Testing de Unidad de Programas Concurrentes

Tesina de grado presentada por

Duilio Javier Protti
P-2618/2

al
Departamento de Ciencias de la Computación
en cumplimiento parcial de los requerimientos
para la obtención del grado de

Licenciado en Ciencias de la Computación

Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario
Av. Pellegrini 250, Rosario, República Argentina

17 de Abril de 2007

Director

Prof. MSc. Maximiliano Cristiá

Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario
mcristia@fceia.unr.edu.ar

RESUMEN

La construcción de software, por ser una tarea humana, está sujeta a errores. En el ámbito de la computación se ha atacado esta dificultad desde distintos ángulos. Uno de ellos es el de realizar experimentos sobre el software una vez construído, para determinar si los resultados del experimento son los esperados. Este tipo de enfoque es lo que se ha dado en llamar testing de programas, y existe para ello una variedad de técnicas diferentes, las que se han desarrollado históricamente tomando como base programas secuenciales. Debido a ello, una de las presunciones sobre la que se basan es el de la reproducibilidad de los experimentos. Sin embargo, esta característica puede estar ausente o puede requerir una reinterpretación cuando se trata con programas concurrentes, pues éstos poseen un no-determinismo inherente.

En épocas recientes se han desarrollado técnicas de testing específicas para programas concurrentes, casi todas ellas basadas en un mismo mecanismo conceptual llamado de *record/replay*, donde en una primera ejecución de un caso de prueba se guarda información de scheduling que luego será utilizada para reproducir esa primera ejecución. De esta manera se puede realizar una ejecución determinista de los casos de prueba, obteniendo así la característica de reproducibilidad requerida por el proceso de testing.

Sin embargo, este tipo de mecanismo conceptual tiene como clara falencia el hecho de que los experimentos de testing de un programa concurrente se repiten siguiendo el scheduling original "guardado" en la primera ejecución del caso de prueba, pero no se experimenta con esos mismos casos de prueba ejecutados en un scheduling diferente del original.

En este trabajo se hace notar esta falencia y se desarrolla un entorno de Testing de Unidad de programas en lenguaje C que permite ejecutar cada caso de prueba en todas las intercalaciones posibles de sus sentencias atómicas. Qué constituye exactamente una sentencia atómica no es algo predeterminado por el entorno, sino que es el propio desarrollador quien tiene la libertad de delimitar explícitamente las fronteras de las acciones atómicas, lo que otorga un manejo muy flexible del grado de granularidad que se desea para el comportamiento concurrente.

ÍNDICE GENERAL

AGRADECIMIENTOS

Al comenzar mis estudios de licenciatura, me ví motivado por el desafío intelectual que planteaba la nueva ciencia de la computación, por las posibilidades profesionales que presentaba justamente por esa característica de "área en gestación", y finalmente por el simple hecho de que de pequeño me sentí atraído hacia el mundo de las computadoras, en una época en que las mismas aún no eran omnipresentes.

Siete años después he obtenido la licenciatura y veo como todas las expectativas que tenía al comenzar la carrera han sido satisfechas por completo, y mejor aún, he obtenido más de lo que esperaba. Es por eso que deseo en esta página agradecer brevemente a algunos de los que me ayudaron a obtener esa satisfacción.

Agradezco a mi pequeña familia por la paciencia, por la ayuda y por el sacrificio que han soportado a lo largo de tantos años: a mi esposa Laura y a mis hijos Joaquín y Julia. Sin duda ellos tres fueron la motivación que me llevó a no perder el rumbo y a enfocarme en el objetivo final durante los momentos de adversidad.

También mis padres han hecho un enorme esfuerzo, sosteniéndome económicamente durante tanto tiempo, y sin imponer restricciones de ningún tipo sobre mis elecciones académicas o profesionales. Siempre alentaron mi educación, desde sus primeros años, y me apoyaron en las decisiones importantes de mi vida. Por todo eso, mi eterno agradecimiento.

Por último agradezco a todo el ambiente académico de la Facultad de Ciencias Exactas e Ingeniería de la UNR: a mi director Maximiliano Cristiá, a todos los compañeros de estudios de estos años, a todos los profesores. Sería muy extenso nombrar a todos, pero mencionaré brevemente el espíritu que quiero rescatar: el de un conjunto de personas siempre dispuestas a atender una pregunta y a reflexionar sobre ella, sobre el tema que sea y en el lugar que sea, un aula, un pasillo o una mesa en un bar.

A todos los que me ayudaron a obtener este logro, muchas gracias.

1. INTRODUCCIÓN

El testing ha sido utilizado ampliamente desde los inicios de la informática como una herramienta de apoyo a la verificación de programas [23][38][31][30]. Se ha realizado mucha investigación en el área, pero gran parte de ella ha tenido como hipótesis implícita que se estaba tratando sobre programas secuenciales [28][21][39][33][40]. Sin embargo, los programas concurrentes plantean nuevos desafíos al proceso de testing, debido a la característica no-determinista de los mismos [11].

Por ejemplo, es difícil implementar testing de regresión [22] sobre programas concurrentes, debido a que no existen garantías de que en dos ejecuciones de un mismo programa con los mismos datos de entrada se ejecutará el mismo conjunto de sentencias y en el mismo orden. Tampoco son adecuados los criterios convencionales de extracción de casos de prueba basados en la estructura del programa, ya que por ejemplo los criterios tradicionales de cubrimiento de sentencias fueron diseñados presuponiendo un comportamiento secuencial del programa [32][19][44].

Esto ha sido en parte subsanado y actualmente se dispone de técnicas especiales para extracción de casos de prueba para programas concurrentes [16][49][18], así como de mecanismos de ejecución determinista de los mismos [34][27]. Estos mecanismos están todos basados principalmente en una metodología de tipo *record/replay*, donde en una primera ejecución del caso de prueba se recolecta información de *scheduling*, para luego utilizarla en sucesivas ejecuciones que "repiten" el mismo scheduling, con lo cual es posible implementar, por ejemplo, testing de regresión y facilitar el proceso de *debugging*. Existen implementaciones de estos mecanismos para lenguajes particulares, para entornos de ejecución como la JVM[1] e incluso para soporte a nivel de sistema operativo [7][37][36].

Sin embargo, hay dos grandes problemas que la literatura sobre testing de programas concurrentes no ha notado aún. El primero es que se asume que para poder realizar testing de tipo cíclico en un programa no-determinista, la *única* solución posible es utilizar mecanismos de tipo record/replay. Esto es dicho explícitamente [34] así como apoyado implícitamente por las publica-

[1] Java Virtual Machine

ciones en el área: todos los trabajos sobre ejecución de casos de prueba para programas concurrentes se centran en mejorar la eficacia de alguna variación de un mecanismo de record/replay (por ejemplo, buscando la forma de minimizar la cantidad de información a recolectar [27]), sin siquiera plantear la posibilidad de que exista una solución de otro tipo.

El segundo problema es que estos mecanismos de record/replay no dan una gran confianza respecto de la correctitud del programa concurrente, ya que ejecutan los casos de prueba para una sola intercalación de entre todas las posibles del programa (la correspondiente a la primera ejecución, en la cual se hizo el record), pero el programa, para ser correcto, debe ejecutarse correctamente **en todas las intercalaciones posibles**. Como se dijo antes, la investigación hasta aquí se ha enfocado más en tratar de mejorar los mecanismos de record/replay que en investigar la adecuación y grado de confianza que dichos métodos ofrecen para encontrar defectos en programas concurrentes.

Por esto, los objetivos fijados para el presente trabajo fueron:

- Hacer notar las falencias mencionadas, es decir, la poca confianza que brindan los mecanismos de record/replay, y mostrar que existen soluciones alternativas. Hasta donde sabe este autor, nadie en el área había alertado sobre estos problemas.

- Brindar un mecanismo de ejecución de casos de prueba para testing de unidad de programas concurrentes escritos en lenguaje C que permita ejecutar un caso de prueba en todas las intercalaciones posibles o en un conjunto de intercalaciones dado.

- Dar un método de extracción de las intercalaciones relevantes a los fines de verificar una determinada propiedad, partiendo solo de una especificación.

Si bien el último objetivo no pudo ser alcanzado, fue explorado en profundidad y se brinda un análisis de por qué no fue posible desarrollar un método así en el tiempo estipulado y cuáles son las dificultades que se deben sortear.

En cuanto al segundo objetivo, se implementó un mecanismo de ejecución de casos de prueba llamado CUThread, y se hizo de una manera que es independiente del sistema operativo y del modelo de *threads* subyacente. CUThread requiere disponer solamente de un compilador C estándar, ya que el sistema de scheduling va entrelazado en la propia estructura de los casos de prueba. Esto es posible gracias al uso de corrutinas en C [41][14] que utilizan para su funcionamiento una característica portable pero muy poco conocida del lenguaje [13]. Se pretende que el producto final del desarrollo sea

incluído como parte del sistema preexistente CUnit[2] para testing de unidad de programas C.

El desarrollo del informe continuará a lo largo de cuatro capítulos. En el capítulo 2 se hará un repaso al proceso de testing tradicional de programas secuenciales convencionales. Este repaso no pretenderá ser exhaustivo sino que consistirá en ver a vuelo de pájaro las técnicas y clasificaciones utilizadas en el área, añadiendo en todo momento comentarios respecto a lo adecuadas que pueden ser estas técnicas tradicionales con respecto al testing de programas concurrentes.

En el capítulo 3 se introducirá de lleno en el testing concurrente. Primero se aclarará cuáles serán los términos referentes a concurrencia utilizados en el resto del informe, pasando luego a analizar las deficiencias que muestran los métodos tradicionales de testing a la hora de someter a prueba a programas concurrentes. Se finalizará presentando algunas técnicas existentes que tratan de resolver estas deficiencias.

Después se llegará al punto central del informe, que es el capítulo 4 en donde se presenta concretamente el entorno desarrollado como parte de esta tesina. Se explicará en detalle el funcionamiento de CUThread como así también el sistema de corrutinas que lo soporta. Además se comentará acerca de las dificultades encontradas durante el desarrollo y sobre las distintas alternativas que se consideraron durante el curso de la misma.

Se finalizará con un capítulo conteniendo las conclusiones derivadas de la investigación desarrollada en esta tesina, así como sugerencias para trabajos futuros sobre el tema.

[2] http://cunit.sourceforge.net

2. TESTING TRADICIONAL

> El testing es verificación por experimentación.
>
> Ghezzi, Mandrioli y Jazayeri
> *Fundamentals of Software Engineering*

2.1. Introducción

El presente capítulo presentará un repaso muy breve de algunos aspectos del testing de programas. Para ello se explicará primeramente cual es el problema que se desea resolver, que es el de la verificación de software, para luego pasar a analizar cómo el testing se presenta como una posible solución a ese problema.

Luego se verán algunos aspectos más prácticos de la instauración de un proceso de testing, es decir sobre cómo se lleva adelante. Se finalizará explicando el caso puntual del Testing de Unidad.

2.2. Validación y verificación de software

En general, validación y verificación incluirá a todas las actividades que se emprendan para asegurar que el software cumple sus objetivos. La distinción entre ambos conceptos proviene de que responden a inquietudes distintas:

Validación ¿Estamos fabricando el producto correcto?

Verificación ¿Estamos fabricando correctamente el producto?

En general se deseará que a la validación la dirija el usuario y a la verificación el encargado de la parte técnica del desarrollo del software. Cabe aclarar que algunos autores consideran que estos términos son utilizados con muchos significados diferentes y de maneras inconsistentes [23]. Sin embargo, en este trabajo se considerarán ambos términos de la manera descripta arriba, es decir, como tendientes a responder cada uno la pregunta mencionada.

Fallas de software. También algunos autores distinguen entre distintos tipos de fallas en base a una categorización dada por cuestiones como de dónde proviene la falla, qué afecta, la observabilidad de la misma, lo necesario para solucionarla u otras cuestiones. Por ejemplo, Huselius distingue entre *faults* (faltas), *errors* (errores) y *failures* (fallas)[20]. Sin embargo, este tipo de categorizaciones no será tenida en cuenta en este trabajo, y se usará en cambio una noción simple de falla de software. Esto no significa que las distinciones mencionadas no sean relevantes, solo que no lo son a los motivos de este trabajo. Se utilizará la clasificación de tipos de defectos dada por Pfleeger [30], quien los divide en:

- algorítmicos.
- de precisión.
- de documentación.
- por estrés o sobrecarga.
- de capacidad o límites.
- de rendimiento o desempeño.
- de sincronización.

En base a esto se definirá la noción de defecto de software utilizada en este informe, como sigue:

DEFINICIÓN **1:** *En este trabajo se llamará* ***defecto de software*** *solo a aquellos del tipo de la primera categoría de la lista anterior, es decir que se considerarán solo defectos algorítmicos.*

Según Pfleeger los defectos algorítmicos se presentan cuando "el algoritmo o la lógica de un componente no producen la salida apropiada para una entrada dada, debido a que algo está mal en los pasos del procedimiento" [30].

Se podría argumentar que también se deberían tener en cuenta los defectos de la última categoría, la de sincronización, dado que ese tipo de problemas pueden estar asociados a una intercalación particular de un programa concurrente, con lo cual una definición así en apariencia sería útil. Lamentablemente esa categoría incluye otro tipo de defectos que son propios de los sistemas de tiempo real, los *timing-related errors*[20], por lo cual se decidió no incluírla en la definición.

Cualidades funcionales vs. no-funcionales. Se considera que incluso las cualidades no-funcionales o *implícitas* del software deben ser cubiertas por un proceso de validación y verificación [23]. Ejemplos de cualidades no-funcionales son la performance y la mantenibilidad. En [30] se propone que el testing puede ser usado para medir el grado en que una pieza de software posee alguna de estas cualidades. Sin embargo, aquí solo se utilizará al testing como forma de verificación de propiedades funcionales de los programas y no se considerarán las cualidades no-funcionales de los mismos.

2.2.1. Modelos de programas

Análisis de programas: estático vs. dinámico. El análisis de tipo estático explora las posibles ejecuciones del programa sin ejecutarlo realmente. Las técnicas de tipo dinámico (como el testing [1]) detectan defectos ejecutando el programa. Los dos enfoques son complementarios [21][23].

El análisis estático puede explorar todos los posibles caminos de ejecución, con lo cual puede encontrar defectos independientemente de los datos de entrada, por lo que puede llegar a ser más completo que el testing. Sin embargo también podría ser menos preciso que el testing, debido a que a veces puede identificar errores espúreos, es decir errores que están en el modelo pero que no corresponden a errores en el programa real representado por el modelo (para más detalles sobre "programa real" y "modelo de programa", ver la explicación que sigue a la definición 2).

Imposibilidad de una relación con un modelo completa y consistente. El análisis de programas generales es equivalente al *halting problem* de Turing. Es decir, es imposible hacer un programa que diga cuando cualquier programa dado ejecutará una acción errónea [21].

Esto causa que las herramientas de análisis de programas deban aproximar el comportamiento de un programa mediante un modelo que, en su relación con el programa real, debe optar entre completitud y consistencia[2] (dado que existe una tensión entre ambos). Para comprender lo que se está declarando, se definirán ambos tipos de relaciones.

[1] Pensando el testing como proceso. De lo contrario, se podría argumentar que la mayoría de técnicas de extracción de casos de prueba son estáticas. Sin embargo esas técnicas son solo una parte del proceso de testing, que es dinámico.

[2] Aunque existen herramientas que utilizan heurísticas que no son ni completas ni consistentes, pero aún así útiles para encontrar defectos.

DEFINICIÓN **2:** *Una relación entre un programa y un modelo de ese programa es* ***completa*** *si cada error identificado usando el modelo corresponde a un error en el programa real (no hay errores espúreos).*

Se considerará como *programa real* al objeto binario que es finalmente ejecutado por el procesador. Es decir, dada una especificación de un programa en, por ejemplo, lenguaje Z, esa especificación será un modelo (en Z) del programa. La implementación, por ejemplo en lenguaje C, de las operaciones especificadas, será un modelo (en C) del programa. Pero solo el binario final generado a partir de esa implementación C (por ejemplo un binario en formato ELF ejecutando en un sistema operativo Linux sobre un microprocesador MIPS 10000) será el programa real.

DEFINICIÓN **3:** *Una relación entre un programa y un modelo de ese programa es* ***consistente*** *si cada error en el programa real está representado por un error correspondiente en el modelo. En el modelo podrían aparecer más cosas, en general artefactos espúreos producto del análisis y que no existen en el programa.*

Notar que estas definiciones de completitud y consistencia no son idénticas a las dadas en [21], la cuales considero erróneas o al menos inadecuadas. Esas definiciones hablan de la consistencia y completitud del modelo, y no de la *relación* entre el modelo y el programa. Un modelo que fuera consistente pero cuya relación con el programa real no fuera consistente, no sería de gran utilidad práctica.

Entonces se debe recordar siempre esto: que en general las técnicas de análisis de programas están basadas en un *modelo* del programa y no en el programa en sí, con lo cual en algún punto se podrían estar abstrayendo elementos del programa que podrían ser cruciales [23]. Como se menciona al comienzo de este capítulo, el testing es verificación por experimentación (del programa real), con lo cual el testing en principio no sufriría de este problema[3]. Sin embargo, por esto mismo, el testing sufre de otros problemas, como por ejemplo que su generalidad no es siempre tan clara, ya que los experimentos están ligados al contexto en el cual fueron realizados.

2.3. Testing como forma de Verificación

Existen diversas técnicas para el descubrimiento de defectos de software. Por ejemplo:

[3] Aunque a veces no se pone a prueba a "el programa" sino a una parte del mismo

- Revisiones/inspecciones: las revisiones son básicamente re-lecturas de lo realizado, en general llevadas a cabo por alguien ajeno al grupo que lo hizo. Las inspecciones son revisiones pero donde existe una lista preestablecida de items a ser buscados específicamente durante el proceso de revisión. Ambas son buenas para encontrar defectos en el código y en el diseño. Según algunos autores, su efectividad a la hora de encontrar errores en el código es incluso superior a la del testing [30].

- Prototipos[4]: dado un sistema a producir, se crea de manera rápida y barata un modelo incompleto y focalizado solo en un subconjunto de los requerimientos. Este será un prototipo del sistema y servirá como un "modelo en funcionamiento" demostrable a los clientes, analistas de negocios y *managers*, quienes lo utilizarán para confirmar o hacer cambios en los requerimientos o en el diseño.

- Testing: técnica dinámica de descubrimiento de defectos, detallada a lo largo de este trabajo.

- Verificación formal de programas.

- Model Checking.

La enumeración anterior podría seguir, ya que existen otros tipos de técnicas. Se las menciona para aclarar que si bien el presente trabajo se enfocará solo en el testing, éste no es el único medio para encontrar errores en software, ni tampoco se insinúa que sea el mejor en comparación con otros enfoques. Por el contrario, en muchos casos estas técnicas son complementarias (por ejemplo, el model checking verifica el modelo pero no el programa real). Por otra parte, se debe tener en cuenta en todo momento que, como reza la famosa frase de Dijkstra, **el testing puede ser usado para mostrar la presencia de defectos, nunca para mostrar su ausencia**[23].

Además de mostrar la presencia de errores, el testing debe ayudar a *localizarlos*. Es decir que ante el hallazgo de un error, el proceso de testing debe brindar información útil sobre la localización del mismo, para que pueda ser utilizada por el proceso de *debugging*.

Otra cualidad deseable del testing, dada su característica de "experimentación", es la de que sea **repetible**, y he aquí el punto central del presente trabajo. El principal problema al que se enfrenta el testing de programas concurrentes es el de la reproducibilidad de la ejecución de los casos de prueba, a causa del no-determinismo inherente de este tipo de programas. Aunque

[4] Si bien en realidad los prototipos son una técnica de validación más que de verificación. Se agradece a Cristian Rosa por esta corrección.

hay que aclarar que la concurrencia no es la única forma en la que un caso de prueba puede resultar no-repetible: el problema podría surgir también en un programa secuencial tradicional, por ejemplo en uno en el que se utilizaran variables no inicializadas[5].

