdp = mep = ped = 0$
$acquiring = waiting = sending = dumping = waitsignal = no$

Los invariantes de estado del sistema se describen en el siguiente esquema:

StateInvariants

$ExpState$

$\#memp = 5 * 43$
$\#memd = 43$
$mep \in 0 \,..\, 43$
$ped \in 0 \,..\, 43$
$mdp \in 0 \,..\, 5 * 43$
$acquiring = no \Rightarrow waitsignal = no$

7.1.3. Operaciones

Por simplicidad se asumirá que, desde un punto de vista abstracto, los comandos se envían desde OBDH a EXP en cuatro etapas (sin importar cuántos paquetes de datos de bajo nivel deban ser enviados):

1. OBDH inicia un comando.
2. OBDH envía el tipo de comando.
3. OBDH envía los parámetros (de ser necesario) de acuerdo al tipo de comando.
4. OBDH termina el comando.

EXP sincronizará con OBDH de la siguiente manera:

1. EXPR esperará a OBDH para iniciar un comando.
2. EXP esperará por un tipo de comando.
3. EXP esperará por el fin del comando.
4. EXP ejecutará una entre varias operaciones internas, dependiendo del tipo de comando recibido en el segundo paso.

   Notar que no se está modelando la recepción de parámetros: simplemente se asume que esto ocurre de algún modo.

Las operaciones correspondientes a los primeros tres pasos se describen en las Sección 7.1.3.2 y las operaciones internas (cuarto paso) son descriptas en las secciones 7.1.3.4 y 7.1.3.5. Sin embargo, la siguiente sección (7.1.3.1) muestra algunas operaciones relacionadas a aspectos temporales, dado que algunas de las otras operaciones tienen requisitos de tiempo.

EXP también tiene que sincronizar con los sensores del satélite para controlar la adquisición de datos. Esto se muestra en la Sección 7.1.3.3.

Debe notarse que todas las operaciones, salvo cuando se diga explícitamente lo contrario, constituyen la interfaz que EXP expone a ODBH y otros componentes tales como los sensores y el reloj.

7.1.3.1. Operaciones relacionadas a aspectos temporales

Tick es una operación externa llevada a cabo por el hardware de EXP y representa el avance del reloj en una unidad de tiempo. De alguna forma, EXP recibe este nuevo valor, o el hardware modifica una de las celdas de memoria de EXP. Esto es irrelevante en este nivel de abstracción.

$$Tick == [\Delta ExpState;\ \Xi Memory;\ \Xi Status \mid ctime' = ctime + 1]$$

Cuando EXP recibe de OBDH el comienzo de un nuevo comando, se establece un timer que expirará despues de 500 ms. $Timeout500ms$ representa esa expiración (es decir, el timer (externo) llamará a $Timeout500ms$ cuando termine). Después de la ejecución de esta operación EXP no esperará más por el resto del mensaje. $Set500msTimer$ y $Stop500Timer$ constituyen la interfaz que EXP debe usar par establecer y detener el timer, y por ahora se las deja sub-especificadas.

$Timeout500ms$

$\Delta ExpState;\ \Xi Memory;\ \Xi Time$

$waiting' = no$
$acquiring' = acquiring$
$sending' = sending$
$dumping' = dumping$
$waitsignal' = waitsignal$
$mode' = mode$
$ccmd' = ccmd$

$Set500msTimer$

$x : \mathbb{Z}$

$Stop500msTimer$

$x : \mathbb{Z}$

El otro requisito de tiempo para EXP tiene que ver con la recolección de registros experimentales. EXP debe emitir una interrupción hacia el hardware del telescopio cada 700 ms. Se procederá de forma análoga a lo anterior:

$Timeout700ms$

$\Delta ExpState;\ \Xi Memory;\ \Xi Time$

$waitsignal' = no$
$sending' = sending$
$waiting' = waiting$
$acquiring' = acquiring$
$dumping' = dumping$
$mode' = mode$
$ccmd' = ccmd$

$Set700msTimer$

$x : \mathbb{Z}$

$Stop700msTimer$

$x : \mathbb{Z}$

7.1.3.2. Inicio de comando, tipo de comando y fin de comando

El documento de requerimientos dice que EXP puede detectar el comienzo de un nuevo comando, momento a partir del cual debe esperar 500 ms por el resto del comando. Si este no llega, EXP esperará por otro nuevo comando.

Las siguientes operaciones son ofrecidas por EXP para que OBDH puede ejecutarlas para transmitir sus comandos.

CommandStartOk

$\Delta ExpState;\ \Xi Memory;\ \Xi Time$

$waiting = no$
$waiting' = yes$
$acquiring' = acquiring$
$sending' = sending$
$dumping' = dumping$
$waitsignal' = waitsignal$
$mode' = mode$
$ccmd' = ccmd$

$CommandStartE == [\Xi ExpState \mid waiting = yes]$

$CommandStart == (CommandStartOk \land Set500msTimer) \lor CommandStartE$

Cuando EXP está esperando, OBDH puede establecer el tipo del comando que quiere ejecutar en EXP. Aquí se tienen tres porque hay dos comandos que envían sus respuestas en más de un paso. Estos comandos son: transmitir datos (TD) y enviar memoria (MD).

CommandTypeOk1

$\Delta ExpState;\ \Xi Memory;\ \Xi Time$
$type? : CTYPE$

$waiting = yes$
$type? \notin \{TD, MD\}$
$ccmd' = type?$
$waiting' = waiting$
$acquiring' = acquiring$
$sending' = sending$
$dumping' = dumping$
$waitsignal' = waitsignal$
$mode' = mode$

CommandTypeOk3

$\Delta ExpState;\ \Xi Memory;\ \Xi Time$
$type? : CTYPE$

$waiting = yes$
$type? = MD$
$ccmd' = type?$
$dumping' = yes$
$sending' = sending$
$acquiring' = acquiring$
$waiting' = waiting$
$waitsignal' = waitsignal$
$mode' = mode$

CommandTypeOk2

$\Delta ExpState;\ \Xi Memory;\ \Xi Time$
$type? : CTYPE$

$waiting = yes$
$type? = TD$
$ccmd' = type?$
$sending' = yes$
$acquiring' = acquiring$
$waiting' = waiting$
$dumping' = dumping$
$waitsignal' = waitsignal$
$mode' = mode$

$$CommandTypeE == [\Xi ExpState \mid waiting \neq yes]$$

$$CommandType == CommandTypeOk1 \lor CommandTypeOk2 \lor CommandTypeOk3 \lor CommandTypeE$$

De alguna forma se asume que OBDH comunica a EXP que el comando actual ha finalizado, pero EXP prestará atención a esta señal si está esperando por ella.

*CommandFinishOk*1

$\Delta ExpState;\ \Xi Memory;\ \Xi Time$

$waiting = yes$
$waiting' = no$
$sending' = sending$
$acquiring' = acquiring$
$dumping' = dumping$
$waitsignal' = waitsignal$
$mode' = mode$
$ccmd' = ccmd$

$$CommandFinishE == [\Xi ExpState \mid waiting \neq yes]$$

$$CommandFinish == (CommandFinishOk1 \land Stop500msTimer) \lor CommandFinishE$$

Notar que tanto *CommandFinishOk* como *Timeout*500*ms* pueden cambiar el valor de *waiting*. Precisamente, el comportamiento de EXP con respecto a si el resto de un nuevo comando es aceptado, depende de cuáles de esas operaciones se ejecuta primero.

Una vez que OBDH completa el comando, EXP puede ejecutar el cuerpo de la correspondiente operación interna (mostrado en secciones 7.1.3.4 y 7.1.3.5).

7.1.3.3. Adquisición de datos

Una vez que OBDH le ordena a EXP la habilitación de la interrupción para adquisición de datos, EXP enviará la interrupción al telescopio cada 700 ms. Después que la interrupción fue enviada, EXP es interrumpido por el telescopio cuando haya información para ser consultada. Esta información es almacenada en un buffer de 43 celdas. *RetrieveEData* es, entonces, la operación de EXP ejecutada por la interrupción enviada por el telescopio.

RetrieveEDataOk

$\Delta ExpState;\ \Xi Time;\ \Xi Status$

$data? : \operatorname{seq} MDATA$

$data? \neq \langle\rangle$

$mep + \#data? \leq 43$

$memd' = memd \oplus \{i : 1 \mathinner{..} \#data? \bullet mep + i \mapsto data?\ i\}$

$mep' = mep + \#data? + 1$

$ped' = 0$

$mdp' = mdp$

$memp' = memp$

$RetrieveEDataE == [\Xi ExpState;\ data? : \operatorname{seq} MDATA \mid data? = \langle\rangle \lor 43 < mep + \#data?]$

$RetrieveEData == RetrieveEDataOk \lor RetrieveEDataE$

No se ha incluido $waiting = no \land acquiring = yes$ en la precondición porque $acquiring = no$ significa que al telescopio no se le solicita realizar una adquisición de datos, y esto implica que el telescopio no enviará la interrupción a EXP.

7.1.3.4. Comandos OBDH simples

En esta sección se mostrará la especificación de aquellos comandos que necesitan respuestas de un paso para finalizar.

El comando para resetear el microcontrolador no cambia el estado de EXP porque el microcontrolador es considerado un componente externo. Entonces, se puede (sub) especificar una interfaz que esta operación puede llamar para reiniciar el microcontrolador. Posiblemente, esta operación externa puede reiniciar, por ejemplo, la memoria del sistema, pero esto no se encuentra el el documento de requerimientos.

MicroReset

$x : \mathbb{Z}$

ResetMicroOk

$\Xi ExpState$

$rsp! : RTYPE$

$waiting = no$

$ccmd = RM$

$rsp! = MR$

$ResetMicroE == [\Xi ExpState \mid waiting \neq no \lor ccmd \neq RM]$

$ResetMicro == (ResetMicroOk \land MicroReset) \lor ResetMicroE$

El comando que consulta el tiempo actual envía el valor actual de *ctime*.

SendClockOk

$\Xi ExpState$
$rsp! : RTYPE$
$rdata! : TIME$

$waiting = no$
$ccmd = SC$
$rsp! = CD$
$rdata! = ctime$

$SendClockE == [\Xi ExpState \mid waiting \neq no \lor ccmd \neq SC]$

$SendClock == SendClockOk \lor SendClockE$

La adquisición de datos se lleva a cabo en dos pasos: primero, OBDH envía a EXP el comando apropiado, y segundo, una operación interna de EXP se ejecuta cada 700 ms. Esta operación interna envía una interrupción al telescopio para ordenar la adquisición de datos. El primer paso se formaliza en el esquema *InitDataAcq* y el segundo en el esquema *InterruptTele*.

