DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

UNIVERSIDAD POLITÉCNICA DE VALENCIA

P.O. Box: 22012 E-46071 Valencia (SPAIN)

Documentación Docente

Asignatura: Herramientas Avanzadas para el Desarrollo de Software
Temas: 1 a 4
Curso: Quinto
Centro: Facultad de Informática

Ref. No: DSIC-DD/02/06 **Versión:** 1.0 **Páginas:** 134
Título: Apuntes de la Asignatura Herramientas Avanzadas para el Desarrollo de Software
Autor (es): Alicia Villanueva García
Fecha: 24 de Agosto de 2006
Materia: Desarrollo de Software

[X] Teoría
[ ] Problemas
[ ] Prácticas de laboratorio

VºBº **Autor (es):**
Responsable de la Unidad Docente

Universidad Politécnica de Valencia
Departamento de Sistemas Informáticos y Computación

Apuntes de la asignatura

Herramientas Avanzadas para el Desarrollo de Software

Autor:
Alicia Villanueva

Agosto 2006

Correo Electrónico del autor: villanue@dsic.upv.es

Dirección del autor:

Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera, s/n
46022 Valencia
España

Índice general

Índice de figuras

Índice de cuadros

Introducción

0.1. Historia y Motivación

El desarrollo de *software* es una de las actividades más importantes y complejas em la informática [Pre97]. Hoy en día nadie discute la gran importancia que tiene desarrollar *software* de calidad y por ello la necesidad de usar formalismos o técnicas que permitan a los programadores alcanzar el grado de exigencia requerido por el cliente. Dichos formalismos van desde los más intuitivos como el *software testing* hasta los más formales como la demostración de teoremas (*theorem proving*).

Durante este curso veremos algunas técnicas que pueden ayudar al programador a mejorar la calidad de su producto, tanto en términos de tiempo empleado para su desarrollo, como en optimización del mismo producto. Otro objetivo de estas técnicas debería ser el de facilitar las cosas al programador (reduciendo así el coste de desarrollo), por ello nos vamos a centrar en técnicas que hacen un especial hincapié en la *automatización* del proceso. Es decir, técnicas que *hagan el trabajo solas*.

Tras la crisis del *software*, la comunidad se convenció de la necesidad de usar técnicas y herramientas que permitieran garantizar una cierta calidad o seguridad en los programas. Los investigadores fueron quizás demasiado ambiciosos ya que inicialmente su objetivo final era el de poder decir de forma general si un sistema era correcto o no, es decir, si daba los resultados esperados sin provocar ninguna catástrofe. Existen herramientas muy potentes que se acercan a dicho objetivo en determinados casos particulares, pero estas herramientas, o bien son demasiado costosas ya que tardan demasiado tiempo en dar una respuesta, o bien a veces sólo un experto es capaz de manejarlas. Estos son los principales motivos por los que los *métodos formales*, es decir, aquéllos que están basados en algún campo matemático y por lo tanto dan una respuesta *segura*, no tuvieron a nivel industrial el éxito esperado desde un principio.

De los métodos formales tradicionales se ha evolucionado a una versión de la verificación menos ambiciosa. Consiste en lo que en inglés es denominado *lightweigh formal methods*, es decir métodos formales ágiles. Estos métodos se caracterizan por querer cubrir unos objetivos mucho más humildes, por ejemplo restringiéndose a la verificación de un tipo de características, limitándose a un único lenguaje de especificación (o incluso a sólo una parte del mismo), etc. Son métodos por tanto *parciales* y esto conlleva que sean más eficientes y puedan automatizarse de forma más sencilla.

Por supuesto, existen también técnicas no formales pero que en ningún caso debemos despreciar por su eficacia y la capacidad de complementar otras herramientas. El ejemplo más claro podría ser el del *software testing* (pruebas de programas), que permite a los programadores comprobar la calidad del producto que están desarrollando [Mye83]. Existen numerosos tipos de pruebas y de métodos que cubren distintos objetivos, como puede ser el de comprobar que el resultado obtenido coincide con el esperado en el mayor número de casos (trazas) posible (idealmente en todos los casos posibles). La tarea de probar los programas es larga y a veces muy complicada, por ello existen herramientas y técnicas que facilitan ese trabajo y es en estas herramientas en las que nosotros estamos interesados.

Uno de los problemas más importantes que cualquier herramienta de verificación, análisis o prueba tiene que afrontar, es el tamaño de los programas, de sus modelos o de las estructuras intermedias que cada técnica pueda construirse. Éste es un problema común a prácticamente cualquier marco de trabajo o paradigma que podamos encontrar relacionado con la ingeniería de *software* y, por lo tanto, ha sido estudiado en profundidad a lo largo de los años. Actualmente existen técnicas que permiten manejar o mitigar el problema (nunca solucionarlo completamente), y una de éstas técnicas es la *interpretación abstracta* (*abstract interpretation*), con la que un espacio de estados gigantesco puede ser reducido a sólo unos cuantos *meta-estados*, convirtiendo de esta forma un problema prácticamente inabordable, en un problema manejable. La interpretación abstracta no es un método para el desarrollo de software en sí, sino una técnica que, aplicada a dichos métodos, permite crear herramientas más eficientes a cualquier nivel de desarrollo. Como veremos, con la interpretación abstracta obtendremos siempre una herramienta ágil, ya que introduce de forma natural *parcialidad*. La idea general es que en vez de analizar un programa, lo que se hará será analizar una *versión abstracta* del mismo. Dicha versión abstracta habrá perdido parte de información del sistema, por lo que en un momento dado, puede ocurrir que la característica que queramos comprobar sea precisamente la que hemos "borrado". En estos casos nuestra herramienta normalmente nos contesta con un *no sé*, pero siempre tendremos la posibilidad de refinar y recuperar algo de información del modelo original.

Como ejemplos de técnicas que permiten mejorar la calidad del *software* tenemos una versión *abstracta* de la técnica de *model checking*, una versión abstracta de la depuración (*debugging*), del análisis estático, etc. Estas técnicas se usan a nivel industrial en sistemas reales. Por ejemplo, la marca FIAT usa un *software* que, de forma completamente transparente al usuario, usa la interpretación abstracta para detectar si el sistema de navegación de sus coches puede llegar a un error por *overflow*. También mentionaremos el proyecto francés ASTREÈ que, aunque está siendo desarrollado por investigadores de los centros más prestigiosos del país, cuenta con un importante apoyo industrial e institucional.

En resumen, este documento docente pretende mostrar una serie de herramientas o técnicas que pueden ser usadas para aumentar la calidad de un producto determinado. Dichas técnicas pueden ser formales o no. Cada una tiene una particularidad distinta y cubre un objetivo diferente.

0.2. Software Testing

Las pruebas de *software*, además de permitir analizar la calidad del producto desarrollado, también son una herramienta para mejorar dicha calidad, además de facilitar o agilizar el desarrollo del producto final siempre que hagamos las pruebas en etapas tempranas del ciclo de vida del desarrollo de *software*. El hecho de realizar una prueba de un programa implica de forma automática que queremos **detectar** posibles **fallos** existentes en el código. Una prueba en pocas palabras es la ejecución de un conjunto de casos de prueba y el análisis de los resultados obtenidos. Decimos que cuanto mayor probabilidad tenga un caso de prueba de encontrar un fallo, mejor será dicho caso de prueba de calidad. Una de las tareas más complicadas cuando queremos realizar pruebas de *software* es la del diseño de casos de prueba, es decir, determinar cuál es el conjunto de casos de prueba que compondrán la prueba del sistema de forma que se maximice la probabilidad de detectar errores en el código. Esta tarea puede llegar a ocupar cerca del 40 % del tiempo total

de desarrollo del producto [DT02]. Además, debemos tener en cuenta que puede que no queramos probar sólo un aspecto de nuestro programa, sino varias cuestiones (corrección, eficiencia, amigabilidad, etc.), lo que hace que tengamos que diseñar una prueba para cada uno de esos aspectos.

Existen muchos tipos de pruebas: unitarias, de integración, de aceptación, de regresión, de carga, *alpha*, *beta*, etc. y en general, algunas son más adecuadas para el estudio de una serie de características que otras. Para pruebas *unitarias*, existen dos enfoques fundamentales: el de caja blanca y el de caja negra. Estos dos enfoques se pueden combinar para obtener una técnica más efectiva.