2.3.1. Un poco de formalización

Para explicar algunos conceptos, se dará una breve formalización, sin pretender que la misma sea exhaustiva. Simplemente se tomará una parte de la formalización dada en [23].

DEFINICIÓN **4:**

- *Se llamará* dominio *de un programa al producto cartesiano entre los tipos de datos de las variables de entrada. Por "tipo" se entenderá el significado común del término y no el de tipo de un lenguaje de programación, ya que un programa a ser verificado podría estar escrito en un lenguaje no tipado (y aún así ser testeable). Observar que la definición dada podría no ser la más adecuada para tratar con el testing de cierto tipo de software. Por ejemplo en GUI testing (sección 2.4.4), el concepto de "variables de entrada" puede no ser tan claro*[6].

- *Dado un programa P con dominio D, un* ***caso de prueba*** *para P será un elemento $d \in D$. Dado un programa P con dominio D un* ***conjunto de prueba*** *para P será cualquier subconjunto de D.*

- *Ahora, dado un programa P y un caso de prueba d para el mismo, se dirá que P es* ***correcto*** *en d si $P(d)$ es la salida esperada para P cuando ejecuta d. Observar que la comprobación de que "$P(d)$ es la salida esperada para P cuando ejecuta d" puede ser no trivial en ciertos contextos.*

- *Entonces, dado un programa P y un conjunto de prueba T para P, se dice que P es correcto en T si P es correcto para todo $d \in T$.*

[5] Aunque es motivo de debate si tal tipo de programas son deterministas o no. Se dirá más sobre esto al final de la próxima sección

[6] E incluso puede no ser claro en programas "tradicionales": algunos autores[34] consideran a las interrupciones como parte de la entrada, lo que dificulta enormemente algunos métodos, ya que para este tipo de entrada se debe tener en cuenta además el momento en la ejecución del programa en la cual esa entrada es "dada" al programa.

Nota. Respecto a la "salida esperada" del programa, se debe considerar a ésta como *todo* comportamiento exhibido por el programa que sea relevante a la comprobación de la correctitud del caso de prueba. En general, esta salida podría ser mucho más que la salida que el programa mostrará bajo uso en condiciones normales (no de testing). Por ejemplo, si para que un caso de prueba sea correcto se requiere que cierto evento o cómputo interno del programa ocurra antes que otro cierto evento, pero dichos eventos no producen ninguna salida en el programa final, esto no quiere decir que no se pueda verificar la correctitud del programa para el caso dado. En este caso la ocurrencia de tales eventos internos producirán una salida a nivel de operaciones internas, que será sobre la que se verificará la correctitud. O sea que la salida de la que se habla aquí es la salida de las operaciones *a nivel de la especificación* de las mismas, no a nivel de la implementación final del programa.

En resumen, 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 del sistema a desarrollar, como se analiza en la sección 2.4.2.

Otro aspecto a considerar es el del **estado inicial** de las variables del programa. Por ejemplo, si se hace un salto condicional basado en el valor de una variable sin inicializar, se podría preguntar ¿el programa es determinista o no? También es discutible si este valor "basura" es parte del programa P o es un valor de entrada que proviene "del entorno".

Ésta y otras cuestiones ponen de manifiesto la importancia de una especificación formal. De esta forma, se pueden hacer explícitas (y sin ambigüedad) las restricciones que se le impongan al estado inicial, de existir alguna, así como determinar exactamente cuales son variables de entrada y cuales son variables locales al programa.

2.3.2. Testing estructural vs. funcional

Estos son los dos grandes enfoques dentro del área de testing. El testing funcional o también llamado de caja negra (**black-box testing**) considera el

testeo de una pieza de software ignorando completamente cómo está construida internamente y mirando solo su especificación. Se puede pensar que se 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), quizá incluso ignorando 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) [23].

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.

En ambos enfoques, el proceso de testing puede automatizarse en gran medida. Para más detalles sobre estos dos enfoques, se puede consultar en este mismo capítulo la sección 2.4.1 sobre extracción de casos de prueba.

2.3.3. Testing no es debugging

El *debugging* está centrado fundamentalmente en dos aspectos: en la observación del estado computacional de un programa y en la necesidad de realizar modificaciones y controlar dicho cómputo con el fin de identificar y corregir defectos [11]. Es decir, testing y debugging no son lo mismo, pero es claro que están íntimamente relacionados, y más aún para el caso particular de programas concurrentes, como se verá más adelante al hablar sobre los problemas asociados a este tipo de programas (sección 3.3.1).

2.4. Testing de programas secuenciales

Una forma natural de verificar que algo funciona es simplemente ponerla en operación en algunas situaciones representativas y comprobar si su comportamiento es el esperado [23]. En general, será imposible realizar esto en todas las condiciones de operación posibles (testing exhaustivo) con lo cual es necesario encontrar casos de prueba que provean suficiente evidencia como para suponer de manera razonable que el comportamiento en todas las demás condiciones de operación será el requerido.

Aquí se presentan dos cuestiones:

1. por un lado, se necesitarán **criterios de selección de pruebas**, es decir estrategias para seleccionar casos de prueba *relevantes*.

2. la noción de caso de prueba "relevante", si bien no puede formalizarse, debe apoyarse en el concepto intuitivo de que, de existir un error en el programa, la ejecución del caso lo pondría de manifiesto.

La mayoría de los criterios manejan el segundo punto de la siguiente manera: para tratar de cubrir todos los casos relevantes sin la necesidad de ir al extremo de hacer testing exhaustivo, dividen el dominio de un programa en **clases de equivalencia**, D_i, con la esperanza de que el programa se comporte igual para todo $d \in D_i$. De esta manera, al momento de ejecutar los casos de prueba, se elige solo uno o algunos *representantes* de cada D_i para ser aplicado al programa P, disminuyendo la cantidad de casos a los que debe ser sometido el mismo.

Sin embargo, la idea de clases de equivalencia tal cual se definió arriba no es adecuada para testing de programas concurrentes. Para este tipo de programas, no se puede esperar que el comportamiento sea el mismo para todo $d \in D_i$ ya que ni siquiera se puede esperar que el comportamiento sea el mismo en dos ejecuciones de *un solo* d fijo, debido al no-determinismo del programa concurrente. Es decir, no habría una relación de equivalencia porque no se cumpliría la propiedad reflexiva. Pero esto tiene solución, por ejemplo redefiniendo la relación sobre pares (d, X), donde d es una entrada y X una intercalación posible del programa.

Un conjunto de prueba ideal sería uno tal que la ejecución del programa con los casos del conjunto, pondría de manifiesto todos los errores [32]. Sin embargo, descubrir tales conjuntos de prueba ideales es en general muy difícil. En la práctica, los casos de prueba son seleccionados de manera tal de aumentar la confianza en que los errores serán descubiertos, pero sin garantizar que una ejecución exitosa de todos los casos para un programa implique

la correctitud de ese programa. Esta confianza en general estará basada en el criterio con el cual se eligieron los casos de prueba, es decir, estará basada en la percepción que se tenga respecto de qué tan bien el criterio elegido "aproxima" la correctitud del programa [32].

Se procede de esta manera para reducir el problema general del testing a un subproblema: una vez elegido un "buen" criterio, el problema pasa a ser encontrar los datos de entrada y de estado inicial que cumplan el criterio.

Cuestiones sin resolver. Existen ciertas cuestiones aún sin resolver u *open-issues* dentro del área de testing, como por ejemplo la definición de criterios de detenimiento para el proceso de testing (¿hasta cuándo testear?) o de modelos para calcular el retorno de inversión del mismo [35]. En lo sucesivo no se volverá sobre estos temas, y en cambio se concentrará la atención solo en los aspectos técnicos del proceso de testing.

2.4.1. Cómo se generan/extraen casos

Extracción de casos en testing estructural

Para el testing estructural (2.3.2), se definen los llamados **criterios de cubrimiento**. Este tipo de criterios de selección de pruebas están motivados por la idea de que un error en un programa puede ser descubierto si la parte del programa que contiene el error es ejecutada y produce un funcionamiento incorrecto.

Tomando por "parte" en la definición anterior distintos artefactos del software, se pueden definir diferentes criterios. A grandes rasgos, existen dos grandes clases: basados en flujo de control y basados en flujo de datos.

Criterios de cubrimiento basados en flujo de control. Antes que nada, se define una representación del programa denominada *grafo de flujo de control* o GFC. Dicha definición depende del lenguaje en que esté escrito el programa original y es básicamente un grafo donde los nodos contienen sentencias del lenguaje y los arcos simbolizan los caminos que se siguen durante la ejecución, es decir que capturan las relaciones existentes en el orden de ejecución de las sentencias. Los arcos pueden estar etiquetados con expresiones lógicas que representan condiciones bajo las cuales el flujo de control sigue ese camino.

Ejemplos de este tipo de criterios son:

- Cubrimiento de sentencias: se elige un conjunto de prueba T tal que ejecutando el programa P para cada $d \in T$, cada sentencia elemental

de P es ejecutada al menos una vez. Qué es una sentencia elemental puede no ser siempre muy claro, por ejemplo en lenguajes funcionales, como se discute en la página 15.

- Cubrimiento de flechas: se elige un conjunto de prueba T tal que al ejecutar P para cada $d \in T$, cada flecha del GFC de P es atravesada al menos una vez.

- Cubrimiento de condiciones: se toma un conjunto de prueba T tal que al ejecutar P para cada $d \in T$, cada flecha del GFC de P es atravesada y todos los posibles valores de las proposiciones simples que componen las condiciones son probados al menos una vez. Es decir, cada proposición simple es hecha verdadera y falsa al menos una vez (pero no necesariamente en una misma condición las proposiciones simples son hechas verdaderas y falsas en todas las combinaciones posibles).

- Si en el criterio anterior se hace lo mencionado al final, es decir, hacer verdaderas y falsas las proposiciones simples en todas las combinaciones posibles para cada condición, se tiene otro criterio diferente.

Existen otros criterios basados en flujo de control. Además se han investigado las jerarquías de subsunción que los relacionan (es decir, cuándo un criterio contiene a otro o se intersecta con otro).

Criterios de cubrimiento basados en flujo de datos. Este tipo de criterios se derivaron originalmente de las técnicas de análisis de flujo de datos utilizadas para optimización de código en el área de compiladores. Están basados en asociar a cada cada punto de un programa en donde una variable es definida, un conjunto de puntos del programa en los cuales el valor de dicha variable es utilizado.

Así como en el enfoque mediante análisis de flujo de control no se podía tener gran confianza respecto de la correctitud del programa bajo prueba sin haber ejecutado cada sentencia del mismo al menos una vez para algún caso de prueba de un conjunto de prueba, en los criterios basados en flujo de datos el fundamento es que tampoco se puede tener gran confianza en la correctitud del programa sin haber observado el efecto de todos y cada uno de los valores producidos por los cómputos y cómo estos se propagan a lo largo de la ejecución [32].

Ejemplos de este tipo de criterios son los criterios **all-defs**, **all-p-uses**, **all-c-uses/some-p-uses** [7] y **all-du-paths**. Todos ellos se pueden consultar en [32].

Caso particular: testing de programas funcionales. Como se vió, en testing estructural la extracción de casos basada en flujo de control está dirigida por los llamados criterios de cubrimiento. Esto presenta una dificultad especial para el caso en que el programa esté escrito en un lenguaje funcional, ya que en ese caso no es muy claro cuál es la noción de *flujo de control*, debido fundamentalmente a la existencia de funciones de alto orden y, en algunos casos, de evaluación *lazy* [9].

Para este tipo de lenguajes, cualquier noción de flujo de control que se defina estará atada no solo al lenguaje en sí sino también al compilador y *runtime* con el cual se trabaje. Tampoco se simplifican las cosas al tratar de extraer casos mediante criterios de cubrimiento basados en flujo de datos, debido al polimorfismo y a que puede aparecer alto orden en los constructores de tipo. En resumen, el testing estructural de programas funcionales presenta dificultades propias.

Se menciona este caso particular para mostrar una cuestión que es general a todo el testing: que las técnicas, sean del tipo que sean, no son independientes de todo el resto del entorno de desarrollo. Es decir, al elegir una u otra técnica o al compararlas, siempre se deberá considerar el marco particular del proyecto en el cual se está trabajando.

Extracción de casos en testing funcional

Este tipo de testing está basado en una especificación del sistema. Y de ser posible, se querrá tener una especificación *formal* del mismo, ya que de esta manera existirán más chances de automatizar el proceso. La idea consiste en, dada solo la especificación de una operación del sistema, derivar a partir de ella un conjunto de casos de prueba para ser aplicados a la implementación de forma de comprobar si el sistema implementa correctamente su especificación.

Observar que los casos de prueba que se derivan de la especificación están expresados en términos propios de la especificación. Para que estos casos puedan efectivamente ser aplicados a la implementación, deberán ser **refinados** a la misma, es decir que los objetos que existen en la especificación deben ser mapeados a los correspondientes objetos en la implementación. Esta etapa de refinamiento es una de las que presenta mayores dificultades para la automatización de todo el proceso.

[7] p-uses y c-uses surge de la distinción que hacen las autoras entre las formas de usar un valor: computacionalmente o como parte de un predicado.

Una de las mayores ventajas del testing funcional es con respecto al mantenimiento y la modificabilidad: si el programa es modificado, gran parte del esfuerzo de testing sigue siendo útil, ya que por ejemplo los casos de prueba a nivel de la especificación seguirán siendo los mismos, pues estos solo cambiarán cuando cambie la especificación de alguna operación, pero son inmunes a los cambios en la implementación (*excepto en la parte concerniente al refinamiento*). Este poco acomplamiento con la implementación hace posible también que el testing funcional pueda comenzar a realizarse apenas comenzado el desarrollo, aún antes de que haya comenzado la etapa de implementación.

Existen una variedad de técnicas de testing funcional, como *Propagación de Subdominios* y *Mutación de Especificaciones*, que no se verán en este informe.

2.4.2. Cómo se evalúa el resultado de la prueba

El problema de determinar cuándo un test ha pasado o no una prueba es llamado el **problema del oráculo**. Una solución es comparar la salida del test con la salida de una versión anterior del mismo test. Otra es compararla con la de una versión más simple pero "obviamente correcta" [9].

Sobre la importancia de la especificación. Las dos soluciones mencionadas al problema del oráculo presentan la desventaja de tener un alto grado de subjetividad o de estar muy ligadas a la interpretación del desarrollador de las pruebas. Una solución alternativa más confiable es el uso de una especificación formal del artefacto de software a ser sometido a prueba por el test. En este caso, estaría especificada sin ambigüedad la salida esperada. Aunque esto es directamente cierto solo en el caso de que se cuente con una especificación *ejecutable*, es decir una donde la "salida esperada" esté expresada en términos de objetos del programa a ejecutar. De lo contrario, deberá haber un paso intermedio de *refinamiento* de la especificación a la implementación, proceso que no será en todos los casos automatizable y, aún cuando lo fuera por completo ¿qué garantías se tendrían de que dicho proceso es a su vez correcto y libre de defectos?

Lo único que se puede decir con certeza es que mediante el uso de una especificación, la complejidad se trasladó de un lugar del desarrollo a otro. Si se cuenta con una especificación no ejecutable, el proceso de refinamiento requerido podría ser fuente de errores, además de requerir tiempo de desarrollo. Si en cambio se decide hacer una especificación ejecutable, quizá se esté incurriendo en un error de diseño, pues la especificación podría resultar ser de mucho más bajo nivel de lo deseado.

Esto muestra que una especificación del sistema es solo una decisión más del desarrollo de una pieza de software, con una serie de consecuencias asociadas. Sin embargo se puede decir de forma general que, debido al alto grado en que ataca el problema de la subjetividad, una especificación será deseable en todo proyecto de desarrollo.

2.4.3. El testing en el ciclo de desarrollo de software

Es sabido que realizar verificación de correctitud solamente una vez que hay disponible código ejecutable torna muy difícil la reparación de los errores detectados. El costo de remover errores una vez que el software se ha completado es mucho más alto que el costo de remoción en etapas más tempranas del desarrollo [23]. Esto hace pensar que al utilizar al testing como forma de verificación de programas, se debería tratar de comenzar a aplicarlo lo más temprano que sea posible en el ciclo de desarrollo. Es decir que o bien se debería utilizar testing funcional o, si se utiliza testing estructural, se debería comenzar a hacerlo apenas se comiencen a construir partes aisladas del sistema. Esto último es una de las justificaciones de la existencia del Testing de Unidad.

Por otro lado, y en relación a lo mencionado sobre la evaluación del resultado de las pruebas ejecutadas sobre un programa, es de particular interés lo observado por Per Runeson en [35], donde como parte de un relevamiento sobre las prácticas de Testing de Unidad en la industria, se pidió a más de 50 compañías de software que completaran un formulario. Los datos así obtenidos mostraron que para esas empresas, la ejecución automática de casos de prueba era más importante que la comprobación automática de los resultados. Se volverá sobre esto hacia el final de la sección de Testing de Unidad (página 20).

2.4.4. Caso particular: GUI Testing

A pesar de que en algunas aplicaciones representa más del 50 % del código implantado total, el testing de interfases gráficas de usuario (GUI) no ha recibido mucha consideración dentro del área de testing hasta el presente [24]. Esto a pesar de que la verificación de tales artefactos de software presenta características propias y merecería una atención especial, y de que de acuerdo con la encuesta realizada a más de 50 empresas como parte del trabajo presentado en [35], la misma constata que el GUI testing representa para esas empresas el área más problemática dentro del proceso de testing, en particular debido a las dificultades para su automatización.

Otro problema es por ejemplo que los criterios de cubrimiento tradicionales no funcionan bien para GUI testing. Los criterios de cubrimiento basados en código no necesariamente ponen al descubierto los problemas que podrían surgir de las interacciones entre los eventos generados por el usuario y la aplicación. También, verificar cuándo una GUI ejecutó correctamente un caso de prueba plantea problemas particulares, debido a la componente en principio gráfica del entorno.

Tampoco se puede pensar siempre a la ejecución de casos de prueba en términos de "valores de entrada" y "valores de salida" al final de la misma, ya que un caso podría estar constituido por múltiples eventos de GUI que deberán ser pasados a la aplicación en distintos momentos.

Por otro lado, debido a que usualmente las implementaciones de GUI suelen utilizar algún tipo de despachador o manejador de eventos ejecutando en un thread aparte del de la aplicación principal (para evitar el posible bloqueo de la espera de tales eventos) se podría pensar en adaptar las técnicas de testing concurrente, que se presentarán más adelante, a la realización de GUI testing[8]. Sin embargo, debido a las características propias y únicas de éste último, considero que no es ello lo más adecuado, por lo cual aquí solo se analizarán técnicas de testing de programas concurrentes en general, sin pretender que tales métodos sean los más adecuados para realizar testing de GUI's o de otro tipo de programas concurrentes con características muy particulares.

2.5. Testing de Unidad

Para algunos autores, el Testing de Unidad comprende el testing de la menor unidad (módulo) *separable* de un sistema [35]. Debería ser separable pues se necesita ejecutar las operaciones de esa unidad de manera aislada del resto del sistema, donde "aislada" no significa que la unidad no utilice otras partes del sistema total, sino solo que esas partes podrían no estar implementadas completamente y aún así la unidad en cuestión debería poder ser ejecutada, ya que la unidad no debería utilizar esas otras partes del sistema de manera intrusiva, sino a través de una interfaz bien definida.