InitDataAcqOk

$\Delta ExpState$; $\Xi Memory$; $\Xi Time$
$rsp! : RTYPE$

$waiting = no$
$ccmd = IDA$
$acquiring = no$
$acquiring' = waitsignal' = yes$
$rsp! = MR$
$waiting' = waiting$
$sending' = sending$
$dumping' = dumping$
$mode' = mode$
$ccmd' = ccmd$

$InitDataAcqE == [\Xi ExpState \mid waiting \neq no \lor ccmd \neq IDA \lor acquiring \neq no]$

$InitDataAcq == (InitDataAcqOk \land Set700msTimer) \lor InitDataAcqE$

Cuando el timer de 700 ms expira, la operación interna *InterruptTele* es habilitada. *InterruptTele* se deshabilita a sí misma después de ejecutarse, pero establece nuevamente el timer de 700 ms para ser iniciada hasta que OBDH le indique a EXP que no debe adquirir más datos.

InterruptTeleOk

$\Delta ExpState;\ \Xi Memory;\ \Xi Time$

$signal! : \mathbb{N}$

$waiting = no$
$acquiring = yes$
$waitsignal = no$
$waitsignal' = yes$
$signal! = 1$
$waiting' = waiting$
$sending' = sending$
$dumping' = dumping$
$acquiring' = acquiring$
$mode' = mode$
$ccmd' = ccmd$

$InterruptTele == InterruptTeleOk \land Set700msTimer$

Cuando OBDH le solicita a EXP detener la adquisición de datos, simplemente cambia el valor de la variable *acquiring* de *yes* a *no*. Notar que esto implica que *InterruptTele* no será habilitada otra vez.

StopDataAcqOk

$\Delta ExpState;\ \Xi Memory;\ \Xi Time$

$rsp! : RTYPE$

$waiting = no$
$ccmd = SDA$
$acquiring = yes$
$acquiring' = waitsignal' = no$
$rsp! = MR$
$waiting' = waiting$
$sending' = sending$
$dumping' = dumping$
$mode' = mode$
$ccmd' = ccmd$

$StopDataAcqE == [\Xi ExpState \mid waiting \neq no \lor ccmd \neq SDA \lor acquiring \neq yes]$

$StopDataAcq == StopDataAcqOk \lor StopDataAcqE$

La reconfiguración es fácil de modelar porque sólo se tiene que cambiar el valor de la variable *mode*, como sigue:

ReconfigOk

$\Delta ExpState;\ \Xi Memory;\ \Xi Time$
$nmode? : CMODE$
$rsp! : RTYPE$

$waiting = no$
$ccmd = RC$
$mode' = nmode?$
$rsp! = MR$
$acquiring' = acquiring$
$sending' = sending$
$waiting' = waiting$
$dumping' = dumping$
$waitsignal' = waitsignal$
$ccmd' = ccmd$

$ReconfigE == [\Xi ExpState \mid waiting \neq no \lor ccmd \neq RC]$

$Reconfig == ReconfigOk \lor ReconfigE$

Cuando EXP recibe el comando de envío de memoria, se prepara para enviarla y devuelve el mensaje RECEIVED a OBDH. El envío real se produce cuando OBDH envía un comando de transmisión de datos.

MemoryDumpOk

$\Xi ExpState$
$rsp! : RTYPE$

$waiting = no$
$ccmd = MD$
$rsp! = MR$

$MemoryDumpE == [\Xi ExpState \mid waiting \neq no \lor ccmd \neq MD]$

$MemoryDump == MemoryDumpOk \lor MemoryDumpE$

Cargar la memoria con datos provenientes de OBDH es más complejo que las operaciones previas, porque es necesario formalizar cómo los 43 bytes dedicados a registros de experimentación pueden ser modificando evitando desbordamientos, dado que ellos pueden dar lugar a errores fatales de programa.

*MemoryLoadOk*1

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

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

*MemoryLoadOk*2

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

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

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

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

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

Cuando un comando de carga de parámetros es recibido, EXP simplemente chequea si el parámetro es posible, y si lo es, EXP lo envía al dispositivo apropiado. La validación de parámetros es modelada por la espera de un valor de tipo $PARAM$. El envío del valor al hardware se representa con una variable de salida.

LoadParamOk

$\Xi ExpState$
$p?, actuate! : PARAM$
$rsp! : RTYPE$

$waiting = no$
$ccmd = LP$
$actuate! = p?$
$rsp! = MR$

$LoadParamE == [\Xi ExpState \mid waiting \neq no \land ccmd \neq LP]$

$LoadParam == LoadParamOk \lor LoadParamE$

7.1.3.5. Transmisión de datos

La transmisión de datos es la operación más compleja del sistema porque involucra el envío de diferentes partes de la memoria, afectándola a ella misma de diversas maneras. Más aún, la transmisión de datos posiblemente necesite más de una respuesta para finalizar.

La operación es invocada para transmitir registros de experimentación y el resultado es un envío de memoria. Los primeros dos esquemas que se presentan a continuación describen la comunicación de datos de experimentación y los últimos dos la transmisión después de un envío de memoria.

Una vez que OBDH solicita registros de experimentación, EXP envía un paquete de 43 bytes siempre que el buffer esté completo. Si el buffer no está completo, EXP responderá con un mensaje NO DATA.

*TransDataOk*1

$\Delta ExpState;\ \Xi Time$
$rsp! : RTYPE$
$rdata! : \text{seq}\, MDATA$

$waiting = no$
$ccmd = TD$
$sending = yes$
$dumping = no$
$ped = 0$
$mep = 43$
$rsp! = EDP$
$rdata! = memd$
$ped' = 43$
$sending' = no$
$acquiring' = acquiring$
$waiting' = waiting$
$dumping' = dumping$
$waitsignal' = waitsignal$
$ccmd' = ccmd$
$mode' = mode$
$memp' = memp$
$memd' = memd$
$mdp' = mdp$
$mep' = mep$

*TransDataE*1

$\Xi ExpState$
$rsp! : RTYPE$

$waiting = no$
$ccmd = TD$
$sending = yes$
$dumping = no$
$mep < 43 \lor ped = 43$
$rsp! = ND$

Enviar la memoria es similar a transmitir registros de experimentación, pero:

- Se ha interpretado que un envío de memoria no libera la memoria.
- EXP envía un paquete de 43 bytes como respuesta a cada comando de transmisión de datos enviado por OBDH, hasta que todos los *memp* han sido transmitidos.

TransDataOk2

$\Delta ExpState;\ \Xi Time;\ \Xi Status$
$rsp! : RTYPE$
$rdata! : \text{seq}\ MDATA$

$waiting = no$
$ccmd = TD$
$sending = dumping = yes$
$mdp < 5 * 43$
$rsp! = MDD$
$rdata! = (mdp + 1\ ..\ mdp + 43) \lhd memp$
$mdp' = mdp + 43$
$memp' = memp$
$memd' = memd$
$mep' = mep$
$ped' = ped$

TransDataOk3

$\Delta ExpState;\ \Xi Time$
$rsp! : RTYPE$

$waiting = no$
$ccmd = TD$
$sending = dumping = yes$
$mdp = 5 * 43$
$rsp! = ND$
$mdp' = 0$
$sending' = dumping' = no$
$memp' = memp$
$memd' = memd$
$mep' = mep$
$ped' = ped$
$acquiring' = acquiring$
$waiting' = waiting$
$sending' = sending$
$waitsignal' = waitsignal$
$mode' = mode$
$ccmd' = ccmd$

$TransDataE2 == [\Xi ExpState \mid waiting \neq no \lor ccmd \neq TD \lor sending \neq yes]$

$TransData ==$
$\quad TransDataOk1$
$\quad \lor\ TransDataOk2$
$\quad \lor\ TransDataOk3$
$\quad \lor\ TransDataE1$
$\quad \lor\ TransDataE2$

7.2. Resultados de testing basado en especificaciones formales Z

Se ha utilizado Fastest para generar casos de prueba abstractos para las operaciones definidas en la especificación Z recién mostrada. Se considera conveniente sólo presentar los resultados que surgen de testear la operación *MemoryLoad*, por cuestiones de espacio y por ser suficiente para evaluar la utilidad de la herramienta.

7.2.1. Poniendo a prueba la operación *MemoryLoad*

La siguiente sucesión de comandos (script) fue utilizada para generar el árbol de pruebas de *MemoryLoad*, mostrarlo en pantalla, y para enviar al archivo `memoryload-clases.tex` las clases de prueba obtenidas:

```
Fastest> loadspec model-test-protocol.tex
Fastest> selop MemoryLoad
Fastest> addtactic MemoryLoad SP \oplus memd \oplus \{i:1 \upto \# data? @
i + addr? - 1 \mapsto data?~i\}
Fastest> genalltt
Generating test tree for 'MemoryLoad' operation..
Fastest> showtt

MemoryLoad_VIS
  !______MemoryLoad_DNF_1
  |       !______MemoryLoad_SP_1
  |       !______MemoryLoad_SP_2
  |       !______MemoryLoad_SP_3
  |       !______MemoryLoad_SP_4
  |       !______MemoryLoad_SP_5
  |       !______MemoryLoad_SP_6
  |       !______MemoryLoad_SP_7
  |       !______MemoryLoad_SP_8
  |
  !______MemoryLoad_DNF_2
  |       !______MemoryLoad_SP_9
  |       !______MemoryLoad_SP_10
  |       !______MemoryLoad_SP_11
  |       !______MemoryLoad_SP_12
  |       !______MemoryLoad_SP_13
  |       !______MemoryLoad_SP_14
  |       !______MemoryLoad_SP_15
  |       !______MemoryLoad_SP_16
  |
  !______MemoryLoad_DNF_3
  |       !______MemoryLoad_SP_17
  |       !______MemoryLoad_SP_18
  |       !______MemoryLoad_SP_19
  |       !______MemoryLoad_SP_20
  |       !______MemoryLoad_SP_21
  |       !______MemoryLoad_SP_22
  |       !______MemoryLoad_SP_23
  |       !______MemoryLoad_SP_24
  |
  !______MemoryLoad_DNF_4
  |       !______MemoryLoad_SP_25
  |       !______MemoryLoad_SP_26
  |       !______MemoryLoad_SP_27
  |       !______MemoryLoad_SP_28
```

```
|       !______MemoryLoad_SP_29
|       !______MemoryLoad_SP_30
|       !______MemoryLoad_SP_31
|       !______MemoryLoad_SP_32
|
!______MemoryLoad_DNF_5
|       !______MemoryLoad_SP_33
|       !______MemoryLoad_SP_34
|       !______MemoryLoad_SP_35
|       !______MemoryLoad_SP_36
|       !______MemoryLoad_SP_37
|       !______MemoryLoad_SP_38
|       !______MemoryLoad_SP_39
|       !______MemoryLoad_SP_40
|
!______MemoryLoad_DNF_6
|       !______MemoryLoad_SP_41
|       !______MemoryLoad_SP_42
|       !______MemoryLoad_SP_43
|       !______MemoryLoad_SP_44
|       !______MemoryLoad_SP_45
|       !______MemoryLoad_SP_46
|       !______MemoryLoad_SP_47
|       !______MemoryLoad_SP_48
|
!______MemoryLoad_DNF_7
        !______MemoryLoad_SP_49
        !______MemoryLoad_SP_50
        !______MemoryLoad_SP_51
        !______MemoryLoad_SP_52
        !______MemoryLoad_SP_53
        !______MemoryLoad_SP_54
        !______MemoryLoad_SP_55
        !______MemoryLoad_SP_56
```

```
Fastest> showsch -tcl -o memoryload-clases.tex
```

El árbol de pruebas generado, como puede apreciarse, consta de dos niveles de nodos. El segundo nivel, que contiene en este caso todas las clases de prueba que son hojas del árbol, es a partir de donde se intentará generar casos de prueba abstractos. El listado de sus clases, tomadas de `memoryload-clases.tex`, puede verse en el Apéndice B.