En las **pruebas de caja blanca** usamos la estructura de control del programa para derivar distintos casos de prueba. El objetivo final debería ser el de cubrir todos los caminos posibles en la ejecución del programa usando el menor número de casos de prueba. Existen distintos criterios de cobertura que pueden ser usados para guiar y evaluar la generación de casos de prueba, y por tanto la prueba de programas. Dependiendo del caso de prueba usado, tendremos un conjunto de casos de prueba diferente para ejecutar. Por ejemplo, una de las técnicas usadas para determinar los casos de prueba es la del *camino básico*, cuyo objetivo es la cobertura de ramas. La idea de esta técnica es derivar casos de prueba a partir de un conjunto dado de caminos independientes, los cuales se obtienen a partir del *grafo de flujo* del programa.

Por otro lado, las **pruebas de caja negra** se centran en los *requisitos funcionales* del *software*. El objetivo que se suele perseguir cuando ejecutamos este tipo de pruebas es el de determinar la calidad *funcional* del *software*, es decir, determinar si existen funciones incorrectas, o incluso si una función que debería estar definida no lo está. También es posible detectar errores relativos a la interfaz de la aplicación, errores en estructuras de datos o en accesos a bases de datos externas, errores debidos al rendimiento, de inicialización o de terminación.

0.2.1. Generación automática de casos de prueba

Ya hemos mencionado que el diseño de casos de prueba es una de las tareas más laboriosas en el *software testing* [Myn90]. Por ello son importante las técnicas de generación automática de casos de prueba: menor coste en las pruebas implica mayor calidad de *software*, a la vez que menor coste en el desarrollo del producto. Las técnicas para la generación de pruebas (ya sean manuales o automáticas) intentan siempre encontrar un conjunto mínimo de pruebas sin pérdida de eficacia. Existen métodos de generación automática de casos basados en metaheurísticas tales como los *algoritmos genéticos*, o la *búsqueda tabú*. Hablamos de metaheurísticas debido principalmente al hecho de que (en general) el número casos de pruebas necesarios para probar que un programa no contiene errores es infinito y por tanto necesitamos diseñar métodos aproximados. Éste es también el motivo por el que el *software testing* no es una herramienta formal, sino semi-formal.

Existen varias aproximaciones usadas en los métodos de generación automática de pruebas. Tenemos por ejemplo la *generación aleatoria*, la generación de test simbólicos (técnicas estáticas), y la generación dinámica de pruebas. Para la generación dinámica de pruebas tenemos a su vez tres distintos caminos: usar los algoritmos genéticos (ésta es la técnica más usada), el recocido simulado o la búsqueda tabú. El Capítulo 1 de este documento está dedicado en parte a profundizar en algunas de estas técnicas.

0.3. Interpretación Abstracta

La interpretación abstracta no es más que una técnica que permite optimizar algoritmos y métodos que implementan problemas demasiado complejos, y por tanto aplicados sin simplificación resultan poco efectivos. La idea principal es la de obtener un modelo más sencillo que resulte manejable por los algoritmos y las máquinas actuales. Esta aproximación resulta útil cuando nos encontramos con la necesidad de ejecutar un algoritmo que resuelve un problema pero cuyo coste temporal es completamente inadmisible, bien porque necesitemos una respuesta antes, bien porque no tenemos una máquina lo suficientemente potente para soportar el cómputo (por ejemplo por falta de memoria).

El objetivo final cuando aplicamos la técnica de la interpretación abstracta es el de poder realizar la computación en un modelo simplificado pero de forma que se garantice que la respuesta obtenida con el modelo simplificado es válidad también para el modelo original. Al simplificar el modelo se asume que se va a perder cierta información relacionada con el sistema. De hecho, puede darse el caso en el que el nuevo modelo no considera ciertas variables ya que se creía que no afectaban a la comprobación comprobación que se quiere hacer. Si efectivamente esas variables no eran necesarias para dar una respuesta, entonces todo saldrá bien y el algoritmo aplicado al modelo simplificado dará una respuesta que será válida para el sistema real (no simplificado). Sin embargo, si una de las variables ignoradas era necesaria para la comprobación del sistema, entonces la herramienta probablemente nos dirá que no tiene información suficiente para obtener un resultado que sea fiel al modelo inicial. Si esto ocurriera, existen técnicas que permiten obtener un modelo intermedio (ni tan complejo ni tan reducido) de forma automática y con el que posiblemente obtengamos un buen resultado.

Ejemplo 0.3.1 Supongamos que, por ejemplo, tenemos una función `suma(N,M)` que calcula la suma de los dos enteros `N` y `M`. Sabemos que el resultado de esta operación va a ser siempre un número par si ambos números son pares o impares, y el resultado va a ser impar si cada número es de una paridad distinta. Para comprobar si nuestra función está (en principio) bien implementada, podríamos intentar comprobar si de verdad nuestra función se comporta de esta forma. Para comprobar que la paridad del resultado es correcta, en realidad no necesitamos saber el valor de cada número introducido. De hecho nos bastaría saber de qué aridad es cada dato de entrada, así que lo que se hará será definir una abstracción para los datos y a continuación una función abstracta *suma* que maneje la versión abstracta de los datos de entrada.

La Figura 1 debe interpretarse como sigue: El conjunto que se muestra a la izquierda representa los valores *reales* de las variables del sistema. En este caso son todos los valores que puede tomar una variable entera. El conjunto que se muestra a la derecha son los valores de las variables en el modelo simplificado, en este caso su paridad. Las flechas indican la relación que existe entre un conjunto de la izquierda (real) con uno de la derecha (reducido). Las flechas que van de izquierda a derecha indican para cada entero, qué elemento del conjunto de la derecha va a *representarle*. Por ejemplo al número 1 le va a representar el elemento `impar`. Por otro lado, las flechas que van de derecha a izquierda indican qué elementos reales están siendo representados por cada cada elemento del conjunto reducido. Por ejemplo, el elemento `par` representará a los elementos {`2,4,6,...`} del conjunto de la izquierda.

La figura ilustra la relación establecida entre el dominio original y el dominio abstracto

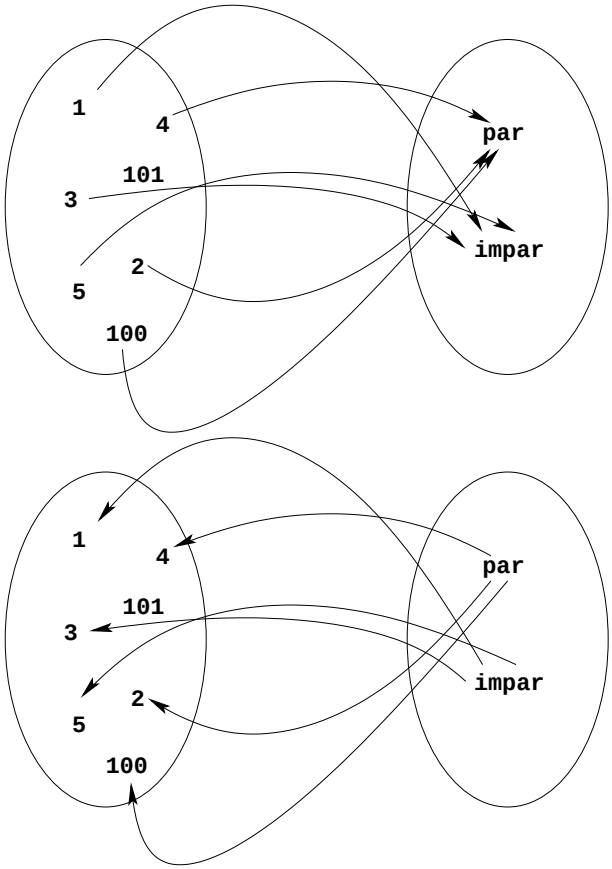


Figura 1: Abstracción y concreción de valores

(o simplificado). En cuanto a la definición de la versión abstracta de la operación suma, decimos que la función *suma* tendrá el siguiente comportamiento:

- suma(`par`,`par`) = `par`
- suma(`impar`,`impar`) = `par`
- suma(`par`,`impar`) = `impar`
- suma(`impar`),`par`) = `impar`

Si, dado dos enteros del dominio concreto `N` y `M`, denotamos sus representantes en el dominio abstracto como abs(`N`) y abs(`M`) respectivamente, entonces podremos calcular la operación abstracta correspondiente a los elementos concretos de la siguiente forma suma(abs(`N`), abs(`M`)). Obviamente, para que este tipo de operación no pierda significado efectivo, las funciones que relacionan los dos dominios deben cumplir una serie de propiedades que veremos en profundidad en el Capítulo 2.