Al existir esta interfaz, cuando esos otros módulos se encontrasen aún sin implementar, ofrecerían una funcionalidad básica o interfaz mínima, sin hacer realmente todo el cómputo efectivo requerido para cumplir con su especificación. Este tipo de módulos son los que se llaman *stubs* [23]. Es justamente

[8] Como se menciona en [24], la mayoría de herramientas actuales de GUI testing utilizan mecanismos de *record/replay*.

por esta característica de "separabilidad" mediante módulos que esconden sus secretos detrás de una interfaz de la cual se tiene una especificación de su funcionalidad, que algunos autores sugieren la siguiente definición:

DEFINICIÓN **5:** *Se llama Testing de Unidad al proceso por el cual se someten a prueba a las menores unidades* especificadas *de un sistema [35].*

La definición anterior solo determina dónde debe ponerse la atención en el momento de ejecutar las pruebas, pero no obliga a que la unidad a probar deba estar literalmente aislada del sistema en el momento de la prueba. Es decir, el Testing de Unidad se haría focalizando la atención solo en la unidad bajo prueba e ignorando el resto del sistema [35].

Documentación. Del relevamiento mencionado en la sección 2.4.3 y descripto en [35], se desprende que la práctica común en la industria es que los casos de prueba para Testing de Unidad en general sean parte del mismo código fuente del programa y que no aparezcan en otra documentación. La explicación para esto, según el autor, es que los casos de prueba de unidad en general prueban características muy técnicas o cercanas a la implementación, y casi nunca prueban aspectos de la lógica de negocio de más alto nivel del sistema. Además esto se haría también para facilitar el trabajo de mantenimiento de los casos de prueba, ya que ahora se suma al costo de mantenimiento del código implementado, el código extra representado por los casos de prueba, que en general deberán ser modificados o al menos revisados ante cambios en el código fuente del programa.

Considero que esto no es totalmente correcto bajo la definición dada de Testing de Unidad, ya que si éste somete a prueba a las menores unidades especificadas de un sistema, entonces existiría una especificación, y por lo tanto los casos de prueba de unidad deberían estar articulados (en particular su documentación) con dicha especificación.

Peligros del "bajo nivel" del Testing de Unidad. Un punto relacionado al de la documentación es que el Testing de Unidad pretende verificar que un módulo tiene la funcionalidad que el desarrollador espera, que no es necesariamente igual a la funcionalidad esperada por otros participantes del desarrollo. Es decir que en la práctica actual, el Testing de Unidad es llevado a cabo y documentado en general por los propios desarrolladores, lo que tiene consecuencias positivas y negativas. Por un lado ese enfoque permite a los desarrolladores tener una rápida retroalimentación durante el proceso de testing, pero por otro lado se enfrentan a situaciones peligrosas, como por

ejemplo al hecho de que en general un programador diseña casos de prueba para un fragmento de código, con la misma lógica con la que programó ese fragmento, con lo que existe una alta propensión a que cometa los mismos errores, diseñando en consecuencia casos de prueba que no pondrían de manifiesto los defectos derivados de esos errores [25].

Lo anterior es apoyado por otros relevamientos de campo, como [35], que muestran que en la práctica actual en la industria los casos de prueba de unidad son en general escritos por los mismos desarrolladores de la unidad en cuestión, quienes conocen muy bien esa unidad, quizá demasiado bien. Es decir, conocen tan bien al módulo para el cual escriben los casos de prueba, que los mismos son escritos pensando en las internalidades del módulo, en lugar de ser escritos pensando en lo que le es requerido al módulo.

Por estas razones, el Testing de Unidad por sí solo no es suficiente como implantación de un proceso de testing, sino que siempre debería estar acompañado de los procesos de Testing de Integración y de Testing de Sistema correspondientes.

Integración con las herramientas de desarrollo. Por otro lado hay que mencionar que una característica muy deseable de todo entorno que brinde soporte al proceso de Testing de Unidad será el que dicho soporte esté coordinado con el sistema de construcción del software bajo desarrollo, como por ejemplo las herramientas de compilación y el sistema de control de versiones de los archivos fuente. Esto debido a que en general, ante un cambio en una componente en algún punto del proyecto, los desarrolladores desearán ejecutar solo los casos de prueba relevantes a las componentes alcanzadas por ese cambio, es decir, solo un subconjunto de todos los casos de prueba.

Si se tienen los casos de prueba de unidad vinculados de una manera ordenada al código implementado, un sistema de construcción que lleve la cuenta de los cambios en dicho código tendría la información necesaria para saber con exactitud qué casos de prueba se deben re-ejecutar luego de un cambio en el código, y de esta manera ejecutaría solo los pertinentes a ese cambio, ahorrándose de esta manera el repetir trabajo ya realizado[9]. Esto es deseable debido a que en la industria no es extraño encontrar software con conjuntos de prueba de unidad que toman horas para ser ejecutados [35].

Automatización. En una sección anterior sobre el proceso de testing en el ciclo de desarrollo de software (2.4.3), se mencionó que en una encuesta

[9] Esto es cierto solo en el caso de que la ejecución de los tests sea repetible. Como se verá cuando se trate testing de regresión (3.4.3), esto no es cierto en general para programas concurrentes.

realizada a más de 50 compañías, se observaba que éstas mencionaban como punto muy importante del proceso de Testing de Unidad el hecho de que la ejecución de los casos de prueba pudiera ser automatizable. La importancia de esto es inmediata a partir del obvio beneficio que brinda en cuanto a aprovechamiento de recursos, pero además porque provocaría que los desarrolladores se vieran más proclives a someter a prueba al sistema ante cada cambio en el mismo durante su desarrollo, ya que no les exigiría gran esfuerzo extra.

2.5.1. Testing de integración y testing de sistema

En la sección 2.3.2 se mencionan los dos grandes enfoques que existen para llevar a cabo el proceso de testing, el enfoque *black-box* y el enfoque *white-box*. En realidad eso es una verdad a medias, ya que ambos enfoques son de aplicación práctica en el proceso de testing de módulos individuales, es decir, son aplicables para hacer *Testing in the small* [23].

Pero al someter a prueba a un sistema de software completo, se requerirá realizar un 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.

Sin embargo, aún para el testing de un sistema completo, las técnicas para *Testing in the small* pueden seguir siendo útiles, si se maneja la complejidad de dicho proceso de manera similar a como se maneja el diseño o la implementación del mismo, es decir, separando el problema en partes. Por ello la actividad de testing debería de alguna manera reflejar la organización de la actividad de diseño. Por ejemplo, se podría utilizar la estructura modular del sistema como guía para dirigir el proceso de testing [23].

El presente trabajo no hace un aporte directo a las técnicas de testing de integración y de testing de sistema, y solo se concentra en el testing de unidad de programas concurrentes, aunque es cierto que podría interpretarse como un aporte indirecto a esas áreas, en el sentido mencionado en el párrafo anterior, o sea, cuando se lleve a cabo el *Testing in the large* mediante una división en módulos y subsecuente aplicación de las técnicas desarrolladas aquí a aquellos módulos que resulten concurrentes.

3. TESTING DE PROGRAMAS CONCURRENTES

El lector de un trabajo como el presente probablemente se encuentre interesado en temas relacionados con los programas concurrentes, con lo cual quizá conozca la relevancia que ésta tiene en los sistemas informáticos actuales. Aún así, se hará un breve repaso a las cuestiones que motivaron el nacimiento de este tipo de programas, para tener presente el marco histórico y los problemas a partir de los cuales se originaron estos programas.

3.1. Relevancia de la concurrencia

Recursos de tiempo compartido. Excepto para programas de cálculo intensivo como por ejemplo los usados para tareas científicas, un programa en general no mantendrá ocupado al 100 % el procesador físico por períodos de tiempo prolongados durante su ejecución. Es decir que por lo general existirán intervalos en los cuales un hilo determinado no estará haciendo uso del procesador (por ejemplo, podría estar haciendo `entrada/salida`), lo que motivó desde los inicios de la computación el desarrollo de sistemas que permitan *compartir* ese recurso ocioso con otras tareas en el mismo sistema. Como se mencionó, esta sincronización del acceso a un recurso compartido constituye la esencia de la concurrencia [45].

No-determinismo inherente. Otro problema que motivó el desarrollo de la concurrencia fue el de la falta de adecuación del paradigma secuencial para la clase de aplicaciones en las cuales el orden en que deberán ejecutarse ciertas operaciones se determina al momento de la ejecución en base a eventos controlados por un entorno que es externo al sistema. Por esta última razón, no sería posible predecir en qué orden ocurrirán dichos eventos, con lo cual el dominio de aplicación es inherentemente no-determinista [4].

Cómputo paralelo. Finalmente, aún para cálculos de cómputo intensivo que ocuparan el procesador físico al 100 %, podrían obtenerse en general

grandes ganancias de velocidad al utilizar varios procesadores físicos y asignar a cada uno de ellos diferentes procesos que trabajen sobre distintas partes del cómputo total.

Por las razones mencionadas anteriormente y por varias otras que pueden encontrarse en la industria de la computación, es que la programación concurrente en la actualidad ha adquirido relevancia dentro de la investigación en ciencias de la computación así como también en el desarrollo de sistemas comerciales de software.

3.2. ¿Qué es un programa concurrente?

A lo largo de la literatura sobre concurrencia, tanto en la academia como en la industria, se pueden encontrar ejemplos donde un término es utilizado para denotar distintas cosas, o donde distintos términos son utilizados para hacer referencia a la misma cosa, o términos que son utilizados de maneras contradictorias.

Esta sección no pretende aclarar ese panorama, ni cubrirlo completamente. Solo se hará mención a los términos relacionados con la concurrencia que se utilizan en este informe, aclarando cual es la definición que se toma como válida. Eso no quiere decir que esa definición sea la más adecuada, solo que es la más adecuada *para describir los conceptos de testing* que se desarrollan en el presente trabajo. Se hará mención a las confusiones más comunes para cada término si las hubiera, y en el caso de usos contradictorios, una pequeña justificación de por qué se eligió tal o cual uso del término y no otro.

3.2.1. Monoprocesador vs. Multiprocesador

DEFINICIÓN **6:** *Se llamará monoprocesador a una computadora con una única unidad de ejecución (procesador) que no ejecuta varias instrucciones simultáneamente* ***desde el punto de vista del programador****.*

Ejemplos: un procesador de propósito general con soporte de Hyper-Threading[1] o con procesadores vectoriales o con coprocesadores de punto flotante serán todos considerados monoprocesadores, siguiendo la línea de [5], siempre que el programador escriba sus programas sin tener en cuenta o sin manipular explícitamente esas unidades extra de ejecución.

[1] HyperThreading es una marca registrada de Intel Corporation.

DEFINICIÓN **7:** *Un multiprocesador será una computadora con más de un procesador compartiendo un conjunto de instrucciones común y con acceso a la misma memoria física (aunque quizá no con el* mismo tipo *de acceso, como por ejemplo en sistemas NUMA*[2]*).*

Estas simples definiciones bastarán a los fines de este informe.

3.2.2. Paralelismo vs. Concurrencia

La confusión entre estos términos proviene del idioma inglés. Según la definición inglesa del término *concurrency*, éste se refiere a cosas que ocurren al mismo tiempo [4]. Sin embargo, el término **concurrencia** en el área de software se refiere a cosas que *parecen* ocurrir al mismo tiempo. Cuando dos cosas ocurren al mismo tiempo en una computadora se dice que hay **paralelismo**.

Considerando lo anterior, se ve que el verdadero paralelismo solo puede ocurrir en sistemas multiprocesadores, mientras que la concurrencia puede aparecer en estos y también en monoprocesadores.

La concurrencia es una ilusión de paralelismo. Mientras que el paralelismo requiere que un programa sea capaz de realizar dos cómputos al mismo tiempo, la concurrencia solo requiere que el programador de ese programa pueda hacer de cuenta de que dos cosas ocurren al mismo tiempo [4].

A pesar de que algunos autores no hacen distinción entre ambos términos [47], aquí sí se hará, y se tomará en particular una noción de concurrencia como la recién citada.

DEFINICIÓN **8:** *Se llamará programa concurrente a cualquiera que permita que el programador "pueda hacer de cuenta" que dos o más hilos de ejecución*[3] *del mismo se ejecutarán al mismo tiempo, pero sin* ***ninguna*** *garantía a priori respecto del orden relativo en que se ejecutarán dos sentencias en dos hilos de ejecución distintos del programa.*

Se debe añadir a lo anterior que tanto la concurrencia como el paralelismo en general solo serán interesantes cuando exista algún recurso compartido entre dos o más hilos de ejecución, ya que se considera que la esencia de la

[2] Non Uniform Memory Access, un tipo de arquitectura para multiprocesadores.
[3] Se define hilo de ejecución en la página 27.

concurrencia es la sincronización del acceso a un recurso compartido [45]. Por ejemplo, si un programa consta solo de dos hilos de ejecución, y no existe ningún recurso compartido entre ellos, entonces cada hilo se puede analizar por separado como dos programas secuenciales.

Resumiendo, un programa podrá ser concurrente pero no paralelo, como ocurre en general en los monoprocesadores, o podrá ser ambos a la vez. Si se agregara a la definición de concurrencia el requisito de que exista al menos un recurso compartido, se tendría que también podría haber programas paralelos que no sean concurrentes (varios hilos que no comparten nada), pero ese tipo de programas para muchos autores carece de interés a los fines de la concurrencia. Aquí no se discutirá esto, y se tomará la definición tal cual está, ya que lo único que se quiere de ella es poder expresar lo siguiente: que el entorno creado en este trabajo, CUThread, solo maneja correctamente programas concurrentes no paralelos.

3.2.3. Concurrencia explícita vs. implícita

Estas son las dos formas en que se puede obtener un programa concurrente (además de que puede provenir de una mezcla de ambas).

La *concurrencia explícita* surge de expresar los distintos hilos de ejecución, puntos de sincronización, regiones críticas u otros elementos relevantes a la concurrencia, utilizando constructores específicos para cada uno de ellos. Estos constructores podrían ser parte del lenguaje en que está escrito el programa o podrían estar proporcionados por una biblioteca de uso compartido o como servicios del sistema operativo o máquina virtual subyacente.

La *concurrencia implícita* puede surgir por ejemplo como consecuencia de tomar un programa secuencial y pasarlo por un compilador paralelizante que identifique oportunidades para paralelizar y genere un programa paralelo semánticamente equivalente al original secuencial [47].

En CUThread solo se consideran programas con concurrencia explícita, y donde no solo están expresadas las construcciones concurrentes de sincronización sino que también están indicadas de manera explícita las sentencias atómicas.

3.2.4. No-determinismo

Se puede considerar que existen dos fuentes de no-determinismo en un programa [34]:

- No-determinismo externo: significa que el programa devuelve diferentes resultados para diferentes ejecuciones con los mismos datos de entrada. Este comportamiento puede ser un síntoma de un error en el programa o bien puede ser aceptable para el problema dado (por ejemplo, si el problema resuelto por el programa admite más de una solución).

- No-determinismo interno: significa que diferentes ejecuciones con los mismos datos de entrada producen el mismo resultado, pero el camino de ejecución interno es distinto para algunas ejecuciones.

Cabe aclarar que esta distinción interno/externo depende del nivel de abstracción al cual se observa el programa. Esto es de fundamental importancia en los mecanismos de *record/replay* que se verán más adelante.

Observar que el no-determinismo también está presente en muchos programas secuenciales, por ejemplo por medio de *interrupciones* asíncronas que alteran el flujo de ejecución. Este es un tipo de concurrencia implícita, que como se mencionó antes, no se considerarán en este trabajo.

En lo que sigue, cuando se hable de no-determinismo y a menos que se especifique lo contrario, se estará haciendo referencia a no-determinismo interno, ya que interesará analizar, para los mismos datos de entrada, los distintos caminos de ejecución posibles (de las sentencias atómicas, marcadas explícitamente).

3.2.5. Procesos, tareas, hilos de ejecución

Es confusa en la bibliografía la distinción entre tarea, hilo de ejecución (*thread*) y proceso [20]. La falta de claridad de estos conceptos proviene de que esos términos pueden interpretarse de distinta forma a distintos niveles (sistema operativo, lenguaje de programación, biblioteca, etc.).

La definición de concurrencia utilizada en este trabajo (definición 8) es independiente incluso de la existencia de un Sistema Operativo controlante de la computadora en la cual es ejecutado el programa concurrente en cuestión. Sin embargo, para intentar comprender los conceptos de proceso e hilo de ejecución, se debe repasar brevemente la evolución de los aspectos de acceso a recursos compartidos en los Sistemas Operativos.

Como ya se mencionó, la esencia de la concurrencia es la sincronización del acceso a un recurso compartido. En particular, en una computadora tradicional, cosas tales como los mecanismos de cómputo (registros de procesador, etc.) y de entrada/salida, así como la memoria, podrían ser compartidos o accesibles a cualquier secuencia de instrucciones ejecutada por la máquina.

Esto plantea un problema: una secuencia de instrucciones podría cometer un error y dejar al sistema en un estado que sería inconsistente para una secuencia de instrucciones posterior. Esta amenaza a la integridad como consecuencia del acceso ilimitado a sus recursos, podría provenir no solo de código premeditadamente malicioso, sino también de código "benigno" pero con algún defecto o que no manejase adecuamente una condición de error.

Para implementar **protección** frente a accesos dañinos o ilegítimos a los recursos de un sistema de cómputo, es que fueron creados los Sistemas Operativos (SO). En particular, este tipo de programas tienen lo que se llama comúnmente un *kernel* o núcleo, que es un programa distinguido del resto de programas que pueden ejecutar en el sistema, con las cualidades (idealmente únicas a él) de que su código siempre (eventualmente) se ejecuta durante todo el tiempo de funcionamiento de la computadora, y que dicha ejecución se lleva a cabo con un acceso *completo* a los recursos del sistema [10].

Es con el advenimiento de los SO que surgió la idea de **proceso**. En primer lugar, un SO define un *entorno de ejecución* para ser utilizado como unidad de administración de los recursos del sistema. Este entorno contiene básicamente:

- Un espacio de direcciones.
- Mecanismos de comunicación, como consolas y sockets.
- Otros recursos de alto nivel, como archivos abiertos y ventanas.

Con esto, se puede decir que

DEFINICIÓN **9:** *Un proceso en un SO será un solo hilo que lleva a cabo sus actividades dentro de un mismo entorno de ejecución.*

Inicialmente fue suficiente un solo hilo por proceso. Sin embargo, con el paso del tiempo se observó que cuando se querían realizar varias tareas, este esquema no era el mejor, ya que el compartir recursos entre distintos procesos (la sincronización entre distintos entornos de ejecución) significaba una sobrecarga computacional que era innecesariamente grande en algunos casos.

Es por esto que se extendió el mecanismo de procesos de manera que admitiese la coexistencia de más de un hilo de ejecución vinculado a un mismo proceso y un mismo entorno de ejecución. Así nacieron los sistemas multihilo. En estos sistemas, un programa se puede dividir o puede bifurcarse en varios

hilos de ejecución, donde cada uno ejecutará su propio conjunto de instrucciones (es decir que seguirá su propio flujo de ejecución) pero compartirá el entorno de proceso junto con todos los demás hilos[4].

Con este nuevo mecanismo el SO sigue manteniendo la protección entre distintos procesos y entornos de ejecución, pero ya no entre distintos hilos de un mismo proceso. Hay que agregar que además se añadieron a los mecanismos de comunicación del entorno de ejecución, mecanismos específicos para sincronización entre hilos (aunque no necesariamente los brinda el SO; pueden estar implementados en espacio de usuario).

En un sistema como el anterior, los distintos hilos pueden ejecutar en distintos procesadores de un multiprocesador, o bien las instrucciones de distintos hilos pueden intercalarse en el tiempo para dar lugar a que los diferentes hilos tengan oportunidad de ejecutar (de esta manera se implementan multihilos en un monoprocesador). Como se mencionó, el SO no mantiene la protección entre distintos hilos ejecutando en un mismo entorno de ejecución, con lo cual un programa multihilo debe estar particularmente bien implementado, pues un error en un hilo puede llevar a la falla de la ejecución de todos los demás hilos en el mismo entorno.

El presente trabajo brinda una herramienta de ayuda a la programación concurrente (multihilo) de programas en lenguaje C que es independiente de los mecanismos de sincronización particulares brindados por un entorno de ejecución.