Para generar los casos de prueba abstractos, y para enviar al archivo `memoryload-casos.tex` los resultados obtenidos, se ejecutan los siguientes comandos:

```
Fastest> genalltca
Fastest> showsch -tca -o memoryload-casos.tex
```

con los cuales se pudo encontrar una cantidad aceptable de casos de prueba[2]. El contenido de `memoryload-casos.tex` puede verse en el Apéndice C. El árbol de pruebas expandido con los casos de prueba se presenta a continuación:

```
Fastest> showtt

MemoryLoad_VIS
  !______MemoryLoad_DNF_1
  |        !______MemoryLoad_SP_1
  |        !______MemoryLoad_SP_2
  |        |        !______MemoryLoad_SP_2_TCASE
  |        |
  |        !______MemoryLoad_SP_3
  |        !______MemoryLoad_SP_4
  |        |        !______MemoryLoad_SP_4_TCASE
  |        |
  |        !______MemoryLoad_SP_5
  |        |        !______MemoryLoad_SP_5_TCASE
  |        |
  |        !______MemoryLoad_SP_6
  |        !______MemoryLoad_SP_7
  |        !______MemoryLoad_SP_8
  |
  !______MemoryLoad_DNF_2
  |        !______MemoryLoad_SP_9
  |        !______MemoryLoad_SP_10
  |        |        !______MemoryLoad_SP_10_TCASE
  |        |
  |        !______MemoryLoad_SP_11
  |        !______MemoryLoad_SP_12
  |        |        !______MemoryLoad_SP_12_TCASE
  |        |
  |        !______MemoryLoad_SP_13
  |        |        !______MemoryLoad_SP_13_TCASE
  |        |
  |        !______MemoryLoad_SP_14
```

[2]El cálculo demandó 8 horas y 6 minutos, corriendo en una computadora con un procesador AMD Athlon 64 3200+ y 2GB de memoria RAM.

```
|       |       !______MemoryLoad_SP_14_TCASE
|       |
|       !______MemoryLoad_SP_15
|       |       !______MemoryLoad_SP_15_TCASE
|       |
|       !______MemoryLoad_SP_16
|               !______MemoryLoad_SP_16_TCASE
|
|
!______MemoryLoad_DNF_3
|       !______MemoryLoad_SP_17
|       |       !______MemoryLoad_SP_17_TCASE
|       |
|       !______MemoryLoad_SP_18
|       |       !______MemoryLoad_SP_18_TCASE
|       |
|       !______MemoryLoad_SP_19
|       |       !______MemoryLoad_SP_19_TCASE
|       |
|       !______MemoryLoad_SP_20
|       |       !______MemoryLoad_SP_20_TCASE
|       |
|       !______MemoryLoad_SP_21
|       |       !______MemoryLoad_SP_21_TCASE
|       |
|       !______MemoryLoad_SP_22
|       |       !______MemoryLoad_SP_22_TCASE
|       |
|       !______MemoryLoad_SP_23
|       |       !______MemoryLoad_SP_23_TCASE
|       |
|       !______MemoryLoad_SP_24
|               !______MemoryLoad_SP_24_TCASE
|
|
!______MemoryLoad_DNF_4
|       !______MemoryLoad_SP_25
|       |       !______MemoryLoad_SP_25_TCASE
|       |
|       !______MemoryLoad_SP_26
|       |       !______MemoryLoad_SP_26_TCASE
|       |
|       !______MemoryLoad_SP_27
|       |       !______MemoryLoad_SP_27_TCASE
```

```
|       |
|       !______MemoryLoad_SP_28
|       |       !______MemoryLoad_SP_28_TCASE
|       |
|       !______MemoryLoad_SP_29
|       |       !______MemoryLoad_SP_29_TCASE
|       |
|       !______MemoryLoad_SP_30
|       |       !______MemoryLoad_SP_30_TCASE
|       |
|       !______MemoryLoad_SP_31
|       |       !______MemoryLoad_SP_31_TCASE
|       |
|       !______MemoryLoad_SP_32
|               !______MemoryLoad_SP_32_TCASE
|
|
!______MemoryLoad_DNF_5
|       !______MemoryLoad_SP_33
|       |       !______MemoryLoad_SP_33_TCASE
|       |
|       !______MemoryLoad_SP_34
|       |       !______MemoryLoad_SP_34_TCASE
|       |
|       !______MemoryLoad_SP_35
|       |       !______MemoryLoad_SP_35_TCASE
|       |
|       !______MemoryLoad_SP_36
|       !______MemoryLoad_SP_37
|       !______MemoryLoad_SP_38
|       |       !______MemoryLoad_SP_38_TCASE
|       |
|       !______MemoryLoad_SP_39
|       |       !______MemoryLoad_SP_39_TCASE
|       |
|       !______MemoryLoad_SP_40
|               !______MemoryLoad_SP_40_TCASE
|
|
!______MemoryLoad_DNF_6
|       !______MemoryLoad_SP_41
|       |       !______MemoryLoad_SP_41_TCASE
|       |
|       !______MemoryLoad_SP_42
```

```
|       !______MemoryLoad_SP_43
|       |       !______MemoryLoad_SP_43_TCASE
|       |
|       !______MemoryLoad_SP_44
|       !______MemoryLoad_SP_45
|       !______MemoryLoad_SP_46
|       !______MemoryLoad_SP_47
|       !______MemoryLoad_SP_48
|
!______MemoryLoad_DNF_7
        !______MemoryLoad_SP_49
        !______MemoryLoad_SP_50
        !______MemoryLoad_SP_51
        !______MemoryLoad_SP_52
        !______MemoryLoad_SP_53
        !______MemoryLoad_SP_54
        !______MemoryLoad_SP_55
        !______MemoryLoad_SP_56
```

El resumen del análisis de los resultados se presenta en la Tabla 7.1. Los siguientes son dos ejemplos de casos de prueba encontrados, mostrados junto a las clases de prueba correspondientes.

MemoryLoad_SP_10

MemoryLoad_DNF_2

$$memd = \{\}$$
$$\{i : 1 \,..\, \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \neq \{\}$$

Hojas	**Éxitos de búsqueda**	**Fracasos de búsqueda**		**Porcentaje**
		Posibles	**Imposibles**	
56	**33**	**5**	**18**	**86,84 %**

Cuadro 7.1: Resúmen de la búsqueda de casos de prueba para la operación MemoryLoad.

MemoryLoad_SP_10_TCASE

MemoryLoad_SP_10

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

MemoryLoad_SP_31

MemoryLoad_DNF_4

$memd \neq \{\}$
$\{i : 1 \,..\, \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \neq \{\}$
$\mathrm{dom}\, memd \subset \mathrm{dom}\{i : 1 \,..\, \#data? \bullet i + addr? - 1 \mapsto data?\ i\}$

MemoryLoad_SP_31_TCASE

MemoryLoad_SP_31

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

Entre las clases de prueba para las que no pudieron encontrarse casos de prueba, y que están marcados como Posibles en la Tabla 7.1, se encuentras las dos siguientes:

MemoryLoad_SP_7

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

$waiting = no$
$ccmd = ML$
$0 < addr?$
$data? \neq \langle\rangle$
$addr? + \#data? \leq 43$
$addr? + \#data? \leq mep$
$memd \neq \{\}$
$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \neq \{\}$
$\text{dom}\, memd \subset \text{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\}$

MemoryLoad_SP_52

$$
\begin{array}{l}
memp, memd : \mathrm{seq}\ MDATA \\
mdp, mep, ped : \mathbb{N} \\
ctime : TIME \\
acquiring, waiting, sending, dumping, waitsignal : BIN \\
mode : CMODE \\
ccmd : CTYPE \\
addr? : \mathbb{N} \\
data? : \mathrm{seq}\ MDATA \\
low? : BIN \\
\hline
43 < addr? + \#data? \\
memd \neq \{\} \\
\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \neq \{\} \\
\mathrm{dom}\ memd = \mathrm{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\}
\end{array}
$$

donde la elección de los modelos finitos particular que hace Fastest, y que fue explicada en la Sección 5.4.3, hace imposible encontrar una combinación de valores que satisfaga los predicados que se presentan. Futuras versiones de Fastest mejorarán notablemente este aspecto, por ejemplo, dando al usuario la posibilidad de indicar qué modelos finitos deben considerarse para cada variable de cada clase de prueba.

Aquellas clases de prueba para las cuales es imposible encontrar casos de pruebas son las que tienen su predicado equivalente a *false*. También se pretenderá que próximas versiones de la herramienta ofrezcan avances en esta cuestión, pudiendo hacer una poda de aquellas clases que cumplan con esta característica.

Capítulo 8

Conclusiones y trabajo futuro

A lo largo de un año y medio de trabajo se desarrolló el prototipo de una herramienta que permite automatizar en gran medida el proceso de testing funcional basado en especificaciones formales Z. Con su ayuda, es posible hacer menos laboriosa la intervención del hombre en la etapa de testing del desarrollo de software. De esta forma, Fastest sería muy útil en cualquier proyecto de software, ya que podrían dedicarse más recursos humanos de la organización a tareas que requieran inteligencia y aptitudes creativas, sin perder tiempo en actividades metódicas.

Por supuesto, ninguna herramienta es la solución a todos los problemas, y ésta no es una excepción. Como todo prototipo, éste no ofrece la funcionalidad completa del sistema final. El lenguaje de especificación Z aún no está completamente soportado, y la búsqueda de casos de prueba todavía tiene muchísimo para mejorar, principalmente en cuestiones de eficiencia. Sin embargo, se ha realizado un desarrollo a partir de una arquitectura y diseño tan flexibles que admiten numerosas posibilidades de evolución, y que por lo tanto, dan lugar a pronósticos muy optimistas.