■

Nótese que con este mecanismo, el comprobar el buen funcionamiento de la función se limita a realizar 4 pruebas, una para cada caso mientras que la misma comprobación en el sistema original (sin la abstracción), necesitaría un número de pruebas infinito ya que tendríamos que realizar una prueba para cada valor natural de cada variable del sistema.

0.4. *Model Checking* abstracto

El *model checking* es una técnica completamente automática que permite comprobar si una propiedad determinada es satisfecha o no por un sistema. La idea original sobre la que se definió la técnica era la exploración *exhaustiva* del espacio de estados del sistema, lo que implica un alto coste computacional, tanto en espacio como temporal. Existen numerosas metodologías que reducen el coste aplicando distintas optimizaciones como, por ejemplo, la versión simbólica del algoritmo que usa la teoría de los OBDD's (*ordered binary decision diagrams*) para obtener algoritmos más eficientes. Sin embargo, ninguna técnica es *la mejor* ya que todas sufren de distintos inconvenientes. Siguiendo con el ejemplo, la técnica simbólica no es eficiente cuando se manejan sistemas que no tienen una simetría clara o usan un número muy elevado de variables. Hay que tener en cuenta que, aunque los algoritmos sean mucho más eficientes cuando trabajan con OBDDs, el espacio usado con esta metodología es mucho más grande que el espacio de estados del sistema orginal ya que cada variable y su dominio tiene que ser codificada de forma binaria.

Por este motivo introducimos el *model checking* abstracto, porque en la combinación de las técnicas de optimización está el futuro. De una forma muy intuitiva, el *model checking* abstracto consiste en los siguientes pasos:

1. a partir del modelo del sistema, se obtiene un *modelo abstracto* abstrayendo tanto el dominio como las operaciones del sistema
2. a partir de una propiedad, se obtiene una *propiedad abstracta*
3. se comprueba si el modelo abstracto satisface la propiedad abstracta
   - en caso afirmativo podemos decucir que el modelo original del sistema satisface la propiedad
   - en caso negativo no estamos seguros de si realmente el modelo no satisface la propiedad, ya que el resultado negativo puede deberse a la pérdida de información en la abstracción. Si esto se produce, se puede refinar el modelo y volver al punto 3

Esta técnica tiene una clara ventaja con respecto a la técnica simbólica, y es la posibilidad de poder trabajar con dominios de variables infinitos. Resulta especialmente apropiada cuando tenemos aplicaciones con variables con dominios muy grandes pero que pueden ser divididos en distintos rangos.

Como desventajas, podemos encontrar la relativa dificultad de definir la abstracción adecuada, pero las herramientas existentes consideran e implementan la posibilidad de aplicar abstracciones predefinidas para lo que no es necesario un conocimiento excesivo de la teoría de la interpretación abstracta.

0.5. Análisis abstracto de programas

Un programa puede tener distintas semánticas dependiendo de qué propiedades de las computaciones (*observables*) nos interesen. La elección de un *observable* α determina una *equivalencia de observables* entre programas. Es decir, dos programas son equivalentes $P_1 \equiv_\alpha P_2$ si, y sólo si, P_1 y P_2 son indistinguibles con respecto al observable α. En

otras palabras, el comportamiento de ambos programas es idéntico con respecto a dicho observable. Dependiendo de lo que queramos hacer, elegiremos un observable u otro. Por ejemplo, si nos interesa la entrada-salida de un programa, podemos usar la semántica de *respuestas computadas* y *fallo* pero si lo que queremos hacer es un cierto análisis estático de un programa, quizás necesitemos la información generada durante una computación, es decir, la traza completa.

Relacionado con el tema de las semánticas, es posible definir la semántica de un programa usando un modelo abstracto, es decir, que dicho modelo abstracto sea precisamente la semáncia del programa. La característica más importante de la interpretación abstracta es el hecho de que, una vez modelada la propiedad que nos interesa analizar usando un dominio abstracto (ver ejemplo de la suma), se puede deducir de forma sistemática una semántica abstracta que nos permitirá computar de forma efectiva una aproximación de la propiedad.

0.6. Depuración abstracta

La depuración de programas es una técnica formal orientada a la detección automática de errores. Además puede incluirse una fase posterior de corrección también automática. No hay que confundir un *depurador* con un *tracer*. Éste último es una herramienta que simplemente permite al programador seguir la ejecución de su programa *paso a paso*, o introduciendo *breakpoints*. La idea principal de la depuración de programas o *program debugging* es la de, a partir de una especificación *esperada* del programa, se comprueba si los resultados obtenidos por el programa coinciden con los de dicha especificación. La especificación puede ser dada en distintos formatos, bien siendo un programa alternativo, bien el conjunto de respuestas, etc.

Nótese que esta técnica es capaz de detectar dos tipos de errores: errores debidos a *incorrecciones*, y errores debidos a *insuficiencias*. Una incorrección ocurre cuando el programa calcula algo que no coincide con lo esperado por el programador, mientras que una insuficiencia ocurre cuando programa falla al intentar calcular un resultado esperado por el programador, es decir, cuando no consigue obtener una respuesta.

La versión abstracta de la depuración lo que pretende es poder extender el dominio de aplicación de programas con un comportamiento finito a programas con un comportamiento infinito definiendo, en vez de la semántica esperada, una propiedad del programa.

El depurador no sólo debe decir si se ha producido un error, sino que debe ser capaz de encontrar el punto desde el cual deriva dicho error. En caso de tratarse de una incorrección, debe encontrar el punto en el cual se produce el desencuentro entre lo esperado y lo obtenido y para ello necesitará la información que pueda darle un *oráculo* (posiblemente el programador) en base a preguntas formuladas por el mismo depurador. En relación a los errores derivados de insuficiencias, el depurador intentará reconstruir el proceso hasta el momento en el que se produce el fallo, detectando así dóndo se ha quedado bloqueado el cálculo del resultado.

0.7. Evaluación parcial

La evaluación parcial (o *partial evaluation*) es una técnica de optimización de programas conocida también como *especialización de programas* (o *program specialization*). La

idea intuitiva de la evaluación parcial se basa en el hecho de que a partir de una función que toma como entrada *dos* parámetros, especializándola podemos obtener una función de *un* sólo parámetro. En realidad, el número de parámetros puede ser distinto, es decir, el discurso anterior se puede generalizar manteniendo la idea de que el programa especializado tendrá menos parámetros de entrada que el original. El proceso consistiría en fijar el valor de uno de los dos parámetros y ver así el resultado de la función dependiendo del parámetro no fijado. En el ámbito del análisis a este proceso se le llama *proyección* o *restricción*, mientras que en el ámbito de la lógica, a este proceso se le llama *currificación*. Estos procesos se diferencian de la evaluación parcial en cuanto a que ésta trabaja con *textos de programas* en lugar de trabajar con funciones.

Así pues, un *evaluador parcial* es un algoritmo que, dado un programa y alguno de sus datos de entrada, produce un *residuo* o *programa especializado*. La ejecución del residuo con respecto a los datos de entrada aún no proporcionados, obtendrá el mismo resultado que la ejecución del programa original proporcionándole todos los datos de entrada.

La evaluación parcial puede ser usada para *optimizar* programas, en *compilación* de programas, *intérpretes* y en la generación de *generadores de programas* (generación de compiladores). Puede verse como un caso particular del campo de la *transformación de programas* pero hace un especial hincapié en obtener algoritmos *completamente automáticos* de evaluación parcial.

1

Software Testing

La prueba de programas (*software testing*) es una etapa muy importante dentro del ciclo de vida del desarrollo de *software* si se quiere garantizar una buena calidad de los programas [Mye83], [Bei90]. De hecho es una de las fases que consume más tiempo. En *software testing*, la ejecución de una prueba implica la intención de descrubrir un error en el programa y consiste normalmente en la ejecución de un conjunto de casos de prueba seguida de una evaluación de los resultados obtenidos. La evaluación es un punto clave en este proceso ya que no sólo se evalúa la calidad del programa que se está probando, sino también la calidad del propio proceso de prueba. Sin embargo, la evaluación no es el único punto clave ya que dentro del proceso de prueba, se puede decir que la tarea más complicada es el diseño de *buenos* casos de prueba. Esta tarea puede llegar a ocupar hasta un 40 % del tiempo de la prueba.

Decimos que un caso de prueba es mejor o peor dependiendo de la probabilidad que tiene al ser ejecutado de encontrar un fallo en el código. De hecho, cuanta mayor sea la probabilidad de encontrar un fallo en el código, mejor será el caso de prueba.