3.2.6. Definición de intercalación

En primer lugar, se toma un concepto que proviene del área de Teoría de Compiladores.

DEFINICIÓN **10:** *Un **bloque básico** es una secuencia de sentencias tal que*

- *La última sentencia es un salto, condicional o incondicional.*
- *La primera sentencia es el objetivo de un salto.*
- *Ninguna otra sentencia en el bloque (excepto las anteriores) es ni un salto ni el objetivo de un salto.*

Es decir que en un bloque básico el control comienza en su primera sentencia y finaliza en la última. No hay saltos dentro del bloque, como tampoco se

[4] A los fines del presente informe, esta definición de hilo de ejecución es más que suficiente, y es la que se usará en lo sucesivo.

salta de algún otro lugar del programa hacia algún punto dentro del bloque, excepto quizá a su primera sentencia.

Ahora, considerando a ; como el operador de secuenciación y tomando `P`, `Q` y `R` como bloques básicos de sentencias en algún lenguaje de programación secuencial, si se escribe el siguiente programa:

```
P;
Q;
R
```

Se tiene que, sin importar lo que cada bloque básico haga, el fragmento anterior le está dando al sistema que ejecutase dicho programa precisas instrucciones acerca del orden en que las sentencias deben ejecutarse.

Se está diciendo que P *debe* preceder a `Q` y que `Q` *debe* preceder a `R`. Cualquier implementación del lenguaje secuencial en cuestión debe asegurar que lo anterior es cierto *para cada posible ejecución del programa* [4]. Antes de ver por qué esto podría interesar a los fines de este trabajo, una breve definición.

DEFINICIÓN **11:** *Una aserción en un lenguaje de programación es un predicado que involucra a los objetos de dicho lenguaje. Una aserción que es verdadera para cada posible ejecución de un programa se dice que es una* ***propiedad*** *de ese programa*[5].

Ahora se puede decir, más formalmente, que para el programa anterior se tiene la siguiente propiedad:

$$\forall e \bullet P \rightarrow Q \rightarrow R$$

donde el operador $\rightarrow$ significa "ocurre antes" y el símbolo $\forall e$ significa "para toda ejecución legítima del programa".

Lo esencial es que cuando se escribe $P \rightarrow Q$ se está indicando que P debe comenzar a ejecutar antes que comience Q, pero también se está diciendo que P deber *terminar* antes de que comience Q (si es que alguna vez termina). Es decir, no debe haber solapamiento en la ejecución de las instrucciones que componen P y Q.

[5] Por simplicidad, y porque será más útil a los fines del trabajo, la definición considera a los programas como si no tuvieran entrada. Sin embargo, la definición de propiedad se puede generalizar para sí tenerla en cuenta, pero lo que interesa aquí son solo las intercalaciones posibles de las sentencias atómicas.

Notar que de los tres bloques básicos anteriores, ninguno es necesariamente atómico. Cualquiera de los tres puede estar:

- Compuesto por varias sentencias del lenguaje de programación.
- Formado por una sentencia indivisible del lenguaje, como por ejemplo una asignación `x = 10`, pero aún así dicha sentencia podría ser representada por varias instrucciones a nivel de código máquina (*assembler*).

Es decir que una sentencia que es **atómica** a un nivel de abstracción, no necesariamente lo es a otro nivel.

Por ejemplo, si P y Q tienen como instrucciones *en código máquina* las secuencias $p1, p2, ..., pn$ y $q1, q2, ..., qm$ respectivamente, se puede escribir la propiedad anterior como:

$$\forall e \bullet p_1 \rightarrow p_2 \rightarrow \ldots \rightarrow p_n \rightarrow q_1 \rightarrow q_2 \rightarrow \ldots \rightarrow q_m$$

Ahora se verá que este tipo de propiedades no valen en general para programas concurrentes.

Un programa concurrente se puede ver como un orden parcial. Piénsese en los tres bloques básicos anteriores P, Q y R, y supóngase que se cuenta con tres procesadores físicos, con lo cual se puede ejecutar cada sentencia como un proceso propio a cada procesador. Supóngase también que se cuenta con una construcción en un lenguaje de programación para expresar tal acción:

```
P || Q || R
```

Es decir, lo anterior representaría la ejecución "en paralelo" de P, Q y R. Ahora bien, una pregunta inmediata es ¿qué pasa en el programa anterior cuando no se cuenta con tres procesadores?

La respuesta es **procesadores lógicos**. Cuando se escribe $P \| Q$ se dice que se están creando dos procesos, uno para ejecutar la sentencia P y otro para ejecutar la sentencia Q, y a cada uno de estos procesos se les asigna un procesador lógico. Cada procesador lógico puede ser visto a su vez como una máquina secuencial [4]. Luego, esos procesadores lógicos serán "mapeados" a procesadores físicos (que puede ser incluso uno solo, como cuando se implementa concurrencia en un monoprocesador).

En este último caso, que es el que nos interesa, lo que se hará para brindar la ilusión de paralelismo será *intercalar* las instrucciones atómicas de los distintos procesos.

DEFINICIÓN **12:** *Dado un bloque básico P con sentencias atómicas ordenadas $p1, p2, ..., pn$ y un bloque básico Q con sentencias atómicas ordenadas $q1, q2, ..., qm$, una* ***intercalación*** *entre P y Q será un programa secuencial cuyas sentencias atómicas resultarán de intercalar las sentencias atómicas de P y las de Q, manteniendo su orden relativo.*

Ejemplo: si $P = \{p1, p2, p3\}$ y $Q = \{q1, q2\}$, entonces $\{q1, p1, p2, q2, p3\}$ es una intercalación entre P y Q, mientras que $\{q1, p3, q2, p1, p2\}$ no lo es. Con esta nueva idea, se puede redefinir la noción de propiedad de un programa para el caso particular de programas concurrentes. Se puede decir que una propiedad es "una aserción que es verdadera para toda intercalación posible de los bloques básicos del programa".

Este es el acercamiento empleado en CUThread: se escriben casos de prueba que contienen aserciones acerca de una o más propiedades a verificar, y luego el entorno ejecuta el caso de prueba en todos las intercalaciones posibles de sus bloques básicos, comprobando en cada caso si las aserciones son verdaderas o no.

3.2.7. Modelos semánticos de concurrencia

No se debe confundir el uso de los términos "orden parcial" e "intercalación" de la sección anterior con el uso que se les da en la bibliografía para refererirse a diferentes modelos semánticos de concurrencia.

Primero, vale aclarar de dónde surgieron. Muchos autores argumentan que un sistema no debe ser representado solamente como una estructura de eventos (o como una red de Petri), pues esta representación no es lo suficientemente abstracta, y en cambio se debe representar mediante una *clase de equivalencia* entre esos objetos [43]. Si se acepta esto, la cuestión pasa a ser qué noción de equivalencia utilizar. Aquí no se verán esas nociones de equivalencia, pero sí se verá cómo el tipo de equivalencia elegida permite separar las distintas semánticas en distintas categorías, dependiendo del nivel de detalle que la clase de equivalencia en cuestión preserva acerca de los eventos ocurridos durante la ejecución. Con ello, las semánticas se pueden dividir en:

Semánticas de *interleaving* (intercalación): modelos en los cuales las ejecuciones son representadas por secuencias de acciones, abstrayéndose de los enlaces causales entre las mismas [43]. Los modelos de este tipo son utilizados para describir sistemas basados en acciones que se suponen que son indivisibles y atómicas.

Semánticas causales: modelos en los cuales las dependencias causales son representadas explícitamente. Un caso particular de éstas son las semánticas de orden parcial, donde dichas dependencias se representan como órdenes parciales.

Las nociones de equivalencia también pueden separarse de acuerdo al nivel de detalle que preservan respecto de las estructuras de decisión de las ejecuciones (esta categorización es ortogonal con la anterior):

Semánticas de trazas (o de *tiempo lineal*): Son la noción más simple de este tipo, en donde el sistema es determinado por el conjunto de todas sus ejecuciones (trazas) posibles.

Semánticas de bisimulación (o de *branching time*): donde se conserva información acerca de dónde dos ejecuciones divergen.

Semánticas de trazas decoradas: se ubican en medio de las dos anteriores. Son aquellas donde una parte de la estructura de decisión es preservada. La semántica de trazas+fallas+divergencias para CSP es un ejemplo de este tipo de semánticas.

Hay mucho para decir sobre este tema (para un resumen se puede consultar [43]), pero no se ahondará más en lo sucesivo.

A los motivos del presente trabajo resulta de interés mencionar las categorizaciones anteriores pues el entorno desarrollado aquí se basa, según la primera de las categorizaciones anteriores, en un modelo de intercalaciones, lo que trae como consecuencia el hecho de que la suposición de la atomicidad de las acciones se transforma en un impedimento clave a la hora de tratar de diseñar métodos para extraer (solo de la especificación) las intercalaciones relevantes para comprobar alguna propiedad de un caso de prueba [6].

La dificultad es básicamente que en los modelos con semántica de *interleaving* la noción de equivalencia no es invariante bajo *refinamiento* (de las acciones atómicas). Esto ha sido notado por Lamport y otros autores[43], quienes también han mostrado que en cambio las semánticas de orden parcial no adolecen de estos problemas, es decir, que en ellas la noción de equivalencia no depende de la granularidad de la atomicidad de las acciones.

En conclusión, debido al modelo elegido, no resulta nada simple diseñar un método de extracción de casos de prueba concurrentes mediante testing funcional, o sea a partir de la especificación, que era uno de los objetivos que se habían fijado para el presente desarrollo al presentarse el proyecto de

[6] Se dice más sobre esto en la sección 4.4.1.

tesina. Esto no quiere decir que no sea posible dar un método funcional, sino solo que debido a la dificultad mencionada, no fue posible desarrollarlo en el tiempo estipulado para la realización del presente trabajo.

3.3. Problemas de la concurrencia

Como ya se dijo (página 29), una propiedad de un programa es una aserción que es verdadera para *cada* ejecución legal de ese programa. Es generalmente aceptado que un programa concurrente correcto debe satisfacer propiedades de dos clases: de *safety* y de *liveness*. Informalmente:

Propiedades de *safety*: dicen que nunca algo "malo" ocurrirá durante la ejecución del programa.

Propiedades de *liveness*: dicen que algo "bueno" eventualmente ocurrirá durante la ejecución del programa.

En el segundo item, la palabra "eventualmente" denota "en algún momento", sin mayores precisiones de tiempo, como por ejemplo alguna restriccón de lapso mínimo ni nada parecido. Ese tipo de restricciones son importantes en algunos sistemas, como los de de tiempo real y otros, pero no se considerarán en éste trabajo.

Ambas propiedades no son independientes, sino que están relacionadas. Por ejemplo, para probar que algo "bueno" eventualmente sucederá, se deberá probar antes que nada "malo" ocurrirá en el camino [4].

La ausencia de *deadlock* y la ausencia de inconsistencia de actualización son ejemplos de propiedades de *safety*, mientras que la ausencia de livelock es un ejemplo de propiedad de liveness, como se verá a continuación.

Deadlock. Es un estado en el cual uno o más procesos quedan bloqueados esperando por algún evento que nunca ocurrirá. La detección de este tipo de problemas en tiempo de ejecución es en general automatizable si se identifican de antemano todas las operaciones bloqueantes. Esto permite que la técnica propuesta en éste trabajo, si se suma a un sistema de detección de *deadlocks* para un entorno dado, permita tener una mayor confianza respecto de esta propiedad de *safety* luego de realizado el proceso de *testing*.

Inconsistencia de actualización. Es una clase de problemas en los cuales un hilo de ejecución actualiza datos en base a una visión inconsistente de la memoria. Por ejemplo, si un primer hilo ejecutara

```
x := x + 1;
```

concurrentemente con otro hilo que también modificara la variable compartida `x`, podría ocurrir que el primer hilo tuviera una visión inconsistente del valor de `x`. Esto es así porque el código anterior podría dividirse, a nivel de código máquina, en más instrucciones de las que se suponen viendo el código original. Por ejemplo, en el caso de una asignación como la anterior, muchos procesadores recibirán *tres* instrucciones: una para traer a un registro del microprocesador el valor de la locación de memoria representada por `x`, otra para incrementar el valor de ese registro en uno, y una tercera para guardar el resultado nuevamente en la memoria en la locación dada por `x`.

Si el programa se ejecuta concurrentemente con otro, podría ocurrir que luego de la primera de las tres instrucciones a nivel de código máquina (es decir luego de leer el valor de `x`), se produjera un cambio de contexto y que el nuevo thread ejecutante cambiara el valor de `x` a otro distinto. En ese caso, en un nuevo cambio de contexto en el que se volviera al primer thread, éste continuaría con las intrucciones de incremento del valor de `x` y posterior guardado de dicho valor, pero haría este cálculo en base a un valor de `x` que ya no es el actual, pues fue cambiado por el otro thread.

El ejemplo es un caso muy simple en donde ni siquiera hay condiciones lógicas que involucren el valor de `x`, pero es fácil ver que este tipo de inconsistencias pueden producir defectos muy difíciles de detectar. En particular, observar que el error no es deducible a partir solamente del código fuente original del programa, sino que lo será a partir de una representación del mismo a más bajo nivel, representación que quizás no sea la misma en todas las plataformas (por ejemplo, puede variar de un procesador a otro).

Livelock. Ocurre cuando un hilo entra en un ciclo de instrucciones tales que no permiten continuar con los cómputos requeridos por la tarea que se quiere realizar. Dicho hilo sigue ejecutando instrucciones, pero esas instrucciones tienen solo un valor administrativo de sincronización y no se realiza trabajo verdadero que permita avanzar en el cómputo deseado [20].

Los tres anteriores son solo algunos ejemplos de los problemas que se encuentran en la programación concurrente. Existen otros tipos, como condiciones de carrera[7] y muchos más. Lo que se pretende establecer aquí es que los programas concurrentes presentan problemas que les son propios, es decir que la programación concurrente es una disciplina en sí misma dentro de la computación.

[7] *Race conditions.*

3.3.1. Costos asociados a la concurrencia

Sobrecarga de cómputo. Para sistemas concurrentes existe una penalización en tiempo de ejecución comparado con su contraparte secuencial, debido a que se debe utilizar parte del tiempo del procesador para realizar tareas de sincronización, que en general no contribuyen directamente al cómputo efectivo que está realizando el programa.

Proceso de *debugging* dificultoso. Esto es debido principalmente al no-determinismo inherente de un programa concurrente, pues como se mencionó en la definición de la página 24, no hay garantía a priori respecto del orden relativo en que se ejecutarán dos sentencias en dos hilos de ejecución diferentes.

También los problemas como la inconsistencia de actualización y similares, que dependen en gran medida de la plataforma en la cual se ejecuta el código objeto final del programa, presentan dificultades adicionales que solo aparecen en programación concurrente.

Testing y debugging. Como se dijo en el capítulo anterior, si bien testing y debugging no son lo mismo, están fuertemente relacionados. Esto es aún más notorio en el caso de programas concurrentes, ya que como el debugging está centrado en la observación y modificación del estado del programa con el fin de detectar y corregir errores, en presencia de concurrencia las herramientas de debugging deberán estar apoyadas por una herramienta de testing que identifique y aisle las intercalaciones sobre los cuales se hará debugging [11], pues como el proceso de debugging es guiado en general de manera manual, en programas grandes se vuelve impracticable ejercitar todos los caminos de ejecución posible. Además, el operador que guía el proceso de debugging querrá asegurarse que un cambio de comportamiento en el programa se debe a las modificaciones introducidas por la herramienta de debugging y no a que el programa está ejecutando en una intercalación diferente.

Esta necesidad de una interfase entre las herramientas de debugging y las de testing añaden un poco más de dificultad al proceso de debugging de programas concurrentes.

En conclusión, un sistema concurrente no debe ser *siempre* la primera opción, sino que debe elegirse solo cuando un estudio previo del problema y del diseño de su solución permita ver que la ganancia que se obtendrá al implementarse mediante un programa concurrente volverá poco significante el "costo administrativo" de la sincronización requerida, así como también se

deberá estar dispuesto a afrontar los desafíos extra que presenta el desarrollo de este tipo de sistemas.

3.4. Falta de adecuación de los criterios tradicionales de testing

En lenguajes tradicionales como Pascal, C, Fortran, etc., un programa se puede ver como una función del conjunto de posibles datos de entrada al conjunto de posibles datos de salida. Esto significa que si se proporciona la misma entrada dos veces a un programa, el mismo produce los mismos resultados en las dos ocasiones. Esta propiedad no vale en el caso de sistemas concurrentes y de tiempo real [23][sección 6.3.7]. Además, cosas que antes se abstraían, ahora en este tipo de sistemas pasan a ser relevantes (políticas de scheduling, arquitectura subyacente, etc.).

3.4.1. Para obtener casos de prueba

Como se explicó en la sección 2.4.1, la obtención de casos de prueba mediante criterios de cubrimiento se basan en la idea de que un error en un programa puede ser descubierto si la parte del programa que contiene el error es ejecutada y produce un funcionamiento incorrecto. Sin embargo, para un programa concurrente erróneo, no basta solo con ejecutar la parte que contiene el error sino que además se debe llegar a ese punto con una intercalación que exhiba el comportamiento incorrecto. Pero esta intercalación podría ser una entre un número inmenso de intercalaciones posibles para el programa concurrente.

Es por esto que los métodos tradicionales de testing estructural usados para extraer casos de prueba no son completamente adecuados para el caso de programas concurrentes, por lo que se han propuesto extensiones a los mismos, como se verá en la sección 3.5. Por el lado del testing funcional, no se han desarrollado hasta el momento técnicas de aplicación a gran escala, probablemente por las dificultades expuestas en la sección 4.4.1, donde se comenta acerca de uno de los objetivos no alcanzados durante el desarrollo del presente trabajo: el de dar una técnica para extraer, a partir solo de una especificación, el subconjunto de intercalaciones relevantes a los fines de verificar una determinada propiedad concurrente.

3.4.2. Para ejecutar casos de prueba

Antes de ver por qué serían inadecuados los criterios tradicionales, notar que se puede desear dos grados de confianza respecto de la correctitud de un programa concurrente:

- Confianza respecto de todos los órdenes de ejecución posibles.
- Confianza respecto de los órdenes de ejecución considerados ("recubiertos")

Esto es lo que en [42] se llama **propiedad 2-D** del testing de programas concurrentes. Los criterios tradicionales de testing, al ignorar los órdenes de ejecución posibles de un programa, no brindan ninguna confianza en ninguno de los dos puntos anteriores. Que un caso de prueba sea exitoso al ser ejecutado, no ayuda a creer que lo sería en todos los órdenes de ejecución posibles, y como además no se lleva cuenta de dichos órdenes, no se puede saber cuáles fueron recubiertos por un mismo caso de prueba.

3.4.3. Cómo esto afecta al ciclo de desarrollo

Testing de regresión. Como se mencionó en la sección 2.2, a medida que aumenta la edad de una pieza de software, el costo de mantenimiento prevalece por sobre el costo de desarrollo de la misma. Parte de éste costo corresponde al testing, y en particular al *testing de regresión*, que permite brindar alguna confianza acerca de que los cambios introducidos en el programa como parte del mantenimiento, no introducen defectos, y que el programa modificado sigue cumpliendo su especificación. Por lo tanto, un proceso de testing de regresión bien establecido ayuda a reducir el costo del software [22][8].

Sin embargo, en presencia de programas concurrentes, el testing de regresión no brinda tal confianza, ya que como se mencionó en la sección 3.4, debido al no-determinismo, la semántica de tales programas no son una función del conjunto de posibles datos de entrada al conjunto de posibles datos de salida, con lo cual no se cuenta con la **reproducibilidad** necesaria para implementar un testing de regresión confiable, y aún cuando se contara con dicha característica utilizando algún tipo de mecanismo de ejecución determinista, como los que se verán en la sección 3.5.2, se sigue teniendo un proceso poco confiable debido a que nada asegura en principio que la ejecución

[8] Según estos autores, el testing de regresión no es solo una actividad de mantenimiento, sino que también constituye una actividad del desarrollo, pero no se analizará este aspecto aquí

determinista elegida es la adecuada para ejecutar un caso de prueba dado. El programa podría ser correcto para ese caso de prueba en esa ejecución determinista, pero incorrecto para el mismo caso en otro orden de ejecución.