Finalmente, se pudo emplear en general sólo tecnología de distribución libre y código fuente abierto, lo que constituía un objetivo tanto del autor de este trabajo como de su director.

8.1. Trabajos futuros

La siguiente lista presenta algunas funcionalidades que pueden agregarse al prototipo de Fastest en versiones futuras:

- Como se explicó en la Sección 5.4.3, la generación de casos de prueba abstractos, en Fastest, depende del modelo finito elegido para representar la clase de prueba para la que quiere encontrarse un caso de prueba. Es por esto que una alternativa interesante sería desarrollar una estrategia de búsqueda de casos de prueba que cuente con la intervención del usuario. Éste podría sugerir modelos finitos para los tipos más simples (como los naturales y los enteros), de cuya composición se obtendrían modelos finitos para tipos más complejos y, eventualmente, el modelo finito de la clase de prueba a analizar. Esta participación del usuario podría hacer más efectiva la generación de casos de prueba, no en cuestiones de tiempo, sino en términos de cantidad de

clases de prueba para las que se encontró un caso sobre cantidad de clases de prueba para las que alguno debería poder encontrarse.

- En relación al punto anterior se encuentra la evaluación de los predicados que resultan de reemplazar sus variables por los valores que le corresponden, tomados del modelo finito de la clase de prueba. Como se vio en la Sección 5.4.3, es en este cálculo, a través del animador ZLive, de CZT, donde radica una de las principales limitaciones actuales de Fastest: lo considerablemente costoso que resulta buscar casos de prueba abstractos. Por esto es que el desarrollo de un evaluador de expresiones y predicados Z propio, más eficiente, sería otro proyecto interesante.

- Un aspecto imprescindible de abordar en el futuro sería el de la poda de los árboles de prueba, a través de la cual se descartarían las clases de prueba vacías (aquellas que tengan su predicado esquivalente a *false*). Esta cuestión es fundamental para evitar que se hagan evaluaciones de predicados que ya se sabe no son satisfactibles, y que también conducen a un proceso de generación de casos de prueba más ineficiente. Para esto, habría que analizar la posibilidad de contar con reglas de simplificación de predicados que permitan hacer evidente cuando se está frente a un predicado con las características mencionadas.

- Naturalmente, si bien este prototipo de la herramienta soporta tres tácticas de testing muy útiles: Forma Normal Disyuntiva (DNF), Particiones Estándar (SP) y Tipos Libres (FT); una línea de trabajo futuro clave sería el agregado de tácticas adicionales. Podrían considerarse las demás tácticas sugeridas en [6]: Mutación de Especificaciones, Propagación de Subdominios, y Causa-Efecto. Por otro lado, sería posible diseñar nuevas tácticas de testing, que sean propias, y agregarlas a Fastest.

- En la Sección 5.4.2 se mencionó la forma en que la herramienta aplica las tácticas de testing para construir árboles de prueba, considerando una lista de tácticas por operación y, al intentar encontrar clases de prueba que cuelguen de cierta clase, recorrer la lista de tácticas hasta que alguna de ellas permita obtener nuevas clases. Como se describió en la Sección 3.5.1, sería deseable poder elegir a qué clases de prueba aplicar qué tácticas de testing, paso por paso, de forma tal de contar con un proceso más flexible de generación de árboles de prueba.

- Como se ha mencionado en repetidas oportunidades, la versión actual de Fastest no soporta completamente el lenguaje Z. Sin lugar a dudas, un cubrimiento total del lenguaje haría que la herramienta sea mucho más útil; ya no sería necesario escribir especificaciones de sistemas de manera forzada, teniendo en cuenta las limitaciones sintácticas que existen al día de hoy.

- El desarrollo de Fastest fue impulsado por la idea de automatizar el proceso de testing funcional basado en especificaciones formales Z. Sin embargo, como se ha descripto a lo largo del trabajo, el prototipo aquí presentado sólo implementa la etapa de generación de casos de prueba abstractos. Evidentemente, será necesario investigar y desarrollar las etapas posteriores en algún momento. Es necesario remarcar que Diego Hollmann está realizando su tesina de grado sobre el tema del refinamiento de casos de prueba. Como el trabajo de Diego se está apoyando en la existencia de Fastest, se considera probable que ambas partes puedan integrarse a la brevedad.

- Si bien fue contemplado a la hora de desarrollar la arquitectura de la herramienta, todavía no fue implementado un servidor de datos. A través de él sería posible centralizar información relacionada al proceso de testing (como casos de prueba abstractos obtenidos en sesiones anteriores, definiciones de nuevas tácticas de testing, reglas de refinamiento, de abstracción, etc.) y a la comunicación de todo el sistema (como ubicación de los servidores de cómputo, por ejemplo). De esta manera se evitaría tener que actualizar cierta información en los clientes y en los servidores, según corresponda, cada vez que ésta se modifica.
- Por último, otra opción interesante de trabajo futuro podría ser desarrollar una interfaz gráfica de usuario para Fastest, de forma de hacer más sencilla y amigable la interacción con el usuario.

Apéndice A

Glosario de Z

Este apéndice hará un resumen no exhaustivo del lenguaje de especificación Z. Además, indicará aquellos elementos del lenguaje que no están soportados actualmente por la herramienta que esta tesina describe.

A.1. Tipos básicos, definiciones y declaraciones

Sean los identificadores X, X_1, X_2, ..., X_n, las expresiones $EXPR_{IZQ}$ y $EXPR_{DER}$ y los conjuntos T, T_1, T_2, ..., T_n.

$[X]$

Introduce un tipo básico llamado X. Su estructura no se restringe en esta definición, pero se lo puede hacer en el resto de la especificación a través de uno o más predicados. Para introducir más de un tipo básico a la vez, se los lista entre comas y encerrados entre corchetes, como en $[X_1, X_2, ..., X_n]$.

$EXPR_{IZQ} == EXPR_{DER}$

Define a $EXPR_{IZQ}$ como expresión equivalente a $EXPR_{DER}$.

$x : T$

Declaración que introduce una nueva variable x de tipo T.

$x_1 : T_1; x_2 : T_2; ...; x_n : T_n$

Declaración que introduce las variables listadas, cada una con su tipo correspondiente.

$x_1, x_2, ..., x_n : T$

Una declaración como esta introduce las variables listadas, todas del mismo tipo T.

A.2. Definiciones axiomáticas

Una definición axiomática introduce variables globales, es decir, variables cuyo alcance es toda la especificación. Sean D una lista de declaraciones y P un predicado. La siguiente es una definición axiomática que introduce las variables declaradas en D, de tal manera que satisfagan el predicado P.

$$\begin{array}{|l} D \\ \hline P \end{array}$$

Por ejemplo,

$$\begin{array}{|l} a, b : \mathbb{Z} \\ \hline a + b = 7 \end{array}$$

introduce dos variables enteras a y b tales que su suma es igual a 7.

Los elementos declarados en una definición axiomática son siempre constantes, no variables. El motivo por el que se los suele llamar variables es porque sus valores (fijos) suelen ser desconocidos o indeterminados. Por ambas razones, es común llamarlos *variables rígidas*.

La versión actual de la herramienta presentada en este trabajo no soporta especificaciones con definiciones axiomáticas.

A.3. Tipos libres

Sean los identificadores X, $constr_1$ y $constr_2$ y la expresión T que denota un tipo. La siguiente construcción del lenguaje Z:

$$X ::= constr_1 \mid constr_2 \langle\langle T \rangle\rangle$$

permite introducir un nuevo tipo X, así como definir los constructores que generan sus elementos, $constr_1$ y $constr_2$. A los tipos definidos de esta manera se los llama *tipos libres* y todos sus elementos deben ser generados por algún constructor. Los constructores pueden ser:

- Un *identificador* ($constr_1$), el cual es un elemento del nuevo tipo.
- Un *constructor de función* ($constr_2$), el cual es una función que toma un argumento de tipo T y devuelve un elemento del nuevo tipo. Si a un constructor de función se le pasan valores distintos como argumentos, se obtienen elementos distintos.

Los tipos libres son útiles para definir conjuntos inductivos, como los árboles. Por ejemplo, el tipo libre dado a continuación, representa un árbol binario:

$$ArbolBin ::= hoja \mid nodoInterno \langle\langle ArbolBin \times ArbolBin \rangle\rangle$$

En el caso más trivial, el tipo libre que se introduce no contiene constructores de función, por lo que no define con un conjunto inductivo. Se dirá que en una situación así el tipo libre es, en particular, un *tipo enumerado*. Un ejemplo de un tipo enumerado es el siguiente:

$Color ::= rojo \mid azul \mid amarillo$

La versión actual de la herramienta presentada en este trabajo sólo soporta los tipos libres que sean tipos enumerados.

A.4. Operadores aritméticos y relacionales

El paquete matemático del lenguaje Z provee los operadores aritméticos usuales de suma, resta y multiplicación, que se denotan $+$, $-$ y $\times$, respectivamente. Como el paquete no define número reales ni racionales, no hay forma de representar fracciones, y por lo tanto, no está disponible la división fraccionaria en el lenguaje. Sin embargo, el paquete matemático sí provee los operadores de división entera y de módulo, a través de los operadores *div* y *mod*, respectivamente. Z también hace posible vincular expresiones aritméticas mediante el uso de los siguientes operadores relacionales: $<$, $\leq$, $>$ y $\geq$.

A.5. Cálculo de predicados

Los dos predicados más simples del lenguaje Z son *true* y *false*. Ambos conforman las *constantes lógicas* o *valores de verdad*, y de hecho, todos los predicados del lenguaje tienen un valor o el otro.

Se pueden combinar predicados simples, para formar otros más complejos, a través de los *conectivos lógicos* usuales: $\neg$ (*negación*), $\wedge$ (*conjunción*), $\vee$ (*disyunción*), $\Rightarrow$ (*implicación*) y $\Leftrightarrow$ (*equivalencia*). El valor de verdad de un predicado que contiene conectivos lógicos está determinado por los valores de verdad de sus predicados constituyentes.

Z provee las cuantificaciones universales y existenciales del cálculo de predicados. La forma general de la cuantificación universal es:

$\forall\, declaracion \bullet predicado$

y la de la cuantificación existencial es:

$\exists\, declaracion \bullet predicado$

A.6. Expresiones, conjuntos y relaciones

Sean la expresión e, las expresiones e_1 y e_2 del mismo tipo y los conjuntos S, T, T_1, T_2, ..., T_n.

$e_1 = e_2$

Denota el predicado de *igualdad* entre las expresiones e_1 y e_2. El predicado es equivalente a *true* si e_1 tiene el mismo valor que e_2 y es equivalente a *false* en caso contrario. El lenguaje provee también el predicado de *desigualdad*, $e_1 \neq e_2$, cuya interpretación es la inversa.