Existen muchos tipos de pruebas: unitarias, de integración, de aceptación, de regresión, de carga, *alpha*, *beta*, etc. Tenemos como ejemplo de tipo de prueba más popular el caso de las pruebas *unitarias*. Existen dos enfoques fundamentales para la implementación de pruebas unitarias: el de caja blanca y el de caja negra. Estos dos enfoques se pueden combinar para obtener una técnica más efectiva.

En las pruebas de caja blanca se usa la estructura de control del programa para derivar distintos casos de prueba. El objetivo final, dado el código del programa, es el de cubrir todos los caminos posibles en la ejecución del programa usando el menor número de casos de prueba. Es importante tener en cuenta algunas observaciones. En primer lugar, los errores lógicos y las suposiciones incorrectas son inversamente proporcionales a la probabilidad de que se ejecute un camino del programa. Además, a veces podemos creer que un camino lógico tiene pocas posibilidades de ejecutarse cuando, en realidad, se puede ejecutar regularmente. No olvidemos tampoco los errores tipográficos, que son aleatorios y pueden causar grandes estragos en un programa. En cualquier caso, uno de los mayores problemas del *software* en general es la dificultad de detectar errores que ocurren muy raramente.

Por otro lado, las pruebas de caja negra se centran en los *requisitos funcionales* del *software*. El objetivo que se suele perseguir cuando ejecutamos este tipo de pruebas es el de encontrar errores tales como la existencias de funciones incorrectas o inexistentes, errores relativos a las interfaces, errores en estructuras de datos o en accesos a bases de datos externas, errores debidos al rendimiento, de inicialización o de terminación.

Existen varios métodos que permiten aplicar las pruebas de caja negra:

- el método de las particiones equivalentes divide el parámetro de entrada de un programa en clases de datos. Normalmente este método hace uso de heurísticas para realizar la división en clases de los dominios de los parámetros de entrada.

- con el método del análisis de valores límite se diseñan casos de prueba con los valores de entrada situados en los límites del dominio. Normalmente para cada límite se necesitarán dos casos de prueba (en el límite y fuera de él). Esta técnica es complementaria a la de particiones de equivalencia.

- mencionaremos también el *fuzzy testing*, que es un método en el que se hace uso de herramientas axiliares para generar datos aleatorios que sirven como datos de entrada del programa.

Existe un tipo de prueba llamado *smoke testing* que puede ser catalogado como un subconjunto de las pruebas de caja negra. Consiste en el análisis de cada componente básica del sistema para asegurar que dicha componente funciona correctamente. Normalmente este tipo de pruebas se realizan justo después del ensamblaje del código. El origen de estas pruebas viene de la mecánica, cuando tras el ensamblaje de un aparato se probaba si *salía humo* de él.

Conviene encontrar una buena combinación de todos estos métodos para obtener algoritmos más efectivos y eficientes.

Aunque en este documento no se trata de forma profunda temas como la metodología seguida para calcular los caminos independientes y los casos de prueba en las pruebas de caja blanca, los métodos de partición equivalente y el del análisis de valores límite para las pruebas de caja negra, etc., hemos querido recordad sus bases para facilitar la comprensión del material de este capítulo que en algunos momentos tiene una relación directa con estos conceptos sin duda más populares en la comunidad de la ingeniería del *software*.

1.1. ¿Qué es el *software testing*?

Como ya hemos mencionado antes, el *software testing* es una técnica usada para analizar (aunque no certificar) la corrección, compleción y calidad de los programas. Se puede definir como el proceso de cuestionar un producto para evaluarlo. En cualquier caso, el *software testing* **no** puede demostrar la corrección de un programa. Para esta tarea debemos recurrir a métodos más elaborados, con una base fundamental matemática, como son los métodos formales.

En el mundo de hoy en día, cualquier cosa tiene que ser probada. Desde la maquinaria de una fábrica, al programa que van a ejecutar miles de usuarios, pasando por el boli que compramos en cualquier papelería. Cada producto tiene una serie de características diferentes, y ésto conlleva que el proceso de prueba para cada producto sea completamente distinto. Para ver claramente a qué nos estamos refiriendo con esta diferenciación, nos vamos a centrar en la diferencia entre la prueba de *hardware* y de *software*: en primer lugar, el *hardware* es capaz de producir (relativamente) pocos tipos de errores, mientras que los programas pueden fallar de muchas más formas. Es más, en general resulta imposible detectar *todos los tipos* de fallos de un programa. Para complicar un poco más la prueba, a los errores de programación, en el *software* suelen unirse los errores que se han producido durante el diseño. Esto no es nada común en el *hardware*, donde los fallos suelen provenir de

errores en la fabricación/construcción/montaje pero no en el diseño. En la otra dirección cabe destacar que existe un problema que el *software* no sufre y sin embargo el *hardware* sí lo hace: la *corrosión*. Por ello las pruebsa relacionadas con la corrosión o desgaste son necesarias en el contexto del *hardware* pero no cuando se trata de *software*.

Hemos dicho que el *software testing* nos permite garantizar una cierta calidad en el producto, pero ¿qué significa *calidad* en el *software*?. La respuesta puede resultar hasta sencilla. Un programa tendrá una buena calidad si cumple una serie de requisitos de diseño establecidos con anterioridad. Como es bien sabido, la calidad es un factor muy importante del *software* ya que un único error en un programa puede causar graves daños materiales o incluso pérdidas personales. Para probar un programa, podemos hacer tanto pruebas positivas como negativas pero nunca debemos olvidar el hecho de que una única prueba negativa nos asegura el *mal funcionamiento* de un programa, mientras que varias pruebas positivas *no pueden* probar el *buen funcionamiento* del programa.

Relacionado con este último aspecto podemos estudiar la *fiabilidad* de una prueba teniendo en cuenta aspectos como el número de casos de prueba ejecutados, la forma en la que se han ejecutado, los resultados, su cobertura o los módulos sobre los que se han hecho las pruebas. El objetivo final de toda prueba es obtener un nivel de confianza en el *software* que sea *aceptable*, donde el término aceptable deberá ser también determinado *a priori* y dependerá en gran medida del tipo de programa que se esté probando. Por ejemplo, el grado de confianza necesario para alcanzar un nivel *aceptable* para un simulador de vuelo de una consola será, lógicamente, menor que el grado de exigencia para una controladora real de cualquier avión.

La *cobertura* de una prueba es una noción muy importante. En toda prueba se debe elegir qué criterio de cobertura se va a utilizar para medir la bondad de la prueba. Podemos usar varios criterios de cobertura como veremos en la Sección 1.3, y dependiendo de cada criterio, podremos obtener un conjunto adecuado de casos de prueba distinto.

1.1.1. Calidad de un programa

Vamos a intentar concretar un poco más el término *calidad* relacionado al *software*. El estándard ISO 9126 establece algunos atributos que determinan la calidad de un programa. A continuación enumeramos algunos de ellos con el objetivo de recalcar cuáles son los puntos en los que debe centrarse el proceso de prueba de un programa si quiere que tenga una buena calidad. Obviamente, la prueba de un sistema se deberá centrar en unos aspectos u otros dependiendo del tipo de programa que se esté analizando.

Funcionalidad: La funcionalidad de un programa viene determinada por las funciones que debe implementar. Es decir, depende de los requerimientos especificados con tanterioridad por el cliente final. Aspectos que intervienen en la determinación de la calidad funcional pueden ser la conveniencia de los métodos usados, la precisión en los cálculos (determinada tanto por el *software* como por el *hardware*), la interoperabilidad con otros programas, la seguridad del producto, y el cumplimiento de los requisitos establecidos.

Fiabilidad: Estos atributos están relacionados con la capacidad del programa de mantener su nivel de eficacia bajo determinadas condiciones estresantes y durante un período de tiempo concreto. Tenemos que estudiar por tanto cosas como la madurez del programa, la capacidad de recuperación tras un error, o la tolerancia a fallos.

Eficiencia: Los aspectos relacionados con el compromiso entre la eficiencia del programa y la cantidad de recursos necesitados/usados bajo unas condiciones determinadas son los atributos que miden la eficiencia del *software*. Por ejemplo podemos estudiar o analizar el comportamiento temporal (tiempo de respuesta) y los recursos usados tales como memoria, espacio, procesador, etc.

Portabilidad: La portabilidad se define como la capacidad de un programa de ser transferido de un entorno a otro, por lo que los atributos que miden la calidad del producto en cuanto a portabilidad serán por ejemplo el modo de instalación, la conformidad con estándares, la facilidad de reemplazamiento, la adaptabilidad a otros componentes, etc.