Otro aspecto a tener en cuenta en el testing de regresión de programas concurrentes surge cuando se utiliza una *estrategia selectiva*. Una estrategia selectiva es una tal que dada una modificación de un programa P con un conjunto de casos de prueba T, debe elegir un subconjunto T' de T que contiene solo los casos de prueba pertinentes a la modificación realizada.

Los criterios para definir una estrategia selectiva son variados y dependen del tipo de testing a realizar (de unidad, de integración o de sistema). Aún en el caso específico de estrategias selectivas para testing de unidad, hay varios enfoques, tanto basados en particiones del dominio de entrada como análisis de flujo de datos, técnicas de slicing y otros [22], pero sin duda que al tratar con programas concurrentes, se añade un factor de complejidad extra al tema, ya que se deberán considerar además las interrelaciones entre los distintos hilos de ejecución.

Una alternativa a las estrategias selectivas son las estrategias de tipo *retest-all*, donde simplemente se reejecutan todos los casos de prueba ante cada modificación del programa asociado. Si bien en principio parecería más costosa que una estrategia selectiva, esto podría no ser así si el esfuerzo necesario para seleccionar casos de prueba pertinentes a una modificación es superior al de aplicarlos todos. Por lo dicho en el párrafo anterior, se puede pensar que en el caso de programas concurrentes una estrategia de tipo *retest-all* puede ser una opción atractiva.

Si bien, en la práctica actual en la industria, la frecuencia con la que se realizan los tests de regresión así como las situaciones que disparan la necesidad de realizarlo, varían mucho de una empresa a otra y de un proyecto a otro, la percepción actual es que un proceso de testing de regresión bien establecido es de fundamental importancia en el desarrollo de una pieza de software [35].

Conclusión. Todo lo anterior sugiere que el costo de mantenimiento de un programa concurrente sería más elevado que el de un equivalente secuencial, y esto es algo que debe tenerse en cuenta en las primeras etapas del proceso de desarrollo. Es decir que se debe balancear este mayor costo de mantenimiento con las posibles ventajas de performance que se obtendrían al hacer un programa (o una parte de él) concurrente. Considerar ambas cuestiones en conjunto ayudaría a tomar una decisión más fehaciente respecto a la relación costo/beneficio al "paralelizar" un programa.

3.5. Estado del arte del testing de programas concurrentes

Algunas de las cuestiones a resolver en testing de programas concurrentes son [47]:

- Cómo lidiar con el no-determinismo.
- Búsqueda de criterios de adecuación de datos (específicos para concurrencia).
- Cómo reutilizar técnicas de testing ya existentes, por ejemplo, la gran cantidad de técnicas que existen para programas secuenciales tradicionales.
- Cómo hacer *timing-related testing*, fundamental en programación de tiempo real y en programación asíncrona.

De estas cuestiones, el presente trabajo solo hace aportes a la primera. Sobre la segunda y la tercera se mencionará el estado del arte actual, y sobre la cuarta, timing-related testing, no se considerará nada en absoluto, no porque carezca de importancia o interés, sino porque su misma complejidad y particularidades requieren que sea tratado en un trabajo específico sobre el tema.

El no-determinismo plantea dos grandes problemas, como ya se vió: por un lado, para la obtención de casos de prueba, las técnicas deben tener en cuenta por lo general todas las intercalaciones posibles del programa en tiempo de ejecución, lo que lleva a un problema de explosión de estados o crecimiento exponencial del número de caminos de ejecución o caminos de datos a considerar, y por otro lado, para la ejecución de casos de prueba, el no-determinismo plantea la dificultad de que rompe la reproducibilidad de la ejecución de un mismo caso. Lo que se hace comúnmente para manejar este último inconveniente es llevar o forzar al programa concurrente a que realice una ejecución determinista. Ambas cuestiones se analizan con más detalle en las próximas secciones, exponiendo algunas de las técnicas existentes.

3.5.1. Criterios para obtener casos de prueba

Se puede citar como ejemplo el propuesto en [47], donde se investiga la factibilidad de reutilizar el conocido criterio ***all-du-path*** (sección 2.4.1) de programas secuenciales, para ser aplicado a la extracción de casos de prueba

para programas concurrentes. El criterio *all-du-path* tradicional consiste en cubrir cada camino de definición-uso en el programa por al menos un caso de prueba en el conjunto de pruebas. En ese trabajo los autores definen un Grafo de Flujo de Programa Paralelo (PPFG por sus siglas en inglés), y redefinen el criterio *all-du-path* para ese tipo de grafos.

Sin embargo, no dan un algoritmo para extraer casos de prueba que cumplan el criterio. Además proponen (sin demostrar formalmente) que para cualquier algoritmo que extraiga casos de prueba que cumplan el criterio, existirán ciertas componentes cuya obtención será no computable (específicamente, será no computable la obtención de lo que los autores definen como "camino de cubrimiento t-ejecutable"). La conclusión del trabajo es que a partir de un grafo PPFG, es imposible automatizar completamente el proceso de extracción de casos de prueba que cumplan el criterio *all-du-path* para programas concurrentes.

Otro método de generación automática de casos de prueba es el dado por los mismos autores en [48], y está basado en la inserción de *delays* (retardos) en el programa concurrente. Esto en principio podría dar como resultado conjuntos de prueba muy costosas (en cuanto a tiempo de ejecución requerido). Por esta razón, el enfoque se complementa con una técnica de reducción de ese costo, mediante la identificación y subsecuente eliminación de **puntos de delay redundantes**. El cálculo de este tipo de puntos se realiza mediante un análisis de flujo de datos del programa. Es decir que dicha reducción se lleva a cabo mediante un análisis estático del programa.

Este enfoque tiene como principales ventajas y desventajas una misma cosa: el estar basado en el tiempo, es decir, en ser básicamente un mecanismo de *timing-related testing*. Sin embargo, se debe mencionar como ventaja importante, el hecho de que al no utilizar como centro del análisis la información de scheduling, o sea, todos las posibles intercalaciones de las acciones del programa, es de fácil implementación en multiprocesadores, ya que no requerirá al momento de la ejecución, utilizar eventos de sincronización. Por la misma razón, no es necesario para el análisis la construcción de un grafo de concurrencia como en el método anterior del criterio *all-du-path* , aunque por otra parte esta aparente ventaja se ve mitigada por el hecho de que debe realizarse un análisis de flujo de datos, tarea que no es trivial para la mayoría de lenguajes.

Aún así, el enfoque podría resultar ventajoso, comparado con el desarrollado en este trabajo, en sistemas multiprocesadores. Sin embargo, debido a que el presente trabajo no incursiona en el terreno del testing de programas concurrentes en multiprocesadores, ni de *timing-related testing* ni de criterios

basados en flujo de datos, no se lo considerará en lo subsiguiente.

Otro acercamiento a la obtención de casos de prueba en programación concurrente es es el llamado **reachability testing** [18]. En primer lugar, se define lo que se llama una *secuencia de sincronización* (SYN-sequence)[9]. Luego los autores presentan un método que es intermedio entre testing no-determinista (donde un programa concurrente se ejecuta varias veces con la misma entrada con la intención de ejecutar distintas SYN-sequences) y testing determinista (donde el programa concurrente se ejecuta con una entrada determinada y en una SYN-sequence dada).

El método trabaja sobre un programa P tomando un par (X, S), donde X es una entrada para el programa P y S es una SYN-sequence para P, y siguiendo estos pasos:

1. Se usa S para derivar un conjunto de prefijos para otras SYN-sequences admisibles para P. Esos prefijos se llaman *race-variants* de S y se obtienen cambiando el resultado de las condiciones de carrera[10] en S.

2. Se realizan ejecuciones (deterministas) de P con entrada X, en cada race-variant de S, de manera de recolectar SYN-sequences admisibles adicionales para P con entrada X.

3. Para cada nueva SYN-sequence obtenida en el paso anterior, se repiten recursivamente los pasos 1 y 2.

En la descripción anterior faltan detalles, como la forma de identificar las condiciones de carrera en una SYN-sequence o qué hacer cuando el número de SYN-sequences admisibles para una entrada dada son infinitas. Esas y otras cuestiones pueden consultarse en el trabajo citado, pero los puntos más importantes a resaltar a los fines del presente trabajo son:

- El método es dinámico, pues se van generando casos de prueba a medida que se ejecutan.

- El método es estructural, pues la identificación de las condiciones de carrera requiere analizar el código del programa.

- Si bien el método es aplicable a distintos lenguajes y a distintos mecanismos de *threading*, no es independiente de ellos, pues una vez fijado

[9] El formato de esta secuencia depende de las construcciones concurrentes utilizadas en el programa.

[10] *Race conditions*.

el lenguaje y el mecanismo concurrente subyacente, el formato de las SYN-sequences y la identificación de las condiciones de carrera está determinado por ellos.

Estas características hacen que, aparentemente, el método no pueda ser utilizado para derivar casos de prueba concurrentes a partir solo de la especificación, es decir, mediante testing funcional. Esa limitación no es propia del método, sino que parece ser un rasgo común a la programación concurrente. Aparentemente es bastante difícil desarrollar una técnica de extracción de casos de prueba para un programa concurrente a partir solo de su especificación. Se volverá sobre esto en la sección 4.4.4.

3.5.2. Métodos para ejecución de casos de prueba

Una vez que ya se cuenta con los casos de prueba con los cuales se va a someter a verificación al sistema concurrente, la cuestión pasa a ser cómo ejecutar esos casos. Esto es lo que se analiza a continuación.

Mecanismos de *record/replay*

La mayoría de métodos de ejecución son variaciones de un mismo mecanismo conceptual llamado de *record/replay* (debido a los conceptos involucrados: *Monitoring o Record* y *Replay*) y se diferencian entre sí solo en la forma y eficiencia con que implementan esos conceptos [20].

En este tipo de mecanismos, en una primera corrida de un caso de prueba se monitoriza o "graba" su ejecución, almacenando información de *scheduling* o de otro tipo que permita reproducir en el futuro la ejecución de ese mismo caso en condiciones similares. Por ejemplo, en el método dado en [36] esto se hace utilizando *Contadores de Instrucciones en Software*, con la limitación de que el método solo funciona en monoprocesadores y a nivel de un solo proceso (quizá con varios threads) que no realiza IPC[11].

Una descripción bastante completa de las variaciones actualmente disponibles de este tipo de mecanismos de testing se puede encontrar en [34]. En todas estas variaciones, un requerimiento fundamental para que la etapa de *replay* sea posible, es que el sistema de *scheduling* subyacente sea visible al mecanismo. También hay que aclarar que las ejecuciones repetibles se pueden utilizar no solo para hacer testing, sino también en otras áreas como monitorización de rendimiento de sistemas o tolerancia a fallos [36].

[11] Inter-Process Communication.

Nivel de abstracción. Un punto importante a tener en cuenta, como se mencionó en la sección 3.2.4, es que en este tipo de métodos juega un papel fundamental el nivel de abstracción al cual se considera el programa. Es decir, una vez elegido un nivel de abstracción, estos mecanismos guardarán información que sea relevante para reproducir una ejecución determinista *a ese nivel* y superiores, aunque en dicha ejecución podría haber no-determinismo a niveles inferiores del elegido. De manera más abstracta, la dificultad radica en que al variar el nivel de abstracción, se está cambiando la noción de **equivalencia de procesos** con la cual se está trabajando (noción que es necesaria pues lo que se desea es reejecutar un programa de manera "equivalente" a cierta ejecución original). Una ventaja muy importante del método presentado en este trabajo con respecto los mecanismos de *record/replay* es que otorga una gran flexibilidad a quien escribe los casos de prueba, ya que éste puede expresarlos al nivel de abstracción que desee.

El entorno desarrollado no utiliza *record/replay*. Si bien es cierto que la técnica descripta en el presente trabajo consiste en forzar al programa a ejecutar de manera determinista, esa ejecución determinista de un caso de prueba no se toma de una ejecución particular "memorizada" previamente como hacen los mecanismos de *record/replay*, sino que un mismo caso de prueba se ejecuta de manera determinista en *todas* las intercalaciones posibles del caso en cuestión, como se verá en el próximo capítulo.

Esto representa una gran ventaja frente a todos los métodos de tradicionales de *record/replay*, que como se dijo son casi el único enfoque a la ejecución de casos de prueba concurrentes, ya que en estos métodos un caso de prueba solo pueden ser usado para mostrar la presencia de errores en *una* intercalación (la guardada en la etapa de record) pero ¿qué pasa con las demás intercalaciones posibles?

4. TESTING DE UNIDAD DE PROGRAMAS C CONCURRENTES

4.1. Introducción

En el presente capítulo se describe el entorno desarrollado, el cual se instrumentó en forma de extensión a una herramienta *open source* preexistente y de gran difusión, llamada CUnit, que es utilizada para realizar testing de unidad de programas en lenguaje C y a la cual le fue añadida soporte específico para testing de unidad de casos de prueba concurrentes. La extensión desarrollada fue bautizada CUThread.

Se comenzará explicando qué es CUnit y cómo se usa, mediante un código de ejemplo, para luego pasar a presentar el mecanismo de corrutinas que le permite a la extensión CUThread soportar concurrencia de manera portable. Luego se presentará la extensión explicando sus objetivos, implementación y modo de uso, así como sus limitaciones. Se finalizará comentando acerca de otras opciones, finalmente descartadas, que se consideraron en las primeras etapas de la tesina para alcanzar los objetivos propuestos.

4.2. CUnit, una herramienta para Testing de Unidad

CUnit es un sistema para escribir y ejecutar casos de prueba de unidad para programas C. Está implementado como una biblioteca, que durante la fase de testing se enlaza con el código a ser testeado. Es una gran ayuda a la hora de administrar los casos de prueba, ya que permite agruparlos en conjuntos de casos de prueba, ejecutar conjuntos específicos o tests específicos, todo esto desde diferentes interfases desde las cuales se pueden redirigir las salidas de los tests para generar distinta clase de reportes. Todo eso se puede hacer de manera automática o bien interactiva.

La posibilidad de automatización de la ejecución que brinda el entorno es de gran importancia debido a que si se piensa al testing como verificación

por experimentación, dicha automatización permite repetir los experimentos todas las veces que sean necesarios sin mayor esfuerzo para los desarrolladores, lo que en general implicará que se testee mucho más seguido a lo largo del ciclo de vida del software.

Los casos de prueba están estructurados alrededor de **aserciones**, que permiten comprobar el cumplimiento (o no) de una determinada propiedad del programa en un punto de la ejecución. La expresión o fórmula de dicha propiedad puede ser cualquier condición lógica del lenguaje C. La aserción no tomará ninguna acción con respecto al programa testeado, solo guardará en una base de datos asociada a la ejecución el resultado de la condición lógica en cuestión, junto con información de en qué punto (archivo y número de línea) particular del programa ocurrió. Al final de la ejecución, o cuando el ejecutante del conjunto de prueba lo pida interactivamente, se generará un reporte con los resultados almacenados en esa base de datos.

Nota histórica. CUnit es un derivado específico para C del exitoso entorno de testing **xUnit**. Se llama así a un diseño genérico basado en las ideas originalmente desarrolladas para sUnit (un entorno de testing de unidad para Smalltalk) creado por Kent Beck. Hoy en día existen versiones disponibles de derivados de xUnit para una enorme cantidad de lenguajes de programación, y en particular para todos los lenguajes de uso común en la industria. Esto es debido a la generalidad de dicho entorno y a lo adaptables que fueron sus ideas básicas (tests organizados en conjuntos de prueba, aserciones para expresar cumplimiento de propiedades, manejo uniforme de la salida, reportes) a una gran variedad de lenguajes.

Ejemplo de uso. A continuación se muestra un programa que hace uso de CUnit para ejecutar dos casos de prueba:

```
#include <string.h>
#include <CUnit/CUnit.h>

int max (int i, int j)
{
  return (i > j) ? i : j;
}

void max_test (void)
{
  CU_ASSERT (max (3,0) == 3);
  CU_ASSERT (max (-1,0) == 0);
```

```
  CU_ASSERT (max (2,2) == 2);
}

void strcat_test (void)
{
  char greeting[32] = "Hello";

  CU_ASSERT_STRING_EQUAL (strcat (greeting, " World"), "Hello World");
  CU_ASSERT (1 > 2);
}

int main (int argc, char *argv[])
{
  CU_pSuite test_suite;
  CU_pTest test_case;

  CU_initialize_registry();
  test_suite = CU_add_suite ("example", NULL, NULL);
  if (!test_suite) {
    perror ("cannot initialize test suite");
    return 1;
    }
  CU_add_test (test_suite, "max_test", max_test);
  CU_add_test (test_suite, "strcat_test", strcat_test);
  CU_basic_run_tests ();

  return 0;
}
```

Un primer test comprueba la correctitud de una función implementada, mientras que un segundo test comprueba que una función de biblioteca esté funcionando como se espera. La segunda aserción de `strcat_test`, que por supuesto fallará, se escribió con el objetivo de mostrar cómo aparecen las aserciones fallidas en el reporte generado por CUnit que se mostrará más adelante.

Observar que un mismo test puede contener una cantidad arbitraria de aserciones. Cuántas será adecuado poner en un mismo test es una decisión del equipo que lleva adelante el testing. La práctica usual es que un test someta a prueba a una característica puntual de una función o unidad específica. Por ejemplo, cuando se desea probar el comportamiento de la función `strcat`, usualmente se escribirán varios tests `strcat_test1`, `strcat_test2`, etc., que

probarán distintas cualidades de la función.

Siguiendo con el ejemplo, se ve que en la función principal se crea un conjunto de prueba llamado "example", a la cual se añaden los dos tests[1]. En el caso de tener una gran cantidad de tests, CUnit ofrece métodos para simplificar este proceso. Por ejemplo, existe la macro `CU_ADD_TEST` que evita el tener que darle un nombre al test, pues lo hace automáticamente en base al nombre de la función que se le pasa. Otra posibilidad es indexar los tests mediante un arreglo de estructuras `CU_TestInfo`, referenciar ese arreglo en una estructura `CU_SuiteInfo` y luego registrar el conjunto de prueba completo (junto con todos los tests referenciados) mediante una sola llamada a `CU_register_suites()`, lo que simplifica el manejo de grandes conjuntos de prueba que contengan muchos tests.

Al compilar y ejecutar el ejemplo, el reporte generado por CUnit es:

```
CUnit - A unit testing framework for C - Version 2.1-0
http://cunit.sourceforge.net/


Suite example, Test strcat_test had failures:
    1. example1.c:21  - 1 > 2

Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      2      2      1      1        0
             asserts      5      5      4      1      n/a

Elapsed time =    0.000 seconds
```

Se informa que en total se ejecutaron 2 tests en 1 conjunto de prueba, así como 5 aserciones en total, de las cuales 1 falló. Sobre esta aserción fallida se dan más detalles: se dice que aparece en la línea 21 del archivo fuente example1.c (que es la línea que contiene la aserción `CU_ASSERT(1 >2)`) y que corresponde al test `strcat_test` del conjunto de prueba llamado "example". Además se muestra la condición que motivó el fallo ($1 > 2$).

[1] Por simplicidad se omite la comprobación de errores de esta parte.

4.2.1. Limitación de CUnit para testing de programas concurrentes

Como se dijo, la automatización de la ejecución de casos de prueba debería permitir repetir los experimentos de testing todas las veces que sean necesarios sin mayor esfuerzo para los desarrolladores. CUnit ofrece esta automatización y además, como trabaja almacenando los resultados de las condiciones lógicas de las aserciones junto con el punto del programa (número de línea) en donde dicha aserción se encuentra, ayuda a la identificación de defectos en programas secuenciales. Pero esto pierde eficacia al tratar con programas concurrentes, ya que para éstos no basta con conocer el punto *en el código* en el cual una aserción es falsa, pues eso no dice lo suficiente acerca del punto *en la ejecución del programa* en el cual la aserción falló. Es decir, no se sabe *con qué secuencia de instrucciones* se llegó a dicha falla.