$e \in S$

Denota el predicado de *pertenencia* entre el elemento e y el conjunto S. El predicado es equivalente a *true* si e es un elemento del conjunto S y es equivalente a *false* en caso contrario. El lenguaje provee también el predicado de *no pertenencia*, $e \notin S$, cuya interpretación es la inversa.

$S \cup T$, $S \cap T$, $S \setminus T$

Denotan, respectivamente, el conjunto *unión*, *intersección* y *diferencia* entre los conjuntos S y T, con sus definiciones usuales.

$S \subseteq T$ y $S \subset T$

Denotan, respectivamente, el predicado de *inclusión* y de *inclusión estricta* entre los conjuntos S y T, con sus definiciones usuales.

$\mathbb{P}\, T$

Denota el *conjunto de subconjuntos* de T. Un variable declarada con tipo $\mathbb{P}\, T$ es tal que sus valores son conjuntos, en particular, subconjuntos de T.

$\#T$

Denota el *cardinal* de T, es decir, un número natural igual a la cantidad de elementos del conjunto T.

$T_1 \times T_2 \times \ ... \ \times T_n$

Describe el tipo que corresponde al *producto cartesiano* entre los tipos T_1, T_2, ..., T_n. Un elemento de este tipo se llama *tupla*. Por ejemplo, si $DIA == 1 \,..\, 31$, $MES == 1 \,..\, 12$ y $ANIO == \mathbb{N}$, entonces (20,6,1983), (12,10,1492) y (9,7,1816) son tuplas de tipo $DIA \times MES \times ANIO$.

$T_1 \leftrightarrow T_2 \leftrightarrow ... \leftrightarrow T_n$

Denota una relación entre T_1, T_2, ..., y T_n. Una *relación* es un conjunto de tuplas, por lo que los operadores definidos para conjuntos pueden ser usados sobre las relaciones.

A.7. Relaciones binarias, funciones y secuencias

Una *relación binaria* es un conjunto de *pares*, los cuales son tuplas de dos elementos. Sean los conjuntos X, Y, y Z; las variables x, x_1, x_2, ..., $x_n : X$; y, y_1, y_2, ..., $y_n : Y$; S un subconjunto de X; T un subconjunto de Y y las relaciones $R, R1 : X \leftrightarrow Y$ y $R2 : Y \leftrightarrow Z$.

$x \mathrel{\underline{R}} y$

Significa que x está relacionado con y a través de R. Equivalentemente, esta expresión se puede escribir como $(x, y) \in R$. Una relación puede ser usada como un operador infijo subrayando su nombre. Contrariamente, puede hacerse referencia a la relación correspondiente a un operador infijo, como $<$, con el nombre que resulta de colocar guiones bajos y paréntesis a los lados del operador. Por ejemplo, al operador $<$ le corresponde la relación $(\_ < \_)$, y por lo tanto, $(x < y) \Leftrightarrow (x, y) \in (\_ < \_)$.

$\mathrm{dom}\, R$

Denota el *dominio* de la relación R, es decir, el conjunto de elementos $x \in X$ que están relacionados por R con algún elemento $y \in Y$.

$\mathrm{ran}\, R$

Denota el *rango* de la relación R, es decir, el conjunto de elementos $y \in Y$ que están relacionados por R con algún elemento $x \in X$.

$R_1 \fatsemi R_2$

Denota la *composición relacional* entre R_1 y R_2. La composición relaciona x con z si hay algún y tal que $(x, y) \in R_1$ y $(y, z) \in R_2$.

$R^{\sim}$

Denota la *relación inversa* de R, que puede definirse a través de la equivalencia $(x, y) \in R^{\sim} \Leftrightarrow (y, x) \in R$.

$\mathrm{id}\, S$

Denota la *función identidad* sobre el conjunto S, que hace corresponder cada $x \in S$ consigo mismo.

R^{+}

Denota la *clausura transitiva* de R. Un par (x_1, x_n) está en la relación R^{+} si y sólo si existe una secuencia finita de valores x_1, x_2, ..., x_n, donde $n \geq 2$, tal que $(x_1, x_2) \in R$, $(x_2, x_3) \in R$, ..., $(x_{n-1}, x_n) \in R$.

R^*

Denota la *clausura reflexiva y transitiva* de R. Equivale a la relación $R^+ \cup \mathrm{id}\, X$.

$R(\!|\ S\ |\!)$

Denota la *imagen relacional* entre la relación R y el conjunto S, es decir, el subconjunto de Y cuyos elementos están relacionados con los de S a través de R.

$S \lhd R$

Denota la *restricción de dominio* entre la relación R y el conjunto S, es decir, la relación R con su dominio restringido a S.

$S \lhd\!\!\!\!- R$

Denota la *substracción de dominio* entre la relación R y el conjunto S, es decir, la relación R con su dominio restringido al complemento de S.

$R \rhd T$

Denota la *restricción del rango* entre la relación R y el conjunto T, es decir, la relación R con su rango restringido al conjunto T.

$R \rhd\!\!\!\!- T$

Denota la *substracción del rango* entre la relación R y el conjunto T, es decir, la relación R con su rango restringido al complemento del conjunto T.

$R_1 \oplus R_2$

Denota la *operación de reescritura* entre las relaciones R_1 y R_2, que resulta en una relación con las tuplas de ambas, excepto que las tuplas de R_2 reemplazan a las de R_1 que tengan la misma primer componente.

$X \to Y$

Denota el conjunto de *funciones totales* de X a Y. En estas funciones, cada elemento de X está relacionado con exactamente un elemento de Y.

$X \nrightarrow Y$

Denota el conjunto de *funciones parciales* de X a Y. En estas funciones, cada elemento de X puede estar relacionado a lo sumo con un elemento de Y.

$X \rightarrowtail Y$, $X \rightarrowtail Y$, $X \twoheadrightarrow Y$, $X \twoheadrightarrow Y$, $X \rightarrowtail\!\!\!\!\!\twoheadrightarrow Y$, $X \twoheadrightarrow\!\!\!\!\!\rightarrowtail Y$ y $X \twoheadrightarrow\!\!\!\!\!\rightarrowtail Y$

Denotan, respectivamente, los conjuntos de *funciones inyectivas*, *funciones inyectivas parciales*, *funciones suryectivas*, *funciones suryectivas parciales*, *funciones biyectivas*, *funciones finitas* y *funciones inyectivas finitas* de X a Y. La versión actual de la herramienta que esta tesina describe no soporta estos tipos de funciones.

$\mathrm{seq}\, X$

Denota el conjunto de secuencias de elementos X. Una *secuencia* es un conjunto ordenado de elementos y pueden verse como funciones cuyos dominios son números consecutivos, a partir del 1. Una secuencia particular se puede representar escribiendo sus elementos separados por comas entre corchetes angulares, como en $\langle 21, 4, 8\rangle$.

$S_1 \frown S_2$

Denota la *concatenación* de las secuencias S_1 y S_2. Por ejemplo, $\langle 1, 2, 3\rangle \frown \langle 4, 5\rangle = \langle 1, 2, 3, 4, 5\rangle$.

A.8. Definición de esquemas

Un esquema permite agrupar declaraciones de variables y un predicado que restringe los valores de éstas variables. Si el predicado se omite, se asume que el mismo es equivalente a *true*, lo que significa que no se hace ninguna restricción sobre las variables declaradas. Los esquemas son las construcciones fundamentales del lenguaje Z, a través de los cuales se definen los estados y las transiciones de estados de un modelo. Por ejemplo, el esquema

S

$x, y : \mathbb{Z}$

$x \geq 0$

$y \geq 0$

describe un estado que representa el primer cuadrante de pares ordenados enteros. Se dice que x e y son las *variables de estado* del esquema S. El mismo esquema también puede definirse más concisamente en forma horizontal, de la siguiente manera:

$$S == [x, y : \mathbb{Z} \mid x \geq 0 \land y \geq 0]$$

A.9. Operadores del cálculo de esquemas

Sean S el esquema definido en la sección anterior y $w : S$.

$w.x$

Denota la aplicación de una *función de proyección*. Los nombres de las componentes de un esquema pueden ser usados como selectores o funciones de proyección. En este caso, $w.x$ es la componente x de w y $w.y$ es su componente y.

Inclusión

Un esquema S puede incluirse en las declaraciones de un esquema R, en cuyo caso las declaraciones de S se unen con las declaraciones de R y los predicados de S y R se conjugan. Por ejemplo,

$$
\begin{array}{|l}
\hline
R \\
\hline
S \\
z : \mathbb{Z} \\
\hline
z < x \\
\hline
\end{array}
$$

es equivalente a:

$$
\begin{array}{|l}
\hline
R \\
\hline
x, y, z : \mathbb{Z} \\
\hline
x \geq 0 \\
y \geq 0 \\
z < x \\
\hline
\end{array}
$$

Decoración

Se puede hacer un renombramiento de todas las variables declaradas en un esquema "decorando" el nombre del esquema con un subíndice, superíndice, o un apóstrofe, entre otros. Por ejemplo, S' es:

$$[x', y' : \mathbb{Z} \mid x' \geq 0 \land y' \geq 0]$$

También se pueden hacer múltiples decoraciones de un esquema. Por ejemplo, es posible escribir S''.