Mantenimiento: A veces hace falta un esfuerzo muy grande para cambiar la mínima cosa en un programa. Los atributos como la estabilidad, la capacidad de ser analizado, la maleabilidad o la sencillez con la que puede ser probado el *software* miden si un programa es fácil o no de mantener.

Amigabilidad: Estos atributos miden el esfuerzo necesario para usar el programa. Podemos analizar aspectos como la facilidad de aprendizaje o la facilidad de comprensión. La técnica más habitual para realizar este tipo de estudios recurre a observar a alguien (por ejemplo el cliente) usando el programa, midiendo y evaluando de esta forma cosas como el tiempo que emplea para realizar una determinada tarea, los errores que ha cometido durante la sesión de trabajo, o incluso la respuesta emocional del usuario tras su uso.

1.1.2. Conceptos básicos en *software testing*

En esta sección vamos a definir algunos conceptos básicos cuando hablamos de prueba de programas. La base de una prueba son los *casos de prueba* ya que una prueba no es más que la ejecución de un conjunto de casos de prueba seguida de un análisis o evaluación de los resultados y del proceso. Un caso de prueba se trata normalmente de una única ejecución del programa (además puede estar restringida a un único paso de ejecución) junto con información importante relacionada con dicha ejecución. Esta información adicional suele consistir en un identificador para el caso de prueba, el orden de ejecución y los requisitos relacionados con el caso de prueba, la profundidad, la clase de la prueba y el autor. Si el caso de prueba es muy grande o complicado, se pueden definir también una serie de prerequisitos. Por último, el caso de prueba debe tener un espacio reservado que contenga el resultado esperado (correcto) que debería obtenerse tras su ejecución. De esta forma será posible comparar el resultado obtenido durante la prueba con el resultado esperado. En pruebas funcionales de corrección, el resultado esperado será la salida del programa, pero cuando hablemos de otros tipos de pruebas, este valor podrían ser otros datos como el tiempo o recursos que debe consumir, etc.

Otro concepto importante es el de *suite* que no es más que una colección de casos de prueba. Normalmente contiene además de los propios casos de prueba, instrucciones u objetivos detallados relacionados con la misma. Debe contener información acerca de la configuración de sistema que se tiene que usar durante la ejecución de los casos de prueba. Además, puede contener prerequisitos y descripciones de pruebas sucesivas. No debemos confundir un *suite* con un *plan de prueba*.

De hecho, un *plan de pruebas* está constituido por un conjunto de pruebas, pero además debe contener una serie de información asociada a dicho conjunto. El concepto es distinto ya que en un plan de pruebas debemos especificar qué tipo de propiedades se están probando y cómo se debe interpretar el resultado de la prueba. También tenemos que describir de forma muy detallada la prueba y cuál es el resultado esperado.

Los *scripts* son pequeños programas que automatizan la ejecución de las pruebas. Normalmente se usan con las pruebas de caja blanca, bien sea para realizar pruebas de unidad, de sistema o de regresión. Para poder usarlos es necesario que sea posible comparar de forma sencilla la salida obtenida en la prueba con la salida esperada. Esto hará que el proceso de comparación pueda ser automatizado.

Si una prueba la definimos en base a una historia *hipotética*, lo que obtenemos es un *escenario*. Un escenario puede ser algo tan sencillo como un diagrama, o incluso puede estar descrito en lenguaje natural y puede ser tan complejo como queramos. Un escenario ideal será aquél que sea una historia motivante, creíble y compleja, pero a la vez fácil de evaluar.

1.1.3. El ciclo de prueba básico

La siguiente lista enumera de forma ordenada los pasos básicos del ciclo de vida de una prueba.

1. Análisis de requisitos
2. Análisis de diseño: ¿qué queremos y bajo qué condiciones?
3. Plan de prueba
4. Desarrollo de las pruebas
5. Ejecución de las pruebas
6. Informe de las pruebas
7. Prueba de los errores

Este ciclo de prueba podría repetirse tantas veces como quisiéranos, así que una buena pregunta que puede surgir es: ¿cuándo hay que parar el *testing*?. El **criterio de parada** de la prueba de un programa depende de distintos factores como pueden ser el presupuesto del proyecto, el tiempo empleado y proyectado y/o la calidad de *software* exigida por el cliente (y el mismo desarrollador). Aun así, podemos definir dos reglas de parada bastantes generales:

- la regla de parada *pesimista* es la más frecuete y consiste en que se para la prueba del sistema cuando se acaba el presupuesto, se acaba el tiempo o se agotan los casos de prueba.
- la regla de parada *optimista* es la que consiste en parar la prueba del programa cuando la fiabilidad del mismo alcanza el nivel exigido por los requisitos, o bien cuando el beneficio de seguir con la prueba no justifica el coste que conlleva.

1.2. Taxonomía

Existen numerosos tipos de pruebas que pueden ser clasificadas siguiendo distintos criterios, dando lugar a distintas clasificaciones independientes. Antes que nada mencionar la diferencia entre las pruebas *estáticas* y las pruebas *dinámicas*. Las pruebas estáticas son aquéllas en las que no es necesario ejecutar un programa para analizarlo. La idea que siguen este tipo de pruebas es la de usar el código del programa para inspeccionarlo y revisarlo. Estas pruebas las suele realizar el mismo programador revisando su propio código. Las pruebas dinámicas sin embargo son aquéllas en las que se ejecuta de una forma más o menos controlada el programa. Nosotros vamos a centrarnos únicamente en las pruebas dinámicas.

Como hemos mencionado, podemos clasificar los métodos de prueba según distintos criterios como puede ser

- la finalidad de la prueba
- el alcance de la misma, o
- la fase del ciclo de vida en el que ocurre.

Tenemos cuatro tipo de pruebas según su **finalidad**:

- pruebas de corrección,
- pruebas de rendimiento,
- pruebas de fiabilidad, y
- pruebas de seguridad.

A partir del criterio de **alcance** de las pruebas y ordenadas de menor a mayor alcance podemos tener

- pruebas de unidad,
- pruebas de componente,
- pruebas de integración, y
- pruebas de sistema.

Si tomamos como criterio el momento en el que se producen las pruebas (es decir, según el **ciclo de vida** del desarrollo del *software*), obtenemos los siguientes tipos de pruebas:

- pruebas de fase de requisitos,
- pruebas de fase de diseño,
- pruebas de fase de programa,
- evaluación de resultados de pruebas,
- pruebas de fase de instalación,

- pruebas de aceptación, y
- pruebas de mantenimiento.

A continuación describiremos de forma un poco más detallada algunos de los tipos de prueba más relevantes.

Pruebas de corrección

Todos estamos de acuerdo en que lo mínimo que cualquier usuario exigirá a un programa es que *sea correcto*. Es decir, aunque podamos aceptar que un programa no sea capaz de dar un resultado determinado, lo que es completamente inadmisible es que el programa de un resultado falso o incorrecto. En las pruebas de corrección se pueden usar tanto pruebas de caja blanca como pruebas de caja negra. El objetivo final, como hemos mencionado ya, es el de garantizar una máxima corrección del producto, pero desgraciadamente, por norma general, no podremos garantizar *total* corrección. Por último diremos que normalmente, en este tipo de pruebas tendremos que recurrir al uso de los requisitos del sistema (a cualquier nivel) para compararlos con el resultado de la ejecución del programa que queremos probar.

Aunque hemos mencionado tanto las pruebas de caja blanca como las de caja negra en el ámbito de la prueba de corrección, tanto unas como otras no sólo sirven para implementar este tipo de pruebas. Las de caja negra por ejemplo pueden usarse también para comprobar la solidez del sistema ejecutando casos que *a priori* no tienen sentido pero que el usuario podría lanzar ya que como programadores no debemos asumir el que usuario va a hacer siempre lo correcto (o lo que esperamos que haga).

Para determinar la corrección de un sistema también resultan útiles las pruebas llamadas *grey tests*. La idea tras estas pruebas es la de permitir manipular el entorno (por ejemplo una base de datos) para ver el estado del producto tras su ejecución. Es decir, analizar los cambios que se han producido al ejecutar el programa. Este tipo de tests son muy útiles para probar aplicaciones cliente-servidor.

Pruebas de rendimiento

Siguiendo con la clasificación según la finalidad de las pruebas, tenemos las llamadas pruebas de rendimiento. Estas pruebas parten de la idea de que no todos los programas tienen los mismos requisitos en cuanto a rendimiento esperado. De hecho algunos sistemas ni los tienen. El objetivo de este tipo de pruebas es el de detectar posibles cuellos de botella en el código, además de comparar y evaluar el rendimiento del programa. Una buena prueba de rendimiento detectará cuándo un determinado componente produce una disminución de rendimiento crítica en el sistema global.