Por otro lado, los beneficios de la automatización de la ejecución se ven disminuidos por el hecho de que nada garantiza que entre una ejecución y otra de un programa concurrente se siga la misma secuencia de instrucciones, con lo que una misma aserción puede resultar verdadera y falsa en distintas ejecuciones de un mismo caso de prueba.

Es para paliar estas limitaciones que se creó la extensión CUThread[2]. Con ésta, CUnit puede hacer frente mucho mejor al testing de programas concurrentes. Pero antes de pasar a ver la extensión en sí, en la próxima sección se presentarán los mecanismos sobre los que está basada la misma.

4.3. Corrutinas

Las corrutinas representan "hilos" en CUThread, es decir, están en la base de su sistema de concurrencia.

Se llama corrutinas a un conjunto de procedimientos o funciones que trabajan cooperativamente para solucionar una tarea, pero con la particularidad de que el control se transfiere de una a otra no necesariamente siguiendo la política tradicional de llamada a procedimiento o haciendo llamadas explícitas a la corrutina que deberá continuar la ejecución.

Recordar que en la política tradicional, luego de una llamada a una función f, el control es transferido a f y luego, cuando f retorna, el control vuelve al punto posterior a donde se hizo la llamada a f. Esta política se ha instaurado históricamente así pues se lleva bien con las arquitecturas de

[2] Se mencionan aquí solo las limitaciones referentes al testing de programas concurrentes. También existen otras, como por ejemplo la falta de un mecanismo para determinar si un conjunto de casos de prueba cumple o no un criterio de recubrimiento dado.

stack sobre las cuales están basados la mayoría de los microprocesadores, y porque es adecuada para razonar sobre programas secuenciales. Sin embargo, para programas concurrentes, por lo general no brinda la mejor expresividad (ver conclusión al final de esta sección).

A continuación se mostrará una característica del lenguaje C que permite implementar un mecanismo de continuaciones (restringidas) en dicho lenguaje, y luego se verá cómo utilizando este mecanismo se puede implementar todo un sistema de programación concurrente mediante corrutinas, el cual tiene la característica de ser independiente de la plataforma (ni siquiera requiere la presencia de un Sistema Operativo).

Las **continuaciones** son un mecanismo alternativo de transferencia de flujo de control estructurado. En el esquema tradicional de trasferencia de flujo de control hay funciones llamantes que invocan o llaman a funciones y esperan que retornen para luego continuar procesando. Con esto, se va formando durante la ejecución un *árbol de invocación* de funciones.

Por otro lado, con un esquema de continuaciones, una función que llama a otra le proporciona junto con el resto de argumentos un argumento especial, una "continuación", que es una cierta estructura que indica de alguna manera dónde la función llamada debe **continuar** la ejecución (transferir el control) una vez que termine su procesamiento. Es decir que la función ya no "retorna", sino que "continúa" en el lugar indicado por la continuación.

Claramente esto permite que se puedan establecer estructuras de flujo de control que no estén restringidas a una estructura de invocación en forma de árbol. Es debido a esto que, como se afirma más arriba, la estructura tradicional de invocaciones de funciones en forma de árbol no presenta la mejor expresividad en ciertos tipos de programas, como por ejemplo en los programas concurrentes. Para estos programas, un esquema de continuaciones parece adaptarse más naturalmente.

4.3.1. Duff's device

Se llama *Duff's device* a un truco de programación en C inventado en el año 1983 por Tom Duff de los laboratorios Bell de AT&T (aunque al momento de la invención se encontraba trabajando en Lucasfilm). La idea surgió a partir del deseo de optimizar una función de tipo *memcpy*():

```
/* Falla si count es 0 */
send (short to, short from, int count)
{
  do
    *to = *from++;
```

```
  while (--count>0);
}
```

Una solución alternativa que se le ocurrió a Tom Duff para mejorar el rendimiento fue la de hacer uso de un *loop unrolling*[3] para disminuir la cantidad de restas y comparaciones con cero que se intercalaban con las copias de *from* hacia *to*, y de esa forma concatenar varias copias seguidas con una sola resta seguida de comparación. Lo novedoso no fue el loop unrolling, sino la forma de implementarlo en C:

```
send (short to, short from, int count)
{
  int n = (count+7)/8;

  switch (count%8) {
  case 0: do { *to = *from++;
  case 7:   *to = *from++;
  case 6:   *to = *from++;
  case 5:   *to = *from++;
  case 4:   *to = *from++;
  case 3:   *to = *from++;
  case 2:   *to = *from++;
  case 1:   *to = *from++;
          } while (--n>0);
  }
}
```

Observar que las etiquetas de la sentencia `switch`, excepto la 0, se encuentran *dentro* del bucle `while`. También recordar que en C, una vez que se salta a una etiqueta en una sentencia `switch`, se sigue ejecutando lo que resta en el bloque a menos que explícitamente se salga de él con una sentencia `break`.

Teniendo en cuenta esto, se puede entender la función anterior: mediante un *loop unrolling* se agrupan las sentencias de ocho iteraciones distintas del `while` original en un sola iteración del nuevo `while`, y para arreglar el problema de qué hacer cuando `count` no sea múltiplo de ocho, se hace un `switch`

[3] También llamada *loop unwinding*. Es una técnica de optimización de programas en la cual se trata de ahorrar tiempo de cómputo reduciendo de instrucciones que el programa debe ejecutar para llevar adelante el loop. Para lograr esto, se toma un conjunto de instrucciones que serían llamadas en distintas iteraciones del loop y se las ejecuta en una sola iteración del mismo. Esto ahorraría la sobrecarga de algunas instrucciones "administrativas" del loop, como el incremento de índices de iteración, etc.

sobre el resto de dividir `count` entre ocho para de esa manera saltar a la sentencia correspondiente **dentro** del while (esto es el Duff's device) de manera de terminar de copiar ese "resto" que falta.

Algo muy importante es que el código anterior cumple con los estándares de C. Más aún, en [13] Tom Duff asegura que Larry Rosler, chairman del subcomité de estandarización X3J11, le comentó que dicha característica (la posibilidad de saltar dentro de un `while`) fue cuidadosamente estudiada y finalmente se decidió que era legal.

Muy bien, hasta ahora, por medio del Duff's device, se tiene un mecanismo para expresar loop unrolling en C. La pregunta es ¿cómo puede ser útil este truco para la programación concurrente? La respuesta es que dicho truco tiene otros usos, por ejemplo, como ya lo notó su descubridor en [13], para implementar máquinas de estado dirigidas por interrupciones, a las que Simon Tatham les dió una estructurada y reutilizable forma final mediante sus *continuaciones locales*[41].

4.3.2. Continuaciones locales

Las continuaciones locales brindan un mecanismo básico para implementar corrutinas en C. Recordando que para las corrutinas lo que se necesita es que haya un mecanismo mediante el cual una corrutina le pueda ceder el control a otras, pero no necesariamente haciendo una llamada explícita a la corrutina que continuará la ejecución, sino simplemente "abandonando" el uso del procesador, se verá cómo utilizar las continuaciones para esto analizando en primer lugar como se puede utilizar el Duff's device para implementar continuaciones locales.

Estas continuaciones se llaman locales pues no son tan generales: no se abstraen por completo del hecho de que se están ejecutando en una máquina de stack con una llamada a función con política tradicional. De hecho, habrá un gran *dispatcher* o manejador general del sistema de corrutinas, que hará todo el tiempo llamadas a funciones convencionales. Es decir que esta rutina, *llamante* de todas las corrutinas, se encontrará en un ciclo `while` haciendo llamadas a funciones según la convención tradicional. El truco estará en las funciones *llamadas*.

La corrutina llamada será una función que tendrá una operación de tipo *return and continue*: hará que se retorne de la función, pero además provocará que la próxima vez que ésta sea llamada, la ejecución se retome a partir de la sentencia inmediatamente posterior a donde se hizo el return anterior. Aquí es donde se utilizará el Duff's device.

A partir del fragmento de código mostrado en la sección anterior, se ve que algo como lo siguiente presenta el comportamiento deseado:

```
int function (void)
{
  static int i, state = 0;

  switch (state) {
  case 0: /* comienzo de la corrutina */
    for (i = 0; i < 10; i++) {
      state = 1; /* la próxima vez el switch irá al "case 1" */
      return i;
  case 1: /* la próxima vez se ejecutará desde aquí */
    }
  }
}
```

Observar que la segunda etiqueta del `switch` se encuentra *dentro* del bucle `for`, lo que provoca que cuando `state` tenga el valor 1 (en este ejemplo, ocurrirá luego de la primera llamada a la función) se salte dentro del bucle. Es decir, se continuará en el punto siguiente al del retorno previo desde la misma función (ya que la etiqueta 1 se encuentra inmediatamente después de la sentencia `return i`).

Entonces esto funciona como se deseaba, pero el código resultante no es muy claro para quien no comprenda como trabaja el Duff device. Para hacerlo más elegante y reutilizable, se pueden esconder los detalles detrás de un conjunto de macros, ya que la estructura básica será siempre la misma. Una primera versión sería:

```
#define crBegin       static int state=0; switch(state) { case 0:
#define crReturn(i,x) do { state=i; return x; case i:; } while (0)
#define crFinish      }

int function (void)
{
  static int i;

  crBegin;
  for (i = 0; i < 10; i++)
      crReturn (1, i);
  crFinish;
}
```

Es decir, el comienzo y el final de la corrutina deberá marcarse con `crBegin` y `crFinish` respectivamente, y `crReturn` es la operación de tipo

return and continue deseada. Se debe tener en cuenta que las variables locales deberán ser `static` si se quiere que sus valores se preserven luego de una llamada a `crReturn`[4], y que esta última macro no deberá ser usada dentro de un `switch` explícito.

Para obtener la versión final solo falta un detalle. En el ejemplo dado, hay un único punto en el cual se retomará la ejecución luego de un retorno (hay un solo `crReturn`). Esto hace que `state` deba tomar dos posibles valores: 0 si nunca se llamó a `crReturn` o 1 para el (único) punto de reinicio. En general, si se tienen n posibles puntos de reinicio de ejecución (n invocaciones a `crReturn`), se necesitarán también $n + 1$ valores *diferentes* para state[5] (para poder distinguir los distintos puntos en los cuales se debe retomar la ejecucón en la próxima invocación). Mantener esto a medida que el programa crece, es muy tedioso y sujeto a errores, además de aumentar el costo de mantenimiento del código.

Para solucionarlo, se lo puede automatizar haciendo uso de la macro `__LINE__`, que es parte del estándar de C, y es expandida por el preprocesador y reemplazada por el número de línea del archivo fuente en la que la misma aparece. Aprovechando la **monotonicidad** con que crecen los números de línea en un mismo archivo fuente, se puede definir `crReturn` como sigue (observar que se eliminó el primer argumento de la versión anterior):

```
#define crReturn(x) do { state=__LINE__; return x; \
                         case __LINE__:; } while (0)
```

con lo cual se libra al programador de la administración de los posibles valores para los puntos de reinicio de ejecución[6]. Con esto se tiene todo lo necesario para implementar continuaciones locales en C de manera portable, que pueden utilizarse para implementar todo un conjunto completo de primitivas para programación concurrente, como se verá en la siguiente sección.

4.3.3. Protothreads

Protothreads es una biblioteca que proporciona soporte para programación multihilo en C de una manera portable y casi sin costo extra tanto en tiempo como en espacio. Implementa esto haciendo uso de un sistema de continuaciones locales creado por Adam Dunkels, que a su vez están basadas en las de Simon Tatham vistas en la sección anterior. Adam Dunkels utilizó estas continuaciones para implementar el sistema operativo embebido

[4] O bien declararlas como globales.

[5] Valores que son pasados mediante el primer argumento de `crReturn`.

[6] Por supuesto, se debe tener cuidado de no colocar dos invocaciones a `crReturn` en una misma línea.

Contiki[7] desarrollado por el Instituto Sueco de Ciencias de la Computación. Contiki es un SO multitarea con soporte de TCP/IP, sistema de ventanas, navegador web y acceso remoto via VNC, además de otras características interesantes, como la posibilidad opcional de multitarea preemptiva y otras.

Protothreads no es en realidad una biblioteca en la forma tradicional del término, sino sólo un conjunto de macros distribuidas en archivos de cabecera C, que al ser utilizadas para implementar un programa concurrente, expanden a código C 100 % "puro" e independiente de cualquier plataforma, es decir, no hace uso de llamadas al sistema ni a bibliotecas externas ni utiliza *assembler* ni ninguna característica particular de algún compilador. Esto también hace que pueda ser utilizada **con o sin un Sistema Operativo subyacente**.

Protothreads cumple su objetivo casi sin costo extra pues la sobrecarga de cómputo para la sincronización es ínfima y el espacio extra utilizado es de **solo 2 bytes por hilo de ejecución**. Además, no se utiliza un stack aparte para cada thread. Al ser la biblioteca solo un conjunto de macros que expanden a código C, el programa compilado no debe enlazar en tiempo de ejecución con ninguna "biblioteca Protothreads" o algo similar: todo la funcionalidad multihilo es llevada a cabo por el propio código del programa.

Para comprender como funciona, primero se presenta la implementación (muy simple) de continuaciones locales dada por Adam Dunkels en el archivo de cabecera `lc.h`:

```
typedef unsigned short lc_t;
#define LC_INIT(s)   s = 0;
#define LC_RESUME(s) switch(s) { case 0:
#define LC_SET(s)    s = __LINE__; case __LINE__:
#define LC_END(s)    }
```

`LC_RESUME` marca el punto de inicio de una continuación local, `LC_SET` es la operación de tipo *return and continue* vista en la sección anterior, y que sirve para marcar un punto de reinicio de ejecución, y `LC_INIT` y `LC_END` son los delimitadores del código de la continuación.

Ya con esto, se pueden definir las construcciones concurrentes brindadas por la biblioteca. Las construcciones básicas son (resumen extraido del archivo de cabecera `pt.h`):

```
#include "lc.h"

struct pt {
  lc_t lc;
```

[7] http://www.sics.se/contiki/

```
};

#define PT_THREAD_WAITING 0
#define PT_THREAD_EXITED  1
#define PT_THREAD(name_args) char name_args
#define PT_INIT(pt)   LC_INIT((pt)->lc)
#define PT_BEGIN(pt) { PT_YIELDING(); LC_RESUME((pt)->lc)
#define PT_END(pt) LC_END((pt)->lc); pt_yielded = 0; PT_EXIT(pt); }
#define PT_SCHEDULE(f) ((f) == PT_THREAD_WAITING)
#define PT_WAIT_UNTIL(pt, condition)         \
  do { \
    LC_SET((pt)->lc); \
    if(!(condition)) { \
      return PT_THREAD_WAITING; \
    } \
  } while(0)
#define PT_WAIT_WHILE(pt, cond)  PT_WAIT_UNTIL((pt), !(cond))
#define PT_YIELDING() char pt_yielded = 1
#define PT_YIELD(pt) \
  do { \
    pt_yielded = 0; \
    PT_WAIT_UNTIL(pt, pt_yielded); \
  } while(0)
#define PT_YIELD_UNTIL(pt, cond) \
  do { \
   pt_yielded = 0; \
   PT_WAIT_UNTIL(pt, pt_yielded && (cond)); \
  } while(0)
```

`PT_THREAD` es usada para declarar la función correspondiente a una corrutina. El cuerpo de dicha corrutina deberá estar encerrado entre `PT_BEGIN` y `PT_END`. Observar que `PT_BEGIN` abre una llave sin cerrarla, ya que será cerrada por `PT_END`. `PT_YIELDING` es usado para marcar el hecho de que el thread puede cederle el lugar a otro, invocando `PT_YIELD`. Las funciones de las corrutinas serán llamadas desde un *dispatcher* central o *scheduler* que les dará a todas ellas la oportunidad de ejecutarse mediante el uso de `PT_SCHEDULE`. Por supuesto, el scheduling aquí no es *preemptable*, sino que el modelo de concurrencia es cooperativo: el flujo de control es transferido de una corrutina a otra por propia decisión de las corrutinas y no por la imposición del scheduler[8]. Antes de llamar por primera vez a la función de una corrutina, el

[8] El scheduler a lo sumo impondrá el orden en el cual se le dará oportunidad a cada

scheduler deberá inicializarla mediante `PT_INIT`.

Existen otras operaciones, como `PT_SPAWN` para "lanzar" una nueva corrutina, `PT_RESTART` para reiniciarla, `PT_EXIT` para abandonarla y para esperar a que una corrutina finalice se brinda `PT_WAIT_THREAD`. También se cuenta con un sistema de semáforos: `PT_SEM_INIT`, `PT_SEM_WAIT` y `PT_SEM_SIGNAL`.

Para comenzar a comprender las operaciones anteriores, nada mejor que un ejemplo, como el que se puede observar en la figura 4.1, que es muy simple y autoexplicativo. De hecho es una sobresimplificación, pues hay una sola corrutina, que no tiene mucho sentido como sistema concurrente, pero más adelante se verán ejemplos más reales con más de una corrutina, al mostrar cómo funciona la extensión CUThread desarrollada en este trabajo.

4.4. Entorno desarrollado

Se creó un entorno de testing de programas concurrentes extendiendo CUnit de manera que dé soporte especial a este tipo de programas. A continuación se presentan los objetivos de la extensión, para luego pasar a ver cómo funciona, junto con sus limitaciones. Se finaliza comentando las dificultades encontradas durante el curso del desarrollo.

4.4.1. Objetivos

El entorno debería brindar soporte:

1. Para la ejecución de casos de prueba concurrentes.
2. Para la obtención de las intercalaciones relevantes a un caso de prueba concurrente dado.

En cuanto al primer punto, el programador debería poder escribir casos de prueba de unidad en lenguaje C, definiendo explícitamente qué acciones serán atómicas al momento de la ejecución. Luego estos casos de prueba se agruparán en conjuntos que el entorno ejecutará de manera secuencial en todas las intercalaciones posibles de sus acciones atómicas. También debería existir un mecanismo para ejecutar ese grupo de casos de prueba solo en las intercalaciones dados por el encargado del testing. Estos objetivos se lograron completamente, como se mostrará en breve.

En cuanto al segundo punto, se debería brindar un mecanismo para obtener solo las intercalaciones relevantes a la verificación de una propiedad dada.

corrutina de ejecutar.

```
#include "pt.h"

static int counter;
static struct pt example_pt;

static
PT_THREAD(example(struct pt *pt))
{
  PT_BEGIN(pt);

  while(1) {
    PT_WAIT_UNTIL (pt, counter == 1000);
    printf ("Threshold reached\n");
    counter = 0;
  }

  PT_END(pt);
}

int main (void)
{
  counter = 0;
  PT_INIT (&example_pt);

  while (PT_SCHEDULE(example(&example_pt)))
    counter++;

  return 0;
}
```

Fig. 4.1: Ejemplo de uso de la biblioteca Protothreads.

Esto se podría hacer derivando los casos de prueba a partir de la especificación (donde también estaría expresada la propiedad a verificar) y luego descartando las intercalaciones irrelevantes mediante algún criterio. Dicho método no se desarrolló en el presente trabajo, aunque sí se analizaron algunas opciones al respecto. Mediante ese análisis, presentado más adelante, se pudieron identificar las dificultades fundamentales por las cuales no fue posible cumplir dicho objetivo.