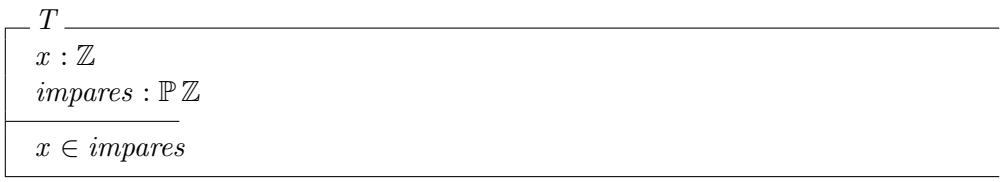
$\neg\, S$

Denota la *negación* del esquema S. Esto es, el esquema S con su predicado negado. Por ejemplo, $\neg\, S$ es:

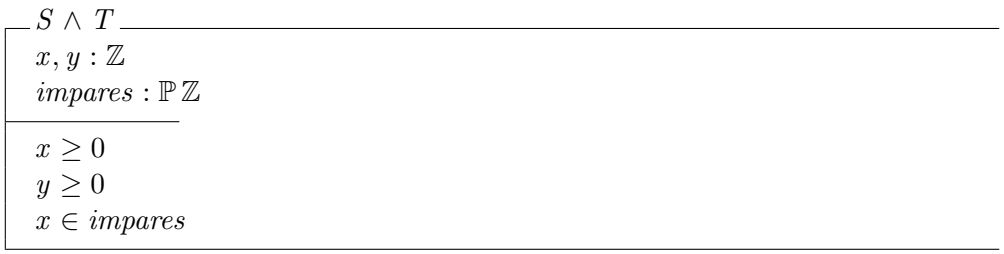
$$[x, y : \mathbb{Z} \mid \neg\, (x \geq 0 \land y \geq 0)]$$

$S \land T$

Denota la *conjunción* de los esquemas S y T. Esto es, el esquema formado por los esquemas S y T uniendo sus declaraciones y conjugando sus predicados. Por ejemplo, dado:

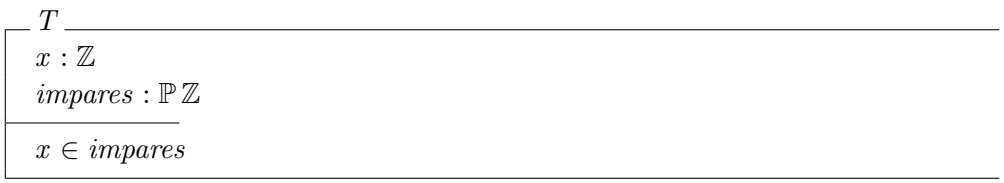


$S \land T$ es:

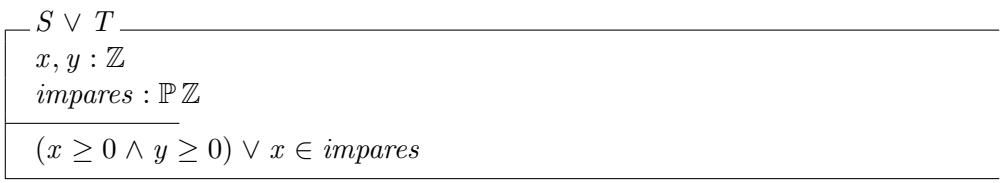


$S \lor T$

Denota la *disyunción* de los esquemas S y T. Esto es, el esquema formado por los esquemas S y T uniendo sus declaraciones y haciendo la disyunción de sus predicados. Por ejemplo, dado:



$S \lor T$ es:



$S \Rightarrow T$ y $S \Leftrightarrow T$

Denotan la *implicación* y la *equivalencia* entre los esquemas S y T, respectivamente. Sus resultados son análogos a los vistos para la conjunción y la disyunción de esquemas.

$S \setminus (v_1, v_2, ..., v_n)$

Denota el *ocultamiento* entre el esquema S y las variables $v_1, v_2, ..., v_n$. Esto es, el esquema S con las variables $v_1, v_2, ..., v_n$ removidas de las declaraciones y cuantificadas existencialmente en el predicado. Por ejemplo, $S \setminus (x)$ es:

$S \setminus (x)$

$y : \mathbb{Z}$

$(\exists\, x : \mathbb{Z} \bullet x \geq 0)$

$y \geq 0$

$S \upharpoonright (v_1, v_2, ..., v_n)$

Denota la *proyección* entre el esquema S y las variables $v_1, v_2, ..., v_n$. Esto es, el esquema S con el ocultamiento de cualquiera de las variables que no aparezcan en la lista $v_1, v_2, ..., v_n$.

La versión actual de Fastest no soporta las operaciones de implicación, equivalencia, ocultamiento y proyección recién listadas.

A.10. Esquemas de operación

Los eventos en un sistema especificado en el lenguaje Z se modelan como operaciones sobre los estados del sistema. Las operaciones se describen en función de un estado inicial y uno final y también se definen utilizando esquemas, los que en particular se denominan *esquemas de operación*. Para distinguir variables que representen un estado inicial de aquellas que representen un estado final, Z usa la convención lexicográfica de agregar un apóstrofe ($'$) al final del nombre de las variables de estado. Similarmente, las variables de entrada (argumentos) de una operación y las variables de salida (resultados) de una operación, se distinguen con un ? y con un ! concatenados al final de sus nombres, respectivamente. Por ejemplo, una operación que permite transformar un estado que representa un par de coordenadas en el estado que representa las coordenadas transpuestas, se puede definir como:

Transposicion

$x, y : \mathbb{Z}$

$x', y' : \mathbb{Z}$

$x \geq 0$

$y \geq 0$

$x' \geq 0$

$y' \geq 0$

$x' = y$

$y' = x$

Por convención, ΔS es el esquema definido como $S \land S'$, representando los estados iniciales y finales e incluyendo los predicados para ambos estados. Teniendo en cuenta esto y la inclusión de esquemas, se puede reescribir el esquema *Transposicion* de la siguiente manera:

Transposicion

ΔS

$x' = y$
$y' = x$

la cual es claramente más concisa que la anterior.

Otro ejemplo es la operación que calcula el cuadrado de distancia al origen de las coordenadas representadas en un estado:

Distancia

ΞS
$dist : \mathbb{Z}$

$dist = x \times x + y \times y$

El esquema ΞS es similar a ΔS en el hecho de que define dos estados, pero además de esto le agrega la restricción adicional de que los dos estados deben ser iguales. Esta decoración Ξ se utiliza en operaciones como *Distancia*, donde no se modifica el estado del sistema.

A.11. Otros operadores del cálculo de esquemas

Sean Q y R dos esquemas de operación. A continuación se describen operadores que son exclusivos para los esquemas de operación.

$Q \fatsemi R$

Denota la *composición* entre los esquemas de operación Q y R. Esto es, el esquema de operación que comienza con el estado inicial de Q y termina en el estado final de R.

$Q \gg R$

Denota el *direccionamiento* entre los esquemas de operación Q y R. Esto es, el esquema de operación que combina la salida de Q con la entrada de R.

pre Q

Denota la *precondición* del esquema de operación Q. Esto es, el esquema de estado que describe los estados sobre los cuales está definida la operación Q. Este esquema es el que resulta de ocultar las variables de salida y las variable de estado primadas. Por ejemplo, dado:

Q

$x?, s, s', y! : \mathbb{N}$

$s' = s - x? \land y! = s'$

pre Q es:

pre Q

$x?, s : \mathbb{N}$

$(\exists\, s', y! : \mathbb{Z} \bullet s' = s - x? \land y! = s')$

Apéndice B

Clases de prueba resultantes del Caso de Estudio

MemoryLoad_VIS

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

$(waiting = no$
$ccmd = ML$
$0 < addr?$
$data? \neq \langle\rangle$
$addr? + \#data? \leq 43$
$addr? + \#data? \leq mep) \lor (waiting = no$
$ccmd = ML$
$0 < addr?$
$data? \neq \langle\rangle$
$addr? + \#data? \leq 43$
$mep < addr? + \#data?) \lor$
$waiting \neq no \lor$
$ccmd \neq ML \lor$
$addr? = 0 \lor$
$data? = \langle\rangle \lor$
$43 < addr? + \#data?$

MemoryLoad_DNF_1

MemoryLoad_VIS

$waiting = no$
$ccmd = ML$
$0 < addr?$
$data? \neq \langle\rangle$
$addr? + \#data? \leq 43$
$addr? + \#data? \leq mep$

MemoryLoad_SP_1

MemoryLoad_DNF_1

$memd = \{\}$
$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} = \{\}$

MemoryLoad_SP_2

MemoryLoad_DNF_1

$memd = \{\}$
$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \neq \{\}$

MemoryLoad_SP_3

MemoryLoad_DNF_1

$memd \neq \{\}$
$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} = \{\}$

MemoryLoad_SP_4

MemoryLoad_DNF_1

$memd \neq \{\}$
$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \neq \{\}$
$\mathrm{dom}\, memd = \mathrm{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\}$

MemoryLoad_SP_5

MemoryLoad_DNF_1

$memd \neq \{\}$
$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \neq \{\}$
$\mathrm{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \subset \mathrm{dom}\, memd$

MemoryLoad_SP_6

MemoryLoad_DNF_1

$memd \neq \{\}$
$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \neq \{\}$
$(\mathrm{dom}\ memd \cap \mathrm{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\}) = \{\}$

MemoryLoad_SP_7

MemoryLoad_DNF_1

$memd \neq \{\}$
$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \neq \{\}$
$\mathrm{dom}\ memd \subset \mathrm{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\}$

MemoryLoad_SP_8

MemoryLoad_DNF_1

$memd \neq \{\}$
$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \neq \{\}$
$(\mathrm{dom}\ memd \cap \mathrm{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\}) \neq \{\}$
$\neg\ \mathrm{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \subseteq \mathrm{dom}\ memd$
$\neg\ \mathrm{dom}\ memd \subseteq \mathrm{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\}$

MemoryLoad_DNF_2

MemoryLoad_VIS

$waiting = no$
$ccmd = ML$
$0 < addr?$
$data? \neq \langle\rangle$
$addr? + \#data? \leq 43$
$mep < addr? + \#data?$

MemoryLoad_SP_9

MemoryLoad_DNF_2

$memd = \{\}$
$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} = \{\}$

MemoryLoad_SP_10

MemoryLoad_DNF_2

$memd = \{\}$
$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \neq \{\}$

MemoryLoad_SP_11

MemoryLoad_DNF_2

$memd \neq \{\}$
$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} = \{\}$

MemoryLoad_SP_12

MemoryLoad_DNF_2

$memd \neq \{\}$
$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \neq \{\}$
$\mathrm{dom}\ memd = \mathrm{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\}$

MemoryLoad_SP_13

MemoryLoad_DNF_2

$memd \neq \{\}$
$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \neq \{\}$
$\mathrm{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \subset \mathrm{dom}\ memd$

MemoryLoad_SP_14

MemoryLoad_DNF_2

$memd \neq \{\}$
$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \neq \{\}$
$(\mathrm{dom}\ memd \cap \mathrm{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\}) = \{\}$

MemoryLoad_SP_15

MemoryLoad_DNF_2

$memd \neq \{\}$
$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \neq \{\}$
$\mathrm{dom}\ memd \subset \mathrm{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\}$

MemoryLoad_SP_16

MemoryLoad_DNF_2

$memd \neq \{\}$
$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \neq \{\}$
$(\mathrm{dom}\ memd \cap \mathrm{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\}) \neq \{\}$
$\neg\ \mathrm{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \subseteq \mathrm{dom}\ memd$
$\neg\ \mathrm{dom}\ memd \subseteq \mathrm{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\}$

MemoryLoad_DNF_3

MemoryLoad_VIS

$waiting \neq no$

MemoryLoad_SP_17

MemoryLoad_DNF_3

$memd = \{\} \land \{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} = \{\}$

MemoryLoad_SP_18

MemoryLoad_DNF_3

$memd = \{\} \land \{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \neq \{\}$

MemoryLoad_SP_19

MemoryLoad_DNF_3

$memd \neq \{\} \land \{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} = \{\}$