El método más común para comprobar el rendimiento de un programa es el uso de *benchmarks*, es decir, de colecciones de ejemplos que permiten comparar los resultados propios con los obtenidos por otras implementaciones. En la introducción ya mencionamos algunos de los atributos a los que se debe prestar especial antención en este tipo de pruebas, como el consumo de recursos, la producción del sistema, el tiempo de respuesta a los estímulos del usuario o de otros programas, o la longitud media/pico de las colas. Los recursos relacionados con este tipo de pruebas y de los que dependerán los resultados

de las mismas pueden ser, por ejemplo, el ancho de banda de la red, los ciclos de CPI, el espacio en disco, la velocidad de las operaciones de acceso a disco, el uso de memoria, etc.

Pruebas de fiabilidad

El tercer tipo de prueba según la finalidad son las pruebas de fiabilidad. La fiabilidad es la probabilidad de que un sistema no contenga operaciones que produzcan errores. Los casos de prueba que se ejecutan en este marco están orientados a analizar datos de errores. Se usa un modelo de estimación para analizar los datos obtenidos y estimar la fiabilidad actual, a la vez que se puede también predecir la fiabilidad futura en base a los posibles errores existentes. De esta forma los responsables pueden tomar una decisión acerca de la conveniencia de seguir con las pruebas o bien de sacar un producto al mercado, a la vez que los usuarios pueden decidir si les conviene usar el producto o no.

Las **pruebas de solidez** y **pruebas de aguante** son variantes de las pruebas de fiabilidad. La solidez mide el grado en que es capaz de funcionar correctamente un sistema ante la presencia de entradas excepcionales o bajo condiciones de entorno estresantes. Con estas pruebas no se pretende determinar la corrección del programa pero, por ejemplo, sí se detecta si, por ejemplo, el programa termina de forma repentina. En general, este tipo de pruebas es menos costoso que las pruebas de corrección.

Por otro lado, las pruebas de aguante intentan determinar la estabilidad del sistema. El programa se prueba en condiciones cercanas o incluso algo superiores a los límites especificados para el sistema. Se puede por ejemplo forzar la ocupación de recursos y ver cuál es el resultado de ejecutar el programa con una ocupación máxima de recursos. Se puede también simular un acceso masivo a los datos y ver la reacción del sistema, etc.

Pruebas de seguridad

Las pruebas de seguridad son la última categoría dentro de la finalidad de las pruebas. La calidad, la fiabilidad y la seguridad de los programas están íntimamente relacionadas: el objetivo de las pruebas de seguridad es el de identificar y eliminar errores en el código que puedan dar lugar a violaciones de seguridad. Estas pruebas pueden también verse como una forma de validar la efectividad de las medidas de seguridad tomadas en la fase de diseño y desarrollo.

Una técnica bastante común para realizar este tipo de pruebas es la de simular ataques para encontrar los puntos vulnerables del sistema.

Los cuatro tipos de pruebas que hemos descrito hasta este momento pertenecen a la clasificación según la finalidad de la prueba. Cambiamos ahora de criterio de clasificación.

Pruebas de unidad

Las pruebas de unidad se encuadran dentro de la clasificación de tipos de pruebas según el alcance de las mismas. Estas pruebas se realizan para verificar que un módulo unitario específico del sistema funciona correctamente. La idea fundamental que hay tras estas pruebas es la de diseñar casos de prueba para cada función y método de un programa. Normalmente, las primeras pruebas de unidad de un programa las realiza el propio programador. Esta tarea puede ser agilizada y simplificada gracias a que hoy en día

existen herramientas de ayuda tanto en el diseño de casos de prueba como para analizar el comportamiento estático de los módulos, es decir, su tipado, su visibilidad, etc.

Los puntos positivos de las pruebas de unidad son numerosos. Por ejemplo, una prueba de unidad puede verse como la definición de un contrato estricto y claro, el cual debe ser satisfecho por cada parte del sistema. Además, estas pruebas-contrato permiten que el programador tenga la posibilidad de modificar y simplificar el código el día de mañana basándose en las especificaciones (contrato) convirtiéndose así en una tarea más ligera ya que si se desea, se puede obviar el código y basar los cambios en el contrato definido. Es fácil intuir que este tipo de pruebas facilita las *pruebas de regresión* (las que se realizan cuando se actualiza un programa y de las que hablaremos algo más en este mismo capítulo). Este tipo de pruebas también simplifica o agiliza la integración de módulos, resultando especialmente útiles cuando usamos un método de prueba *bottom-up*. *Bottom-up* es el término usado para definir el método que consiste en empezar probando los módulos más básicos y pequeños y a partir de éstos se van probando los que están en un nivel superior (es decir, las que llaman a los del nivel más bajo).

Nótese que las pruebas de unidad casan perfectamente con la programación orientada a objetos. Las pruebas de unidad pueden servir de documentación, ya que observándolas se puede deducir el modo de uso de un determinado módulo.

En cuanto a la ejecución de este tipo de pruebas, se puede realizar tanto manualmente como automáticamente, aunque suele ser más común el método automático: un *script* puede encargarse de automatizar la ejecución de la batería de casos de prueba considerada. Por último, diremos que para garantizar la independencia del módulo con respecto a otros módulos, las pruebas de un determinado módulo unitario suelen ejecutarse en un entorno distinto al sistema donde se integrará a posteriori.

Para realizar las pruebas de unidad se pueden usar tanto pruebas de caja blanca como de caja negra ya que ambas técnicas son complementarias. Con las pruebas de caja blanca se comprueba que el programa hace bien *lo que hace*, pero no comprobamos si hace *lo que queremos que haga*. Por otro lado, las pruebas de caja negra se centran en comprobar que un módulo hace *lo esperado*. Así pues, en realidad deberían llevarse a cabo tanto pruebas de caja blanca como pruebas de caja negra para obtener un mejor resultado.

Pruebas de integración

Siempre dentro de la clasificación por alcance vamos a describir las pruebas de integración, las cuales pueden verse como el paso siguiente a las pruebas de unidad. Las pruebas de integración toman como entrada varios módulos unitarios que previamente han pasado las pruebas de unidad de forma satisfactoria. Estos módulos se unen y se prueban de nuevo, obteniendo así un sistema listo para pasar a la siguiente fase de prueba (*prueba de sistema*).

Puede verse que las pruebas de integración son pruebas a una mayor escala, y de hecho pueden llegar a alcanzar dimensiones industriales dependiendo de la cantidad y tamaño de los módulos que se ensamblen. Existen dos puntos de vista para llevar a cabo estas pruebas. Uno es el punto de vista *estructural* y el segundo el *funcional*. Existe una analogía con las pruebas de unidad, ya que el punto de vista estructural podría verse como una prueba de caja blanca pero a nivel de módulo, no de código, mientas que el punto de vista funcional podría verse como una especie de prueba de caja negra a nivel de sistema. De forma similar a lo que ocurre con las pruebas de caja negra, cuando usamos el punto de vista funcional

en las pruebas de integración se usan las técnicas de partición en clases de equivalencia y el análisis de casos límites.

La idea fundamental tras las pruebas de integración es la de comprobar la coherencia semántica entre módulos abarcando tanto la semántica estática (es decir, si los módulos se llaman de forma correcta entre ellos, etc.), como la semántica dinámica (dicho en otras palabras, si los distintos módulos reciben en realidad lo que esperan). Normalmente, para la ejecución de estas pruebas se sigue una metodología incremental, aunque dentro de esa metodología existen dos alternativas:

1. podemos usar un método descendente en el que se parte de probar módulos generales, asumiendo la existencia y corrección de los módulos más concretos (es decir, los que son llamados desde el módulo que está siendo probado), o bien

2. podemos usar un método ascendente en el que se parte de los módulos más sencillos y concretos y se va subiendo en nivel de generalidad, evitando de esta forma tener que crear módulos ficticios.

Pruebas de sistema

Las pruebas de sistema abarcan el sistema completo. Son por tanto de mayor alcance que las pruebas de integración, y por suspuesto que las de unidad. En esta fase se realiza la prueba en el sistema entero, completamente ensamblado incluso con el *hardware*. Nótese que en las pruebas de integración se hablaba de la parte relacionada con el *software* y en ningún momento se mencionaba el *hardware*. El objetivo es el de analizar si el sistema cumple con todos los requerimientos especificados en el contrato inicial. Las pruebas que se realizan suelen ser de caja negra, es decir, pruebas funcionales.