4.4.2. Extensión a CUnit para testing de programas concurrentes: CUThread

La extensión desarrollada provee primitivas para programación concurrente, que se implementan a partir de Protothreads. Además, se diseñó un mecanismo de identificación de todas las intercalaciones posibles, junto con un método de ejecución determinista de una intercalación dada. También se hicieron modificaciones para que las aserciones fallidas de CUnit muestren no sólo el lugar en el código del programa en el cual se produjo el fallo[9], sino que junto con ello muestre la secuencia de instrucciones atómicas con las cuales se llegó a ese punto.

El código de toda la funcionalidad añadida se concentra en dos archivos fuente, CUThread.c y CUThread.h, además de los archivos propios de Protothreads, y no se modificaron en nada los archivos C originales de CUnit, sino solo algunos archivos de compilación, para que consideraran en la construcción del framework la existencia de los archivos añadidos. Todo el desarrollo se realizó tomando como base a CUnit versión 2.10.

Instrucciones atómicas. Las instrucciones atómicas sobre las que se calculan las intercalaciones no están predeterminadas, sino que es el propio diseñador del caso de prueba el que decide la granularidad de las mismas, marcando explícitamente lo que quiere que sea una sentencia atómica pasándola como parámetro a la macro `CU_STM()`. Esta macro cumple una doble función: durante una primera etapa, la de inicialización, obtiene los números de línea en los cuales se encuentran las instrucciones atómicas, y luego, en la etapa de ejecución efectiva del test, fuerza a la corrutina que ejecutará la sentencia a esperar hasta que le llegue su turno.

[9] Esto no se logra en los tests CUnit tradicionales, donde solo se puede localizar el punto *en el test* en donde falló una aserción. En cambio en CUThread ello sí es posible debido a que los fragmentos de código que se desea someter a prueba de forma concurrente con otros fragmentos del mismo programa son extraídos del programa original y transformados en tests CUThread, con lo cual de esa manera se pueden localizar los puntos en el programa (transformado en test) en el cual fallaron las aserciones.

De esta manera, un scheduler central puede realizar una ejecución determinista de un caso de prueba, ordenando los números de línea de las instrucciones a ejecutar, y luego lanzando todas las corrutinas que se deseen en paralelo, ya que cada corrutina esperará a ejecutar (la sentencia que recibió como parámetro mediante la macro `CU_STM`) hasta que le llegue el turno fijado por la ordenación que hizo el scheduler.

Lo anterior se ilustrará con un ejemplo más adelante, pero primero se verá cómo está definida la macro:

```
#define CU_STM(stm)                                                 \
 if (!_CU_thread_initialized)                                       \
  ins_per_thread[CU_current_thread][CU_current_stm++] = __LINE__;\
 else                                                               \
  do {                                                              \
   PT_WAIT_UNTIL (pt, ins[CU_current_stm] == __LINE__);             \
   stm;                                                             \
   CU_current_stm++;                                                \
 } while (0)
```

`ins_per_thread` contiene una entrada para cada corrutina. Dicha entrada es un arreglo donde en la etapa de inicialización se guardarán los números de línea donde comienzan las sentencias atómicas de la corrutina en cuestión[10]. Esa información se utilizará luego para generar las intercalaciones posibles de todas las corrutinas del caso de prueba mediante la función `CU_thread_build_interleaves()`.

Otros métodos brindados por la extensión son: `CU_THREAD_BEGIN_INIT` y `CU_THREAD_BEGIN_END` para indicar el comienzo y fin de la etapa de inicialización, respectivamente, en la cual como se dijo se obtienen los números de las líneas en las que se encuentran las instrucciones atómicas, y por otro lado `CU_thread_show_interleaves()` para mostrar un conjunto de intercalaciones dado. Además se dispone de todas las primitivas de concurrencia ofrecidas por Protothreads, aunque cada una con su prefijo `PT` cambiado por `CU_THREAD`[11]. También se implementaron y reimplementaron algunas macros y funciones internas de CUnit para el manejo de los fallos en las aserciones.

Finalmente, se brinda la función `CU_thread_set_interleave()` que permite fijar la intercalación a ser ejecutada, que era uno de los objetivos a cumplir por el entorno.

[10] Recordar que a pesar de que `CU_STM` parece estar definida en varias líneas, en realidad su expresión ocupa una sola, pues los caracteres de nueva línea están escapados.

[11] No se deben mezclar en un mismo programa macros de Protothreads con las correspondientes macros de tipo `CU_THREAD`.

Ejemplo de uso de la extensión. Supongamos que se tiene un programa que define dos funciones `f` y `g`, y se desea probar el comportamiento concurrente de esas dos funciones. Para hacer esto usando la extensión desarrollada, se deben escribir ambas funciones como corrutinas CUThread. Luego se ejecutarán estas corrutinas en todas sus intercalaciones posibles.

Por ejemplo, si en el programa se tienen `f` y `g` como sigue:

```
static int a;

int f()
{
    int b;

    a = 1;
    b = a;

    return b;
}

void g()
{
    a = 1;
    a = 0;
}
```

y se espera que `f()` siempre devuelva 1, cosa razonable en un programa secuencial tradicional, se puede comprobar la validez o no de esa afirmación utilizando CUThread.

Si se desea comprobar si `f()` siempre devuelve 1 cuando es ejecutada concurrentemente con el fragmento de código correspondiente a la función `g`, se deberán crear dos corrutinas CUThread, una para `f` y otra para `g`, indicando cuáles sentencias se considerarán como atómicas, y escribiendo lo que se quiere comprobar en forma de aserción, al final de `f`, antes de que esta retorne:

```
1   #include <stdlib.h>
2   #include <assert.h>
3   #include <stdio.h>
4   #include <CUThread.h>
5
6   static int a;
7
```

```
8   CU_THREAD(f)
9   {
10     int b;
11
12     CU_THREAD_PROLOGUE();
13
14     CU_STM(a = 1);
15     CU_STM(b = a);
16     CU_ASSERT (b == 1);
17
18     CU_THREAD_EPILOGUE();
19  }
20
21  CU_THREAD(g)
22  {
23     CU_THREAD_PROLOGUE();
24
25     CU_STM(a = 1);
26     CU_STM(a = 0);
27
28     CU_THREAD_EPILOGUE();
29  }
30
31  void scheduler (void)
32  {
33     CU_list *interleaves = NULL;
34     CU_list *current_interleave;
35     int running, fr, gr;
36
37     CU_THREAD_BEGIN_INIT();
38     CU_THREAD_INIT(f);
39     CU_THREAD_INIT(g);
40     CU_THREAD_END_INIT();
41
42     /* Interleaves must be built AFTER thread initialization */
43     interleaves = CU_thread_build_interleaves();
44     CU_thread_show_interleaves (interleaves);
45
46     while (current_interleave = CU_list_item (interleaves)) {
47          CU_thread_set_interleave (current_interleave);
48          CU_THREAD_RESET(f);
```

```
49          CU_THREAD_RESET(g);
50          a = 0;
51          fr = gr = (!0);
52          do {
53              if (fr)
54                  fr = CU_THREAD_SCHEDULE(f);
55              if (gr)
56                  gr = CU_THREAD_SCHEDULE(g);
57              running = (fr || gr);
58          } while (running);
59         interleaves = CU_list_next (interleaves);
60      }
61  }
```

La función `scheduler` se registrará como test dentro de un conjunto de prueba y se ejecutará desde la función `main()` de la manera usual vista en el ejemplo anterior de uso de CUnit (página 45). Al momento de ejecutar el test, `f` y `g` se ejecutarán en todas las intercalaciones posibles de sus acciones atómicas. Claramente algunas de estas ejecuciones harán que falle la aserción de la corrutina `f` (concretamente, aquellas en que `a` haya sido modificada antes de ser asignada a `b`). El resumen generado por CUnit al ejecutar el test del ejemplo es el siguiente:

```
     CUnit - A unit testing framework for C - Version 2.1-0
     http://cunit.sourceforge.net/

{
25 26 ,
14 15 ,
}
{
25 26 14 15 ,
14 25 26 15 ,
25 14 26 15 ,
14 15 25 26 ,
14 25 15 26 ,
25 14 15 26 ,
}

Suite concurrent, Test test1 had failures:
    1. wrong1.c:16  - b == 1 [interleave 14, 25, 26, 15]
```

```
    2. wrong1.c:16  - b == 1 [interleave 25, 14, 26, 15]

Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      0      1        0
             asserts      6      6      4      2      n/a

Elapsed time =    0.000 seconds
```

Antes de iniciar la ejecución de las corrutinas, el scheduler calcula las intercalaciones y luego las muestra, debido a que se invoca a la función `CU_thread_show_interleaves()`. Esta función hace que al comienzo del resumen se muestren las secuencias de instrucciones de cada corrutina (`{25, 26}` para g y `{14, 15}` para f), seguidas de una lista con todas las intercalaciones posibles de dichas secuencias.

Luego de ejecutar el conjunto de prueba (con su único caso de prueba), se observa que se realizaron 6 aserciones (la aserción de `f`, ejecutada una vez para cada intercalación), de las cuales fallaron 2. En ambos casos se trató de la misma condición fallida (`(b == 1)` en la línea 16 del archivo fuente wrong1.c) pero con distintas intercalaciones, que son los mostrados entre corchetes.

Es decir que se muestra el punto del programa en el cual la aserción fue falsa, junto con la intercalación con la cual se llegó a ese punto. Por ejemplo, si se sigue la primera intercalación fallida $\{14, 25, 26, 15\}$, se ve que la secuencia de acciones fue:

```
a = 1; /* línea 14 */
a = 1; /* línea 25 */
a = 0; /* línea 26 */
b = a; /* línea 15 */
```

Luego de la cual claramente no vale que `b == 1`. El ejemplo presentado es artificiosamente simple. Sin embargo, no es difícil imaginarse programas concurrentes reales que utilicen el mismo esquema. Por ejemplo, cinco corrutinas que implementen el problema de los filósofos, donde en el medio se introduzcan aserciones que comprueben la propiedad de exclusión mutua de los tenedores usados por los filósofos.

Como se puede observar, CUThread no solo da un mecanismo para ejecutar casos de prueba concurrentes, de manera automática y ayudando a poner de manifiesto errores, sino que luego de la aparición de uno, provee de información sumamente útil a la hora de localizar el lugar en donde se

encuentra así como el camino de ejecución con el cual se llegó a ese punto del programa. Y todo esto independientemente de la propiedad que se esté queriendo verificar.

4.4.3. Limitaciones

El ejemplo presentado, además de ser muy simple, tiene una particularidad: no tiene bucles, y el número de sentencias a ser ejecutado por cada corrutina es determinado y finito. Con lo cual el conjunto de todas las intercalaciones posibles será finito. ¿Pero qué pasa si para la corrutina correspondiente a la función `g` se tuviera, a diferencia del ejemplo anterior, lo siguiente?:

```
CU_THREAD(g)
{
  CU_THREAD_PROLOGUE();

  while (1) {
    CU_STM(a = 1);
    CU_STM(a = 0);
  }

  CU_THREAD_EPILOGUE();
}
```

En este caso, que no es poco común[12], la extensión CUThread simplemente no es aplicable. Esto es debido a la característica dinámica de la misma, al menos en cuanto a la determinación de las intercalaciones. En un programa donde se utilice una corrutina como la `g` mostrada arriba, el proceso de inicialización (cuando se recolectan las secuencias de cada thread) no terminaría nunca (debido a que para inicializar necesita ejecutar todas las corrutinas, incluída `g`, y requiere que éstas terminen) con lo cual ni siquiera comenzaría el proceso de cálculo del conjunto de intercalaciones.

Lo que se puede hacer para manejar algo como lo anterior, es modificar manualmente el caso de prueba para que la secuencia sea finita:

```
CU_THREAD(g)
{
  static int i;
```

[12] Los programas que no paran son muy comunes en programación concurrente.

```
  CU_THREAD_PROLOGUE();

  for (i = 0; i < 5; i++) {
    CU_STM(a = 1);
    CU_STM(a = 0);
  }

  CU_THREAD_EPILOGUE();
}
```

Ahora sí, CUThread podrá calcular y ejecutar todas las intercalaciones posibles. El reporte generado con la última versión de la g anterior puede verse en el apéndice 5.1.

Si se considera que el número de iteraciones elegido para un bucle es demasiado pequeño como para brindar confianza acerca de la correctitud del programa, siempre se podrá incrementar dicho número, pues lo único que se requiere es que sea finito. El recálculo de las intercalaciones y la re-ejecución de los casos resultantes insumirá tiempo de cómputo, pero no mayor esfuerzo por parte de los desarrolladores que dirigen las pruebas.

Finalmente, otras de las limitaciones es la explosión de estados resultantes de generar todas las intercalaciones posibles de las sentencias atómicas. Para solucionar esto, sería de gran utilidad el contar con algún mecanismo que permita seleccionar un subconjunto del conjunto de intercalaciones que sea relevante a la verificación de la propiedad que se quiera probar. Pero como ya se mencionó, en el presente trabajo no se desarrolla dicho mecanismo, aunque sí se analizan algunas de las características y de los problemas se encontrarían al tratar de desarrollarlo.

4.4.4. Dificultades encontradas

Como ya se dijo, no fue posible alcanzar el objetivo de brindar un mecanismo para extraer, solo de una especificación de un caso de prueba, las intercalaciones relevantes a los fines de verificar una cierta propiedad.

La dificultad fundamental radica en que en la especificación se tiene un modelo del sistema que estará comúnmente a un mayor nivel de abstracción que el programa final (la implementación). Las técnicas de derivación de casos de prueba a partir de una especificación trabajan en general en dos etapas: primero derivan a partir de ésta, mediante algún tipo de transformación, casos de prueba *a nivel de la especificación*, es decir, casos que contienen objetos que existen en la especificación. Luego, estos casos de prueba son

llevados, mediante un proceso de *refinamiento*, a casos de prueba concretos a nivel de la implementación.

El punto clave de la cuestión es que las propiedades de los sistemas concurrentes modelados mediante secuencias entrelazadas (intercaladas) de acciones atómicas, en general no se preservan bajo refinamiento, como se vió en la sección 3.2.7, o sea cuando acciones que en un cierto nivel se consideraban atómicas, luego en un nivel inferior de abstracción pasan a ser estructuras compuestas a su vez por varias acciones.

Como consecuencia de esto, para que en el entorno desarrollado se pudiesen inferir las intercalaciones relevantes a partir únicamente de la especificación, ésta debería incluír todas las acciones atómicas que aparecen en la implementación. Esto claramente no sería adecuado, ya que se estaría bajando el nivel de la especificación al nivel de la implementación.

Entonces el camino alternativo al enfoque puramente funcional anterior, sería el de extraer las intercalaciones relevantes a partir de la implementación final, mediante un análisis estructural del programa [13]. Sin embargo, el objetivo fijado en el proyecto de tesina consistía en extraer las intercalaciones relevantes mediante una técnica de testing funcional. Lo que probablemente se podría desarrollar es una técnica "híbrida", como se sugiere en el próximo capítulo en la sección Trabajos Futuros.

4.5. Otras opciones analizadas

Ahora que ya se ha expuesto el entorno desarrollado, el lector podría preguntarse por qué se concretó de esa manera en particular, como extensión a CUnit y mediante el uso de corrutinas Protothread y de un marcado explícito de sentencias atómicas. A eso puede responderse que durante el desarrollo de la tesina se analizaron otras posibilidades, que se presentan a continuación, junto con las razones por las cuales fueron descartadas en última instancia.

Vault. Un enfoque que se podría haber utilizado es el de extender el lenguaje C con anotaciones específicas para testing de programas concurrentes, a la manera de Vault[12], que si bien no fue hecho para dar soporte a este tipo de testing, sus mecanismos podrían haber sido utilizados para ello.

Vault es una extensión al lenguaje C desarrollada en Microsoft Research que permite al programador describir reglas de manejo de recursos de cómputo de manera tal que un compilador pueda verificar estáticamente que éstas se cumplen. Se puede especificar que ciertas operaciones sean ejecutadas en

[13] Para ejemplos de algunas técnicas de este tipo, se puede ver la sección 3.5.1.

cierto orden, que ciertas operaciones hayan sido realizadas antes de acceder a determinado objeto de datos, y otras cosas más. Para llevar a cabo esto, Vault extiende el sistema de tipos de C añadiendo *guardas de tipo* [14], que es básicamente un predicado que impone una condición al uso de un valor de un tipo dado. Así como los tipos del programa describirán *qué* operaciones son válidas sobre un dato de ese tipo, una guarda de tipo indicará *cuándo* una operación es válida sobre un dato del tipo dado.

Mediante esas extensiones, se marcan en el programa los recursos y las restricciones sobre los mismos, y luego un compilador de Vault comprueba estáticamente que dichas restricciones sean respetadas. El sistema de tipos es lo suficientemente rico como para permitir expresar restricciones de una gran variedad de programas no triviales, aunque teniendo en cuenta la decidibilidad. Por ejemplo, los invariantes de loops deberán ser declarados explícitamente, a menos que puedan ser inferidos en un número fijo de iteraciones.

Sin entrar en detalles sobre cómo trabaja Vault y sobre sus características ventajosas, lo interesante con respecto a este *framework* es que ya provee lo necesario para marcar recursos, que en el caso de testing de programas concurrentes serían recursos compartidos entre distintos threads, y luego las restricciones sobre estos recursos serían las relevantes a la concurrencia (ausencia de deadlock, exclusión mutua) y se harían explícitas mediante los mecanismos ya definidos en Vault. Esto permitiría comprobar estáticamente todo lo que sea computable (decidible), pero luego, para las cuestiones no decidibles[15] aún seguiría siendo útil el "marcado" previo de los recursos, ya que en base a ellos y a la estructura del programa, se podrían derivar todas las intercalaciones que sean relevantes al manejo de esos recursos que respeten las restricciones dadas, *y solo ellos*, y entonces se podría tratar de comprobarlos (aunque no verificarlos) mediante la ejecución del programa en ese conjunto reducido de intercalaciones.

Es decir, a diferencia del entorno presentado en este trabajo, habría una forma automática de extraer solo las intercalaciones relevantes a las características que se quieren testear, disminuyendo así drásticamente el número de casos de prueba a ejecutar. Y aunque a primera vista parecería que una técnica así sería de tipo estructural, en realidad no lo sería, o lo sería a medias, ya que se podría articular muy bien con la especificación, pues las restricciones sobre los recursos provendrían directamente de ella, con lo cual se podría desarrollar un método de extracción de casos de tipo funcional.

Sin embargo, este enfoque tiene la desventaja de que el programador debe

[14] *Guard types*. No confundir con *type qualifiers*.

[15] Como por ejemplo la restricción de que una operación dada se encuentre en el futuro computacional de un thread dado.

previamente aislar los recursos de interés y luego expresar las restricciones que se les desea imponer. Además, las restricciones estarán escritas en un lenguaje que ya no es el lenguaje original en el que estaba escrito el programa, sino en una extensión del mismo, lo que va en contra de uno de los objetivos del entorno desarrollado. Por otra parte, al ser Vault un software propietario, no se podría modificar, con lo cual se debería haber programado toda la funcionalidad desde cero, aún la que no es relativa al testing de programas concurrentes. Por todas estas razones, se decidió no seguir esta línea de investigación.

Lógica temporal. También se analizó el framework presentado en [15] que consiste en una combinación de modelado orientado a objetos y *workflows*[16], donde las propiedades a ser verificadas se especifican en una lógica temporal definida por los propios autores (una particularmente dirigida a un modelo orientado a objetos). Luego, los casos de prueba se especifican mediante patrones de control[17] expresados en un modelo de ejecución llamado POEM (*Parallel Object Execution Model*), con lo cual la especificación resulta ejecutable. De lo anterior se derivan un conjunto de trazas de ejecución (deterministas), las que son pasadas a un *model checker*, que será el encargado de verificar que se satisfacen las fórmulas lógicas (refinadas). Es por esto que, a pesar del título del artículo, el framework presentado es de model checking, no de testing, términos que no se consideran intercambiables en el presente trabajo, como se aclaró en 2.3.