MemoryLoad_SP_20

MemoryLoad_DNF_3

$memd \neq \{\}$
$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \neq \{\}$
$\mathrm{dom}\, memd = \mathrm{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\}$

MemoryLoad_SP_21

MemoryLoad_DNF_3

$memd \neq \{\}$
$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \neq \{\}$
$\mathrm{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \subset \mathrm{dom}\, memd$

MemoryLoad_SP_22

MemoryLoad_DNF_3

$memd \neq \{\}$
$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \neq \{\}$
$(\mathrm{dom}\, memd \cap \mathrm{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\}) = \{\}$

MemoryLoad_SP_23

MemoryLoad_DNF_3

$memd \neq \{\}$
$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \neq \{\}$
$\mathrm{dom}\ memd \subset \mathrm{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\}$

MemoryLoad_SP_24

MemoryLoad_DNF_3

$memd \neq \{\}$
$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \neq \{\}$
$(\mathrm{dom}\ memd \cap \mathrm{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\}) \neq \{\}$
$\neg\ \mathrm{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \subseteq \mathrm{dom}\ memd$
$\neg\ \mathrm{dom}\ memd \subseteq \mathrm{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\}$

MemoryLoad_DNF_4

MemoryLoad_VIS

$ccmd \neq ML$

MemoryLoad_SP_25

MemoryLoad_DNF_4

$memd = \{\}$
$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} = \{\}$

MemoryLoad_SP_26

MemoryLoad_DNF_4

$memd = \{\}$
$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \neq \{\}$

MemoryLoad_SP_27

MemoryLoad_DNF_4

$memd \neq \{\}$
$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} = \{\}$

MemoryLoad_SP_28

MemoryLoad_DNF_4

$memd \neq \{\}$
$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \neq \{\}$
$\mathrm{dom}\ memd = \mathrm{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\}$

MemoryLoad_SP_29

MemoryLoad_DNF_4

$memd \neq \{\}$
$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \neq \{\}$
$\mathrm{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \subset \mathrm{dom}\ memd$

MemoryLoad_SP_30

MemoryLoad_DNF_4

$memd \neq \{\}$
$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \neq \{\}$
$(\mathrm{dom}\ memd \cap \mathrm{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\}) = \{\}$

MemoryLoad_SP_31

MemoryLoad_DNF_4

$memd \neq \{\}$
$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \neq \{\}$
$\mathrm{dom}\ memd \subset \mathrm{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\}$

MemoryLoad_SP_32

MemoryLoad_DNF_4

$memd \neq \{\}$
$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \neq \{\}$
$(\mathrm{dom}\ memd \cap \mathrm{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\}) \neq \{\}$
$\neg\ \mathrm{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \subseteq \mathrm{dom}\ memd$
$\neg\ \mathrm{dom}\ memd \subseteq \mathrm{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\}$

MemoryLoad_DNF_5

MemoryLoad_VIS

$addr? = 0$

MemoryLoad_SP_33

MemoryLoad_DNF_5

$memd = \{\}$

$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} = \{\}$

MemoryLoad_SP_34

MemoryLoad_DNF_5

$memd = \{\}$

$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \neq \{\}$

MemoryLoad_SP_35

MemoryLoad_DNF_5

$memd \neq \{\}$

$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} = \{\}$

MemoryLoad_SP_36

MemoryLoad_DNF_5

$memd \neq \{\}$

$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \neq \{\}$

$\operatorname{dom} memd = \operatorname{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\}$

MemoryLoad_SP_37

MemoryLoad_DNF_5

$memd \neq \{\}$

$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \neq \{\}$

$\operatorname{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \subset \operatorname{dom} memd$

MemoryLoad_SP_38

MemoryLoad_DNF_5

$memd \neq \{\}$

$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \neq \{\}$

$(\operatorname{dom} memd \cap \operatorname{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\}) = \{\}$

MemoryLoad_SP_39

MemoryLoad_DNF_5

$memd \neq \{\}$
$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \neq \{\}$
$\mathrm{dom}\ memd \subset \mathrm{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\}$

MemoryLoad_SP_40

MemoryLoad_DNF_5

$memd \neq \{\}$
$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \neq \{\}$
$(\mathrm{dom}\ memd \cap \mathrm{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\}) \neq \{\}$
$\neg\ \mathrm{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \subseteq \mathrm{dom}\ memd$
$\neg\ \mathrm{dom}\ memd \subseteq \mathrm{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\}$

MemoryLoad_DNF_6

MemoryLoad_VIS

$data? = \langle\rangle$

MemoryLoad_SP_41

MemoryLoad_DNF_6

$memd = \{\}$
$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} = \{\}$

MemoryLoad_SP_42

MemoryLoad_DNF_6

$memd = \{\}$
$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \neq \{\}$

MemoryLoad_SP_43

MemoryLoad_DNF_6

$memd \neq \{\}$
$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} = \{\}$

MemoryLoad_SP_44

MemoryLoad_DNF_6

$memd \neq \{\}$
$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \neq \{\}$
$\mathrm{dom}\ memd = \mathrm{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\}$

MemoryLoad_SP_45

MemoryLoad_DNF_6

$memd \neq \{\}$
$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \neq \{\}$
$\mathrm{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \subset \mathrm{dom}\ memd$

MemoryLoad_SP_46

MemoryLoad_DNF_6

$memd \neq \{\}$
$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \neq \{\}$
$(\mathrm{dom}\ memd \cap \mathrm{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\}) = \{\}$

MemoryLoad_SP_47

MemoryLoad_DNF_6

$memd \neq \{\}$
$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \neq \{\}$
$\mathrm{dom}\ memd \subset \mathrm{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\}$

MemoryLoad_SP_48

MemoryLoad_DNF_6

$memd \neq \{\}$
$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \neq \{\}$
$(\mathrm{dom}\ memd \cap \mathrm{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\}) \neq \{\}$
$\neg\ \mathrm{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \subseteq \mathrm{dom}\ memd$
$\neg\ \mathrm{dom}\ memd \subseteq \mathrm{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\}$

MemoryLoad_DNF_7

MemoryLoad_VIS

$43 < addr? + \#data?$

MemoryLoad_SP_49

MemoryLoad_DNF_7

$memd = \{\}$

$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} = \{\}$

MemoryLoad_SP_50

MemoryLoad_DNF_7

$memd = \{\}$

$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \neq \{\}$

MemoryLoad_SP_51

MemoryLoad_DNF_7

$memd \neq \{\}$

$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} = \{\}$

MemoryLoad_SP_52

MemoryLoad_DNF_7

$memd \neq \{\}$

$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \neq \{\}$

$\operatorname{dom} memd = \operatorname{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\}$

MemoryLoad_SP_53

MemoryLoad_DNF_7

$memd \neq \{\}$

$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \neq \{\}$

$\operatorname{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \subset \operatorname{dom} memd$

MemoryLoad_SP_54

MemoryLoad_DNF_7

$memd \neq \{\}$

$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \neq \{\}$

$(\operatorname{dom} memd \cap \operatorname{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\}) = \{\}$

MemoryLoad_SP_55

MemoryLoad_DNF_7

$memd \neq \{\}$
$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \neq \{\}$
$\mathrm{dom}\ memd \subset \mathrm{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\}$

MemoryLoad_SP_56

MemoryLoad_DNF_7

$memd \neq \{\}$
$\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \neq \{\}$
$(\mathrm{dom}\ memd \cap \mathrm{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\}) \neq \{\}$
$\neg\ \mathrm{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\} \subseteq \mathrm{dom}\ memd$
$\neg\ \mathrm{dom}\ memd \subseteq \mathrm{dom}\{i : 1 \mathinner{..} \#data? \bullet i + addr? - 1 \mapsto data?\ i\}$

Apéndice C

Casos de prueba resultantes del Caso de Estudio

MemoryLoad_SP_2_TCASE

MemoryLoad_SP_2

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

MemoryLoad_SP_4_TCASE

MemoryLoad_SP_4

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

MemoryLoad_SP_5_TCASE

MemoryLoad_SP_5

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

MemoryLoad_SP_10_TCASE

MemoryLoad_SP_10

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

MemoryLoad_SP_12_TCASE

MemoryLoad_SP_12

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

MemoryLoad_SP_13_TCASE

MemoryLoad_SP_13

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

MemoryLoad_SP_14_TCASE

MemoryLoad_SP_14

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

MemoryLoad_SP_15_TCASE

MemoryLoad_SP_15

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

MemoryLoad_SP_16_TCASE

MemoryLoad_SP_16

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

MemoryLoad_SP_17_TCASE

MemoryLoad_SP_17

$dumping = no$
$mep = 0$
$ccmd = RM$
$low? = no$
$ped = 0$
$ctime = 0$
$mdp = 0$
$data? = \emptyset$
$memp = \emptyset$
$acquiring = no$
$waiting = yes$
$memd = \emptyset$
$waitsignal = no$
$addr? = 0$
$sending = no$
$mode = COF$

MemoryLoad_SP_18_TCASE

MemoryLoad_SP_18

$dumping = no$
$mep = 0$
$ccmd = RM$
$low? = no$
$ped = 0$
$ctime = 0$
$mdp = 0$
$data? = \langle mdata0 \rangle$
$memp = \emptyset$
$acquiring = no$
$waiting = yes$
$memd = \emptyset$
$waitsignal = no$
$addr? = 0$
$sending = no$
$mode = COF$

MemoryLoad_SP_19_TCASE

MemoryLoad_SP_19

$dumping = no$
$mep = 0$
$ccmd = RM$
$low? = no$
$ped = 0$
$ctime = 0$
$mdp = 0$
$data? = \emptyset$
$memp = \emptyset$
$acquiring = no$
$waiting = yes$
$memd = \langle mdata0 \rangle$
$waitsignal = no$
$addr? = 0$
$sending = no$
$mode = COF$

MemoryLoad_SP_20_TCASE

MemoryLoad_SP_20

$dumping = no$
$mep = 0$
$ccmd = RM$
$low? = no$
$ped = 0$
$ctime = 0$
$mdp = 0$
$data? = \langle mdata0 \rangle$
$memp = \emptyset$
$acquiring = no$
$waiting = yes$
$memd = \langle mdata0 \rangle$
$waitsignal = no$
$addr? = 1$
$sending = no$
$mode = COF$

MemoryLoad_SP_21_TCASE

MemoryLoad_SP_21

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

MemoryLoad_SP_22_TCASE

MemoryLoad_SP_22

$dumping = no$
$mep = 0$
$ccmd = RM$
$low? = no$
$ped = 0$
$ctime = 0$
$mdp = 0$
$data? = \langle mdata0 \rangle$
$memp = \emptyset$
$acquiring = no$
$waiting = yes$
$memd = \langle mdata0 \rangle$
$waitsignal = no$
$addr? = 0$
$sending = no$
$mode = COF$