Para realizar una prueba de sistema, el programa ha debido superar con anterioridad de forma satisfactoria las pruebas de integración. Las pruebas *alpha testing* y *beta testing* (que describiremos más adelante en este mismo capítulo) son consideradas como subcategorías de las pruebas de sistema. Los problemas que se suelen detectar son inconsistencias entre módulos integrados entre sí (ensamblaje) e inconsistencias entre los módulos ensamblados y el *hardware*.

Normalmente, al llegar a esta fase de prueba será la primera vez que tengamos ocasión de probar el sistema de forma completa, fijándonos en la especificación de requisitos únicamente. A parte de las pruebas funcionales, en esta fase se pueden realizar por ejemplo pruebas de solidez y de estrés, comprobando así el comportamiento del sistema tanto dentro de los límites impuestos por los requisitos, como fuera de ellos.

La siguiente lista enumera los distintos tipos de pruebas que se pueden realizar durante una prueba de sistema: pruebas funcionales, de interfaz de usuario, pruebas basadas en el modelo, de salida por error, pruebas de la ayuda de usuario, pruebas de seguridad, de capacidad, de prestaciones, de regresión, de fiabilidad, de recuperación, de instalación, pruebas de mantenimiento o pruebas de documentación. Es decir, cualquier tipo de prueba que esté relacionada con el funcionamiento del producto final.

Pruebas de aceptación

El tipo de prueba con mayor alcance son las pruebas de aceptación, las cuales ocurren una vez que el producto ha pasado las pruebas de sistema. La principal característica de

estas pruebas es que están planteadas por el cliente final y en base al resultado, el cliente dará por bueno el producto y lo comprará o aceptará, o no lo hará. La idea básica que persiguen estas pruebas es la de realizar pruebas funcionales que obtengan una cobertura total de la especificación de requisitos y del manual de usuario. Estas pruebas permiten además comprobar el funcionamiento del producto cuando no toda la interacción del usuario con el programa sigue las pautas esperadas, ya que el usuario final no tiene por qué usar el programa siempre de forma correcta.

Dejamos ahora a un lado ahora las clasificaciones según alcance o finalidad, y describimos brevemente algunos otros tipos de prueba que son interesantes y que ya han sido mencionados en este texto.

Alpha testing

Ya hemos mencionado anteriormente que este tipo de prueba se puede ver como una subcategoría de las pruebas de sistema. Estas pruebas se realizan siempre antes de sacar un producto al mercado. El proceso seguido normalmente es el de invitar al cliente a que pruebe el producto en el mismo entorno de desarrollo. Es decir, un entorno controlado por el desarrollador. De esta forma el usuario tendrá también la facilidad de tener un experto al lado durante las pruebas que pueda resolver cualquier problema surgido.

Tras la prueba, cliente y desarrollador pueden realizar un análisis en común con el que evaluarán el sistema y sacarán sus propias conclusiones. Es bastante usual usar un *tracer* para poder detectar de forma rápida errores que puedan surgir durante la prueba.

Beta testing

Las pruebas *beta* se pueden ver también como un caso especial de pruebas de sistema. La principal diferencia de estas pruebas con respecto a las pruebas *alpha* es que en ellas, el producto no es probado en el entorno de desarrollo, sino en el entorno del cliente. Es decir, las pruebas se ejecutan en un ambiente más real, más adecuado al uso del sistema que se hará tras su instalación y son por tanto en general, pruebas más exigentes que las anteriores.

Normalmente se proporciona una versión *beta* del programa a un grupo de personas capaces de detectar errores en el funcionamiento del sistema. A veces las versiones *beta* se hacen también públicas, de forma que se incrementa el número de *testers* y por tanto la probabilidad de detectar errores.

Pruebas de regresión

El objetivo de las pruebas de regresión es el de asegurar que cuando se produce un cambio en un programa (nuevas versiones, etc.), las modificaciones hechas sobre la antigua versión no afecten a la funcionalidad que tenía (y de hecho debía tener) el programa, es decir, que el programa siga satisfaciendo al menos los mismos requisitos funcionales que ya satisfacía. Estas pruebas son muy importantes porque es frecuente que con un cambio de versión, aparezcan errores que antes no ocurrían, es decir, que lo que antes funcionaba, con la nueva versión ya no lo haga.

Normalmente, el método seguido es el de repetir pruebas realizadas con la versión anterior. De hecho, una buena práctica es la de, cuando un caso de prueba es capaz de detectar

un error automáticamente se convierte en un caso de prueba *interesante*. Entonces dicho error se corrige en el programa pero a la vez, el caso de prueba se almacena en la colección de casos de prueba del programa para volver a ejecutarlo cuando el programa sea modificado en el futuro. Este mecanismo de almacenamiento de casos de prueba interesantes suele poderse automatizar de forma bastante sencilla.

Load testing

Este tipo de pruebas está relacionado con la forma en que se espera que el usuario utilice el programa. Una prueba típica es la de simular el acceso simultáneo al sistema de varios usuarios. De hecho es un tipo de prueba especialmente útil para sistemas multiusuario. Otra prueba básica es la de observar el comportamiento del sistema cuando se le proporciona una entrada de tamaño grande (pero dentro de los límites especificados en los requisitos).

Un tipo de test relacionado con éste son las *pruebas de aguante*, o *stress testing*, las cuales llevan al límite el nivel de exigencia en sus pruebas.

Debates

Como último apartado de esta sección, no queremos dejar de mencionar algunos debates que surgen en cuanto a distintas metodologías que pueden seguirse cuando se realiza *software testing*. Tenemos por ejemplo el debate entre la *prueba ágil* y la *prueba tradicional*. La prueba tradicional es la que tenemos todos en la cabeza, donde tenemos el código del programa, o el programa ejecutable y realizamos las pruebas en base al mismo. Las pruebas ágiles se realizan bajo condiciones de incerteza y cambio continuo.

Podemos también diferenciar entre el método *exploratorio* y el método *guiado* para la realización de las pruebas. Las pruebas exploratorias implican una simultaneidad entre el aprendizaje, el diseño y la ejecución de las pruebas, lo que puede introducir un mayor grado de complejidad en el proceso. Las pruebas guiadas sin embargo implican que el aprendizaje y el diseño de las pruebas ocurren antes que la ejecución de las mismas.

También podemos elegir entre *automatizar* las pruebas o bien hacerlas *manuales*. Automatizar las pruebas puede resultar en algunos casos demasiado costoso debido a la necesidad de usar *oráculos* más o menos potentes y por lo tanto puede no valer la pena el exceso de coste. Los oráculos consisten en algún mecanismo que nos proporcione la información necesaria para poder comparar los resultados de las pruebas con los esperados. Son especialmente costosos los oráculos que tienen que proporcionar información acerca de la corrección de los programas (es decir, los resultados funcionales esperados).

Normalmente las técnicas automáticas no se pueden escalar o aplicar de forma general haciendo necesaria la intervención humana en algún punto del proceso. El grado de intervención depende de la planificación de las pruebas. En cualquier caso, hay algunos tipos de prueba que son más fáciles de automatizar que otros, por ejemplo es más fácil automatizar las pruebas de solidez que las de corrección o incluso las de fiabilidad.

1.3. Grado de cobertura

El grado de cobertura de una prueba es un concepto crucial dentro del *software testing*. Con la cobertura podemos medir lo buena o mala que ha resultado ser una determinada

prueba. Para medir la cobertura podemos seguir varios criterios como veremos más adelante y la elección de uno u otro dependerá en gran medida de la naturaleza y requisitos del sistema mismo que se esté probando y del objetivo que se quiera alcanzar con la prueba.

Una vez elegido el criterio que se seguirá para medir la cobertura, es muy importante que se defina *el nivel de cobertura* que consideraremos como *una buena cobertura*, es decir, cuándo aceptaremos una prueba como satisfactoria. Este *nivel de aceptación* depende principalmente de lo crítico que sea un programa. Se debe valorar por un lado el riesgo (o coste) que implica un fallo durante la ejecución del programa una vez puesto en el mercado. También tenemos que tener en cuenta el número de usuarios que puede tener el sistema y por tanto podrían verse afectados ante un fallo del producto. Normalmente, para un *software* donde un fallo puede afectar únicamente a la imagen dada por el fabricante (es decir, un *software* poco crítico), puede bastar una cobertura entre el 60 % y el 80 %.