Aún así, el trabajo mencionado resultó en un principio particularmente interesante, pues permitiría una conexión, quizá automatizable, entre los casos de prueba y la especificación, la cual en CUThread está ausente o al menos está librada a lo que el desarrollador considere apropiado.

Sin embargo, finalmente se descartó, pues la parte que podría resultar más provechosa a los fines del presente trabajo, esto es, la representación de las propiedades deseadas mediante lógica temporal (lo que traería aparejada la conexión con la especificación), es justamente la que no está automatizada en [15]. Allí se dice explícitamente que será el desarrollador el encargado de hacer el refinamiento entre las fórmulas temporales (que contienen estados abstractos) y las fórmulas correspondientes que contengan los tipos de datos del programa final. Es decir que el trabajo citado no brinda ninguna pista

[16] En un modelo de este tipo, el sistema se representa como un conjunto de uno o más workflows, compuestos por *contenedores* y *actividades*. Conceptualmente, las actividades toman objetos de los contenedores, trabajan en ellos y los pasan a otras actividades o los depositan en contenedores.

[17] *Control patterns*, similares a expresiones regulares, pero especialmente adaptados para describir secuencias de acceso a datos.

acerca de cómo se podría automatizar dicho refinamiento o parte de él (ni tampoco el camino inverso, o sea la abstracción desde los resultados de una prueba o el estado final del programa de nuevo hacia la especificación), con lo cual no resuelve la dificultad central de la conexión entre una especificación del programa y la representación de sus casos de prueba de unidad a nivel de código.

5. CONCLUSIONES

A lo largo de seis meses de trabajo se desarrolló un entorno de testing de unidad de programas concurrentes que permite expresar casos de prueba para éstos. Los casos de prueba contienen código extraído de las operaciones del programa junto con aserciones que comprueban si ciertos predicados son verdaderos o falsos en determinados puntos del mismo. El entorno ayuda en la tarea de automatización de la ejecución de los casos así como en la localización de los errores exhibidos por éstos durante la ejecución.

Por supuesto, ninguna herramienta es la solución a todos los problemas, y ésta no es una excepción. El entorno posee limitaciones, como se mencionó en la sección 4.4.3. Sin embargo, la automatización que provee para realizar tareas mecánicas y que no requieran intervención humana, como podría serlo la ejecución de grandes conjuntos de prueba con miles de tests, es de gran ayuda en cualquier proyecto de software, ya que los desarrolladores podrán dedicar su tiempo a aspectos más creativos y que requieren de inteligencia.

Por otra parte, la ayuda que brinda CUThread para la localización de los errores también es muy valiosa. En vez de tener que buscar el punto del error de manera manual mediante debugging, en un programa que es no-determinista (con lo que probablemente se esté rogando que el error se muestre nuevamente en la próxima ejecución), gracias a CUThread se contará con el punto en el cual una aserción dejó de ser verdadera, junto con la secuencia de acciones del programa que llevaron hasta ese punto.

Finalmente, una vez localizado el error, debido a que se puede usar el mismo entorno para ejecutar un caso de prueba en una intercalación dada, el programador podrá comprobar si las soluciones propuestas para corregirlo son efectivas o no, ya que podrá ejecutar tantas veces como desee el mismo caso de prueba en la misma intercalación que originalmente puso de manifiesto el error.

5.1. Trabajos futuros

Una línea de desarrollo que se puede seguir inmediatamente a partir de este trabajo es tratar de alcanzar el tercer objetivo planteado (y no logrado)

con CUThread: el de dar una técnica funcional para extraer las intercalaciones relevantes a la verificación de una propiedad dada.

Un método así será de gran interés cuando se desee ejecutar conjuntos de prueba que contengan tests extensos, o cuando se desee ejecutar conjuntos de prueba en un corto intervalo de tiempo. De no contarse con dicha técnica, se deberá afrontar la dificultad de que el número de intercalaciones sufre de una explosión exponencial, con lo cual se estará restringido a mantener el tamaño de los tests dentro de un cierto rango, o bien se deberán aceptar conjuntos de prueba que tarden mucho tiempo en ejecutarse.

Tal técnica para extraer las intercalaciones a partir solo de la especificación quizá sea irrealizable, como se expuso en 4.4.4, pero lo que sí es posible sin duda es el diseño de una técnica funcional que a partir de la especificación *y guiada por la implementación* extraiga las intercalaciones relevantes. Probablemente la implementación se usaría como guía de manera similar a como se usa en el testing funcional tradicional cuando se refinan los casos de prueba a nivel de la especificación a casos de prueba a nivel de la implementación.

APÉNDICE

. SALIDAS DE PROGRAMAS

.1. Salida para el programa de la página 64

```
     CUnit - A unit testing framework for C - Version 2.1-0
     http://cunit.sourceforge.net/

{
28 29 28 29 28 29 28 29 28 29 ,
14 15 ,
}
{
28 29 28 29 28 29 28 29 28 29 14 15 ,
28 29 28 29 28 29 28 29 14 28 29 15 ,
28 29 28 29 28 29 14 28 29 28 29 15 ,
28 29 28 29 14 28 29 28 29 28 29 15 ,
28 29 14 28 29 28 29 28 29 28 29 15 ,
14 28 29 28 29 28 29 28 29 28 29 15 ,
28 14 29 28 29 28 29 28 29 28 29 15 ,
28 29 28 14 29 28 29 28 29 28 29 15 ,
28 29 28 29 28 14 29 28 29 28 29 15 ,
28 29 28 29 28 29 28 14 29 28 29 15 ,
28 29 28 29 28 29 28 29 28 14 29 15 ,
28 29 28 29 28 29 28 29 14 15 28 29 ,
28 29 28 29 28 29 14 28 29 15 28 29 ,
28 29 28 29 14 28 29 28 29 15 28 29 ,
28 29 14 28 29 28 29 28 29 15 28 29 ,
14 28 29 28 29 28 29 28 29 15 28 29 ,
28 14 29 28 29 28 29 28 29 15 28 29 ,
28 29 28 14 29 28 29 28 29 15 28 29 ,
28 29 28 29 28 14 29 28 29 15 28 29 ,
28 29 28 29 28 29 28 14 29 15 28 29 ,
28 29 28 29 28 29 14 15 28 29 28 29 ,
28 29 28 29 14 28 29 15 28 29 28 29 ,
28 29 14 28 29 28 29 15 28 29 28 29 ,
14 28 29 28 29 28 29 15 28 29 28 29 ,
28 14 29 28 29 28 29 15 28 29 28 29 ,
28 29 28 14 29 28 29 15 28 29 28 29 ,
28 29 28 29 28 14 29 15 28 29 28 29 ,
28 29 28 29 14 15 28 29 28 29 28 29 ,
28 29 14 28 29 15 28 29 28 29 28 29 ,
14 28 29 28 29 15 28 29 28 29 28 29 ,
28 14 29 28 29 15 28 29 28 29 28 29 ,
28 29 28 14 29 15 28 29 28 29 28 29 ,
28 29 14 15 28 29 28 29 28 29 28 29 ,
14 28 29 15 28 29 28 29 28 29 28 29 ,
28 14 29 15 28 29 28 29 28 29 28 29 ,
14 15 28 29 28 29 28 29 28 29 28 29 ,
14 28 15 29 28 29 28 29 28 29 28 29 ,
```

```
28 14 15 29 28 29 28 29 28 29 28 29 ,
28 29 14 28 15 29 28 29 28 29 28 29 ,
14 28 29 28 15 29 28 29 28 29 28 29 ,
28 14 29 28 15 29 28 29 28 29 28 29 ,
28 29 28 14 15 29 28 29 28 29 28 29 ,
28 29 28 29 14 28 15 29 28 29 28 29 ,
28 29 14 28 29 28 15 29 28 29 28 29 ,
14 28 29 28 29 28 15 29 28 29 28 29 ,
28 14 29 28 29 28 15 29 28 29 28 29 ,
28 29 28 14 29 28 15 29 28 29 28 29 ,
28 29 28 29 28 14 15 29 28 29 28 29 ,
28 29 28 29 28 29 14 28 15 29 28 29 ,
28 29 28 29 14 28 29 28 15 29 28 29 ,
28 29 14 28 29 28 29 28 15 29 28 29 ,
14 28 29 28 29 28 29 28 15 29 28 29 ,
28 14 29 28 29 28 29 28 15 29 28 29 ,
28 29 28 14 29 28 29 28 15 29 28 29 ,
28 29 28 29 28 14 29 28 15 29 28 29 ,
28 29 28 29 28 29 28 14 15 29 28 29 ,
28 29 28 29 28 29 28 29 14 28 15 29 ,
28 29 28 29 28 29 14 28 29 28 15 29 ,
28 29 28 29 14 28 29 28 29 28 15 29 ,
28 29 14 28 29 28 29 28 29 28 15 29 ,
14 28 29 28 29 28 29 28 29 28 15 29 ,
28 14 29 28 29 28 29 28 29 28 15 29 ,
28 29 28 14 29 28 29 28 29 28 15 29 ,
28 29 28 29 28 14 29 28 29 28 15 29 ,
28 29 28 29 28 29 28 14 29 28 15 29 ,
28 29 28 29 28 29 28 29 28 14 15 29 ,
}

Suite concurrent, Test test1 had failures:
    1. wrong2.c:16  - b == 1 [interleave 28, 29, 28, 29, 28, 29, 28, 29, 14, 28, 29, 15]
    2. wrong2.c:16  - b == 1 [interleave 28, 29, 28, 29, 28, 29, 14, 28, 29, 28, 29, 15]
    3. wrong2.c:16  - b == 1 [interleave 28, 29, 28, 29, 14, 28, 29, 28, 29, 28, 29, 15]
    4. wrong2.c:16  - b == 1 [interleave 28, 29, 14, 28, 29, 28, 29, 28, 29, 28, 29, 15]
    5. wrong2.c:16  - b == 1 [interleave 14, 28, 29, 28, 29, 28, 29, 28, 29, 28, 29, 15]
    6. wrong2.c:16  - b == 1 [interleave 28, 14, 29, 28, 29, 28, 29, 28, 29, 28, 29, 15]
    7. wrong2.c:16  - b == 1 [interleave 28, 29, 28, 14, 29, 28, 29, 28, 29, 28, 29, 15]
    8. wrong2.c:16  - b == 1 [interleave 28, 29, 28, 29, 28, 14, 29, 28, 29, 28, 29, 15]
    9. wrong2.c:16  - b == 1 [interleave 28, 29, 28, 29, 28, 29, 28, 14, 29, 28, 29, 15]
    10. wrong2.c:16  - b == 1 [interleave 28, 29, 28, 29, 28, 29, 28, 29, 28, 14, 29, 15]
    11. wrong2.c:16  - b == 1 [interleave 28, 29, 28, 29, 28, 29, 14, 28, 29, 15]
    12. wrong2.c:16  - b == 1 [interleave 28, 29, 28, 29, 14, 28, 29, 28, 29, 15]
    13. wrong2.c:16  - b == 1 [interleave 28, 29, 14, 28, 29, 28, 29, 28, 29, 15]
    14. wrong2.c:16  - b == 1 [interleave 14, 28, 29, 28, 29, 28, 29, 28, 29, 15]
    15. wrong2.c:16  - b == 1 [interleave 28, 14, 29, 28, 29, 28, 29, 28, 29, 15]
    16. wrong2.c:16  - b == 1 [interleave 28, 29, 28, 14, 29, 28, 29, 28, 29, 15]
    17. wrong2.c:16  - b == 1 [interleave 28, 29, 28, 29, 28, 14, 29, 28, 29, 15]
    18. wrong2.c:16  - b == 1 [interleave 28, 29, 28, 29, 28, 29, 28, 14, 29, 15]
    19. wrong2.c:16  - b == 1 [interleave 28, 29, 28, 29, 14, 28, 29, 15]
    20. wrong2.c:16  - b == 1 [interleave 28, 29, 14, 28, 29, 28, 29, 15]
    21. wrong2.c:16  - b == 1 [interleave 14, 28, 29, 28, 29, 28, 29, 15]
    22. wrong2.c:16  - b == 1 [interleave 28, 14, 29, 28, 29, 28, 29, 15]
    23. wrong2.c:16  - b == 1 [interleave 28, 29, 28, 14, 29, 28, 29, 15]
    24. wrong2.c:16  - b == 1 [interleave 28, 29, 28, 29, 28, 14, 29, 15]
    25. wrong2.c:16  - b == 1 [interleave 28, 29, 14, 28, 29, 15]
    26. wrong2.c:16  - b == 1 [interleave 14, 28, 29, 28, 29, 15]
    27. wrong2.c:16  - b == 1 [interleave 28, 14, 29, 28, 29, 15]
    28. wrong2.c:16  - b == 1 [interleave 28, 29, 28, 14, 29, 15]
    29. wrong2.c:16  - b == 1 [interleave 14, 28, 29, 15]
    30. wrong2.c:16  - b == 1 [interleave 28, 14, 29, 15]
```

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      1      1      0      1        0
             asserts     66     66     36     30      n/a

Elapsed time =    1.150 seconds
```

BIBLIOGRAFíA

[1] Gregory R. Andrews. *Multithreaded, Parallel, and Distributed Programming.* Addison-Wesley, 2000.

[2] C. Artho. Finding faults in multi-threaded programs, 2001.

[3] D. Bruening. Systematic testing of multithreaded Java programs, 1999.

[4] Alan Burns and Geoff Davies. *Concurrent Programming.* Addison-Wesley, 1993.

[5] David Butenhof. *Programming with POSIX threads.* Addison-Wesley, 1997.

[6] J. Choi and A. Zeller. Isolating failure-inducing thread schedules, 2002.

[7] Jong-Deok Choi and Harini Srinivasan. Deterministic replay of Java multithreaded applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools, Welches, Oregon*, pages 48–59, August 1998.

[8] Mark Christiaens, Michiel Ronsse, and Jong-Deok Choi. Record/replay in the presence of benign data races.

[9] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *ACM SIGPLAN Notices*, 35(9):268–279, 2000.

[10] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems, Concepts and Design.* Pearson Education, third edition, 2001.

[11] J. C. Cunha, J. Lourenço, J. Vieira, B. Mosc ao, and D. Pereira. A framework to support parallel and distributed debugging. *Lecture Notes in Computer Science*, 1401:708+, 1998.

[12] Robert DeLine and Manuel Fahndrich. Enforcing high-level protocols in low-level software. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 59–69, 2001.

[13] Tom Duff. Message at comp.lang.c newsgroup. Message-ID 8144@alice.UUCP., August 1988.

[14] Adam Dunkels and Oliver Schmidt. Protothreads library, *http://www.sics.se/∼adam/pt/* .

[15] Maximilian Frey and Michael Oberhuber. Testing parallel and distributed programs with temporal logic specifications, 1997.

[16] Maximilian Frey, Michael Oberhuber, and Markus Podolsky. Framework for testing based development of parallel and distributed programs. In *PDSE*, pages 246–253, 1998.

[17] Guillermo L. Grinblat. Uso de la forma SSU en la alocación de registros. Tesina de Grado. Universidad Nacional de Rosario, Argentina, 2006.

[18] Kuo-Chung Tai Gwan-Hwan Hwang and Ting-Lu Hunag. Reachability testing: An approach to testing concurrent software. *International Journal of Software Engineering and Knowledge Engineering*, 5(4), December 1995.

[19] Hans-Martin Horcher and Jan Peleska. Using formal specifications to support software testing. *Software Quality Journal*, 4(4):309–327, 1995.

[20] Joel Huselius. Debugging Parallel Systems: A State of the Art Report. Technical Report 63, Mälardalen University, Department of Computer Science and Engineering, September 2002.

[21] James R. Larus, Thomas Ball, Manuvi Das, Manuel Fähndrich, Jon Pincus, Sriram K. Rajamani, and Ramanathan Venkatapathy. Righting software. *IEEE Software*, 21(3):92–100, May/June 2004.

[22] Yuejian Li and Nancy J. Wahl. An overview of regression testing. *Software Engineering Notes, ACM SIGSOFT*, 24(1):19–73, January 1999.

[23] Dino Mandrioli, Carlo Ghezzi, and Mehdi Jazayeri. *Fundamentals of Software Engineering.* Prentice Hall Professional Technical Reference, 1991.

[24] Atif M. Memon. GUI testing: pitfalls and process. *IEEE Computer*, 35(8):87–91, August 2002.

[25] Félix Nanclares Echarri. Fundamentos del proceso de testing. Talk given at the National Institute of Industrial Technology, Argentina, 2006.

[26] Ana M. Moreno Natalia Juristo and Wolfgang Strigel. Software testing practices in industry. *IEEE Software*, 23(4):19–21, July/August 2006.

[27] Robert H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *Proceedings of ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 1–11, San Diego, California, 1993.

[28] Jeff Offutt, Yiwei Xiong, and Shaoying Liu. Criteria for generating specification-based tests. In IEEE Computer Society Press, editor, *Proceedings of the Fifth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '99), Las Vegas, NV*, pages 119–131, October 1999.

[29] J. Peleska and M. Siegel. Test automation of safety-critical reactive systems. *South African Computer Journal*, 19:53–77, 1997.

[30] Shari Lawrence Pfleeger. *Software Engineering.* Pearson Education, 2002.

[31] Roger Pressman. *Software Engineering.* McGraw-Hill, 1998.

[32] S. Rapps and EJ Weyuker. Data flow analysis techniques for program test data selection. In *Proceedings of the 6th International Conference on Software Engineering*, pages 272–278, September 1982.

[33] Cherniavsky J. C. Richards Adrion W., Branstad M. A. Validation, verification, and testing of computer software. *ACM Computing Surveys*, 11(2):159–192, June 1982.

[34] Michiel Ronsse, Koenraad De Bosschere, and Jacques Chassin de Kergommeaux. Execution replay and debugging. In *Automated and Algorithmic Debugging*, 2000.

[35] Per Runeson. A survey of unit testing practices. *IEEE Software*, 23(4):22–29, July/August 2006.

[36] Mark Russinovich and Bryce Cogswell. Replay for concurrent non-deterministic shared-memory applications. In *PLDI '96: Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, pages 258–266, New York, NY, USA, 1996. ACM Press.

[37] Viktor Schuppan, Marcel Baur, and Armin Biere. JVM independent replay in Java. In Klaus Havelund and Grigore Rosu, editors, *Proceedings of the Fourth Workshop in Runtime Verification, RV 2004, Barcelona, Spain*, pages 76–94, April 2004.

[38] Ian Sommerville. *Software Engineering.* Pearson Education, 2005.

[39] P. Stocks and D. Carrington. A framework for specification-based testing. *IEEE Transactions on Software Engineering*, 22(11):777–793, November 1996.

[40] K. Tai. Theory of fault-based predicate testing for computer programs. *IEEE Transactions on Software Engineering*, 22(8):552–562, August 1996.

[41] Simon Tatham. Coroutines in C, *http://www.chiark.greenend.org.uk/ sgtatham/coroutines.html*.

[42] H. Thane and H. Hansson. Testing distributed real-time systems. (24):463–478, 2001.

[43] Rob J. van Glabbeek and Ursula Goltz. Refinement of actions and equivalence notions for concurrent systems. *Acta Informatica*, 37(4/5):229–327, 2001.

[44] S. Vilkomir and J. Bowen. Formalization of software testing criteria using the Z notation. In *Proceedings of the 25th International Computer Software and Applications Conference (COMPSAC)*, 2001.

[45] Jeanette M. Wing. Handout 12 of the course *Models of Software Systems*, Carnegie Mellon University, 1995.

[46] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22th International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, 1995.

[47] C. Yang and L. Pollock. The challenges in automated testing of multithreaded programs. In *Proceedings of the 14th International Conference on Testing Computer Software*, pages 157–166, June 1997.

[48] Cheer-Sun Yang. Identifying redundant test cases for testing parallel language constructs, 1997.

[49] Cheer-Sun D. Yang, Arnie L. Souter, and Lori L. Pollock. All-du-path coverage for parallel programs. In *International Symposium on Software Testing and Analysis*, pages 153–162, 1998.