MemoryLoad_SP_23_TCASE

MemoryLoad_SP_23

$dumping = no$
$mep = 0$
$ccmd = RM$
$low? = no$
$ped = 0$
$ctime = 0$
$mdp = 0$
$data? = \langle mdata0, mdata1 \rangle$
$memp = \emptyset$
$acquiring = no$
$waiting = yes$
$memd = \langle mdata0 \rangle$
$waitsignal = no$
$addr? = 0$
$sending = no$
$mode = COF$

MemoryLoad_SP_24_TCASE

MemoryLoad_SP_24

$dumping = no$
$mep = 0$
$ccmd = RM$
$low? = no$
$ped = 0$
$ctime = 0$
$mdp = 0$
$data? = \langle mdata0, mdata1 \rangle$
$memp = \emptyset$
$acquiring = no$
$waiting = yes$
$memd = \langle mdata0, mdata1 \rangle$
$waitsignal = no$
$addr? = 0$
$sending = no$
$mode = COF$

MemoryLoad_SP_25_TCASE

MemoryLoad_SP_25

$dumping = no$
$mep = 0$
$ccmd = RM$
$low? = no$
$ped = 0$
$ctime = 0$
$mdp = 0$
$data? = \emptyset$
$memp = \emptyset$
$acquiring = no$
$waiting = no$
$memd = \emptyset$
$waitsignal = no$
$addr? = 0$
$sending = no$
$mode = COF$

MemoryLoad_SP_26_TCASE

MemoryLoad_SP_26

$dumping = no$
$mep = 0$
$ccmd = RM$
$low? = no$
$ped = 0$
$ctime = 0$
$mdp = 0$
$data? = \langle mdata0 \rangle$
$memp = \emptyset$
$acquiring = no$
$waiting = no$
$memd = \emptyset$
$waitsignal = no$
$addr? = 0$
$sending = no$
$mode = COF$

MemoryLoad_SP_27_TCASE

MemoryLoad_SP_27

$dumping = no$
$mep = 0$
$ccmd = RM$
$low? = no$
$ped = 0$
$ctime = 0$
$mdp = 0$
$data? = \emptyset$
$memp = \emptyset$
$acquiring = no$
$waiting = no$
$memd = \langle mdata0 \rangle$
$waitsignal = no$
$addr? = 0$
$sending = no$
$mode = COF$

MemoryLoad_SP_28_TCASE

MemoryLoad_SP_28

$dumping = no$
$mep = 0$
$ccmd = RM$
$low? = no$
$ped = 0$
$ctime = 0$
$mdp = 0$
$data? = \langle mdata0 \rangle$
$memp = \emptyset$
$acquiring = no$
$waiting = no$
$memd = \langle mdata0 \rangle$
$waitsignal = no$
$addr? = 1$
$sending = no$
$mode = COF$

MemoryLoad_SP_29_TCASE

MemoryLoad_SP_29

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

MemoryLoad_SP_30_TCASE

MemoryLoad_SP_30

$dumping = no$
$mep = 0$
$ccmd = RM$
$low? = no$
$ped = 0$
$ctime = 0$
$mdp = 0$
$data? = \langle mdata0 \rangle$
$memp = \emptyset$
$acquiring = no$
$waiting = no$
$memd = \langle mdata0 \rangle$
$waitsignal = no$
$addr? = 0$
$sending = no$
$mode = COF$

MemoryLoad_SP_31_TCASE

MemoryLoad_SP_31

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

MemoryLoad_SP_32_TCASE

MemoryLoad_SP_32

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

MemoryLoad_SP_33_TCASE

MemoryLoad_SP_33

$dumping = no$
$mep = 0$
$ccmd = RM$
$low? = no$
$ped = 0$
$ctime = 0$
$mdp = 0$
$data? = \emptyset$
$memp = \emptyset$
$acquiring = no$
$waiting = no$
$memd = \emptyset$
$waitsignal = no$
$addr? = 0$
$sending = no$
$mode = COF$

MemoryLoad_SP_34_TCASE

MemoryLoad_SP_34

$dumping = no$
$mep = 0$
$ccmd = RM$
$low? = no$
$ped = 0$
$ctime = 0$
$mdp = 0$
$data? = \langle mdata0 \rangle$
$memp = \emptyset$
$acquiring = no$
$waiting = no$
$memd = \emptyset$
$waitsignal = no$
$addr? = 0$
$sending = no$
$mode = COF$

MemoryLoad_SP_35_TCASE

MemoryLoad_SP_35

$dumping = no$
$mep = 0$
$ccmd = RM$
$low? = no$
$ped = 0$
$ctime = 0$
$mdp = 0$
$data? = \emptyset$
$memp = \emptyset$
$acquiring = no$
$waiting = no$
$memd = \langle mdata0 \rangle$
$waitsignal = no$
$addr? = 0$
$sending = no$
$mode = COF$

MemoryLoad_SP_38_TCASE

MemoryLoad_SP_38

$dumping = no$
$mep = 0$
$ccmd = RM$
$low? = no$
$ped = 0$
$ctime = 0$
$mdp = 0$
$data? = \langle mdata0 \rangle$
$memp = \emptyset$
$acquiring = no$
$waiting = no$
$memd = \langle mdata0 \rangle$
$waitsignal = no$
$addr? = 0$
$sending = no$
$mode = COF$

MemoryLoad_SP_39_TCASE

MemoryLoad_SP_39

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

MemoryLoad_SP_40_TCASE

MemoryLoad_SP_40

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

MemoryLoad_SP_41_TCASE

MemoryLoad_SP_41

$dumping = no$
$mep = 0$
$ccmd = RM$
$low? = no$
$ped = 0$
$ctime = 0$
$mdp = 0$
$data? = \emptyset$
$memp = \emptyset$
$acquiring = no$
$waiting = no$
$memd = \emptyset$
$waitsignal = no$
$addr? = 0$
$sending = no$
$mode = COF$

MemoryLoad_SP_43_TCASE

MemoryLoad_SP_43

$dumping = no$
$mep = 0$
$ccmd = RM$
$low? = no$
$ped = 0$
$ctime = 0$
$mdp = 0$
$data? = \emptyset$
$memp = \emptyset$
$acquiring = no$
$waiting = no$
$memd = \langle mdata0 \rangle$
$waitsignal = no$
$addr? = 0$
$sending = no$
$mode = COF$

Bibliografía

[1] C. Ghezzi, M. Jazayeri, y D. Mandrioli, Fundamentals of Software Engineering, Prentice Hall, Upper Saddle River, 1991.

[2] I. Sommerville, Software Engineering, Addison-Wesley, Harlow, 1995.

[3] P.A.V. Hall, *"Towards testing with respect to formal specifications"*, Second IEEE/BCS Conf. Software Eng. 88. pp. 159-163, July 1988.

[4] P.A.V. Hall, *"Relationship between specifications and testing"*, Information and Software Technology, vol. 33, no. 1, pp. 47-52, 1991.

[5] J. Dick y A. Faivre, *"Automating the generating and sequencing of test cases from model-based specification"*, FME '93: Industrial Strength Formal Methods, Fifth Int'l Symp. Formal Methods Europe, J.C.P. Woodcock and P.G.Larsen, eds., Odense, Denmark, vol. 670 of Lecture Notes in Computer Science, pp. 268-284. Springer-Verlag, April 1992.

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

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

[8] P. Stocks y D. Carrington, *"A framework for specification-based testing"*, IEEE Transactions on Software Engineering, vol. 22, no. 11, pp. 777-793, November 1996.

[9] Practical Model-based Testing: A Tools Approach, Mark Utting and Bruno Legeard, Morgan-Kaufmann 2006.

[10] J.M. Spivey, The Z Notation: A Reference Manual, Series in Computer Science. Prentice Hall Int'l, second edition, 1992.

[11] Maximiliano Cristiá, Especificación Z de parte de los sistemas de un banco, 2007.

[12] Maximiliano Cristiá, Testing funcional basado en Especificaciones Z, 2007.

[13] S. Souza, J. Maldonado, S. Fabbri, and P.Masiero, "Statecharts specifications: A family of coverage testing criteria," in *CLEI'2000 - XXVI Latin-American Conference of Informatics*. CLEI, 2000.

[14] J.Peleska and M.Sieguel, "Test automation of safety-critical reactive systems," *South African Computer journal*, vol. 19, pp. 53-77, 1997.

[15] UML (Unified Modelling Language). `http://www.uml.org`

[16] Conformiq. `http://www.conformiq.org`

[17] Verifysoft. `http://www.verifysoft.com/en_qtronic_testautomation.html`

[18] Linux Verification Project. `http://linuxtesting.org/`

[19] UniTESK, industrial technology of reliable testing. `http://www.unitesk.com`

[20] Smartesting. `http://www.smartesting.com`

[21] Rave (T-VEC's Requirements-based Automated Verification). `http://www.t-vec.com/solutions/rave.php`

[22] T-VEC Tester for Simulink and Stateflow. `http://www.t-vec.com/solutions/simulink.php`

[23] Reactis. `http://www.reactive-systems.com`

[24] Telelogic Statemate & Telelogic Rhapsody. `http://modeling.telelogic.com/products/`

[25] Model JUnit. `http://www.cs.waikato.ac.nz/ marku/mbt/modeljunit/`

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

[27] A. Berson, Client/Server Architecture, McGraw-Hill, 1992.

[28] D. Garlan, G. Kaiser, D. Notkin, *"Using tool abstraction to compose systems"*, IEEE Computer, 25(6):30-38, june 1992.

[29] L. Bass, P. Clements, R. Kazman, Software architecture in practice, 2da edición, Addison-Wesley, 2003.

[30] Sun Microsystems, 2008. `http://java.sun.com/docs/books/tutorial/`

[31] Petra Malik and Mark Utting. *"CZT: A Framework for Z Tools"*. `http://www.cs.waikato.ac.nz/ marku/papers/zb2005_czt.pdf`

[32] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Patrones de diseño, Addison-Wesley, 2003.

[33] Martin, A.C.: Acyclic visitor. In Martin, R.C., Riehle, D., Buschmann, F., eds.: *Pattern Languages of Program Design 3*, Addison-Wesley Longman Publishing Co., Inc. (1997)

[34] Nordberg III, M.E.: Default and extrinsic visitor. In Martin, R.C., Riehle, D., Buschmann, F., eds.: *Pattern Languages of Program Design 3*, Addison-Wesley Longman Publishing Co., Inc. (1997)

[35] ISO/IEC 13568: Information Technology—Z Formal Specification Notation— Syntax, Type System and Semantics. First edn. ISO/IEC (2002)

[36] EXP–OBDH Communication Protocol Definition - A case study for PLAVIS. MINISTÉRIO DA CIÊNCIA E TECNOLOGIA - INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS - BRASIL.