A continuación describiremos los distintos criterios de cobertura usados normalmente para la medición:

Cobertura de segmentos o de sentencias: Un segmento es una secuencia de sentencias sin puntos de decisión. Es decir, son los trozos de código que se encuentran situados entre los puntos de decisión de un programa. Una prueba definida en función de este criterio consistirá en el conjunto de casos de prueba capaz de cubrir el máximo número de segmentos del programa.

Cobertura de ramas: El objetivo de las pruebas definidas a partir de este criterio es el de recorrer todas las posibles salidas de los puntos de decisión. Nótese que una cobertura del 100 % de ramas implica de forma automática una cobertura del 100 % de segmentos (excepto si nos encontramos en el raro caso de tener un programa sin puntos de decisión). Este criterio necesita un refinamiento adicional si tratamos con lenguajes de programación que soportan excepciones, ya que habrá que añadir pruebas que provoquen excepciones en el programa a parte de las que cubren las ramas del código. Técnicamente, cada punto de código donde pueda producirse una excepción, se convierte en un punto de decisión.

Cobertura de condición/decisión: La cobertura de condición trocea las expresiones booleanas complejas de las guardas en los puntos de decisión de forma que intenta cubrir todos los posibles valores que puede tomar cada componente de la condición. Este criterio en general mejora el criterio de cobertura de ramas, ya que analiza las posibles evaluaciones de cada componente de un punto de decisión y los valores de las variables de sus guardas.

Cobertura de bucles: Con este criterio, las pruebas definidas tendrán en cuenta la ejecución de los bucles. Los bucles en realidad pueden verse como segmentos controlados por decisiones. Dependiendo del tipo de bucle que estemos considerando, se definirán distintos casos de prueba. Por ejemplo, para un bucle de tipo *while*, ejecutaremos tres casos de prueba: el caso en el que el cuerpo del bucle no se ejecuta, el caso en el que se ejecuta una única vez, y el caso en el que se ejecuta más de una vez. Para bucles de tipo *repeat* sin embargo, se definirán sólo dos casos de prueba: el caso en el que se ejecuta una sóla vez, y el caso en el que se ejecuta más de una vez. Por último, si queremos definir las pruebas para un bucle de tipo *for*, tendremos los mismos casos que en el caso del bucle *while*.

Los bucles *for* pueden resultar peligrosos si no se cumplen una serie de reglas o condiciones. Un bucle *for* será considerado como *seguro* siempre y cuando no se altere la variable de control del bucle dentro del cuerpo del propio bucle. Si se modificara alguna variable de la que dependa la variable de control el bucle *for* dejaría también de ser seguro. Además, para ser seguros tendrán que cumplir también que no tengan ninguna sentencia de tipo *exit*, *return*, o peor aún, *goto*. En los casos en los que un bucle *for* no sea seguro, entonces en vez de usar la cobertura de bucles tendremos que recurrir a la cobertura de ramas ya que los puntos *peligrosos* mencionados arriba representarían en realidad puntos de decisión dentro del bucle.

1.4. Generación automática de casos de prueba

Ya sabemos que dos momentos cruciales de la prueba de programas son la definición de casos de prueba *buenos* y la evaluación del proceso de prueba. De estos dos puntos, el primero es especialmente costoso en términos de coste temporal en general, de ahí la importancia que tienen las técnicas para automatizar la generación de casos de prueba. Estas técnicas pueden reducir dicho coste (y por tanto el coste presupuestario), a la vez que permiten incrementar la calidad de las pruebas (y del producto). Dentro de la generación automática de pruebas podemos encontrar tres paradigmas o metodologías distintas. Por un lado, podemos usar la generación aleatoria, donde se generan entradas aleatoriamente hasta que se encuentra una entrada significativa (o útil) según el criterio de cobertura elegido. El problema de la generación aleatoria de casos de prueba radica en el hecho de que cuanto más complejo sea el programa, más difícil es encontrar un caso de prueba útil.

La segunda metodología es la generación de tests simbólicos con la que básicamente se asignan valores simbólicos a las variables. Por último, también podemos usar la generación dinámica de pruebas, con la que se hace una búsqueda del caso de prueba a través de la ejecución del programa que queremos probar, es decir. la misma ejecución del programa va guiando la generación de los casos de prueba. Para este último método se usan técnicas de búsqueda metaheurísticas como son los algoritmos genéticos [Sch01, Sch04, Mit96, Mic99], el recocido simulado (*simulated annealing*) [KGV83] o la búsqueda tabú [GL97].

A continuación vamos a presentar de forma más detallada algunas de las técnicas para la generación automática de casos de prueba.

1.4.1. Método aleatorio para la generación de pruebas

En la generación aleatoria de casos de prueba, al contrario de lo que ocurre en otras metodologías, *a priori* no se necesita información adicional sobre el código para generar los casos de prueba (grafos de flujo, requisitos, etc.), sino que éstos se definen de forma aleatoria. En general, este método tiene un coste computacional menor que el resto de aproximaciones. A pesar de esta ventaja, no se puede decir que los métodos aleatorios sean mejores que los métodos basados en conocimiento previo que suelen tener un coste mucho mayor, igual que tampoco se puede afirmar que lo contrario sea cierto de forma absoluta debido a que muchas veces, el tiempo empleado en generar un único caso de prueba por un método basado en conocimiento previo, puede ser el que necesite el método aleatorio para generar de forma completamente aleatoria un conjunto de casos de prueba que contenga un único caso útil.

Como hemos adelantado, el objetivo de esta técnica es el de generar un conjunto de casos de prueba de forma aleatoria, sin embargo los casos de prueba deben ser útiles, en el sentido de que tienen que tener sentido para el programa. Sólo las pruebas con sentido serán significativas, por lo que aun siendo una técnica muy eficiente, puede resultar poco efectiva. Sin embargo, existen métodos dependientes de los lenguajes de programación usados, que permiten guiar la generación de los casos haciéndolo algo menos aleatorio, descartando por ejemplo de forma automática los casos triviales y obteniendo de esta forma un mayor número de casos significativos tras un número determinado de intentos (o generaciones) [Nta98].

A lo largo de los años se han hecho numerosas comparativas de muchos métodos con respecto al método aleatorio ya que es muy fácil de implementar y, aunque muchos autores defienden la poca eficacia de esta técnica, en la práctica muchas veces es el mejor método. Veremos más adelante que esto es debido en gran parte a que los demás métodos necesitan bastante información extra acerca del código y bastante tiempo de pre-proceso en base a dicha información extra, por lo que como hemos explicado antes, el tiempo que necesitan para generar un caso de prueba puede ser usado por el método aleatorio para generar un conjunto de casos de prueba suficientemente grande como para que contenga un caso útil.

1.4.2. Método simbólico de generación de casos de prueba

Los métodos simbólicos intentan generar casos de prueba a partir de cierta información *estática* relacionada con las pruebas del programa que se quiere probar. En esta sección presentaremos una técnica en particular, la presentada en [DO91], que está basada en la generación a partir de *mutaciones del código*. En particular, esta técnica genera restricciones algebraicas para describir casos de prueba cuyo objetivo es el de encontrar un tipo predeterminado de fallos.

Una de las mayores desventajas de las metodologías simbólicas con respecto a la generación dinámica de pruebas (que veremos con más detalle en la Sección 1.4.3) se produce al permitir el uso de ciertas primitivas en el lenguaje de programación considerado, por ejemplo la definición de *arrays* cuyas dimensiones dependan de los datos de entrada. Es decir, *arrays* cuyo tamaño no está definido en tiempo de compilación, introduciendo así complejidad añadida al método. Aunque este problema ha encontrado alguna solución parcial ya, los métodos usados son muy ineficientes ya que, por ejemplo, pueden consistir en la generación de casos de prueba para cada dimensión del *array* posible.

En los métodos dinámicos este problema no existe ya que se trabaja con datos (y ejecuciones) reales y por tanto no se deben considerar todas las posibles dimensiones sino sólo las determinadas por los datos de entrada actuales.

Notación y conceptos básicos

En primer lugar usaremos la siguiente notación: si P es un programa, entonces $P(t)$ denota el valor de la función computada por P en el punto t. El conjunto de todos los puntos en los que P está definida es el *dominio* D de P. El criterio de prueba usado es el de *adecuación* definido como sigue:

Definición 1.4.1 (Adecuación) *Si P es un programa que implementa la función F con dominio D, entonces un conjunto de pruebas $T \subset D$* es adecuado *para P y F si para todos los programas Q, si $Q(D) \neq F(D) \Rightarrow \exists t \in T$ tal que cumple que $Q(t) \neq F(t)$.*

Es decir, que un conjunto de casos de prueba es adecuado si consigue detectar todas las versiones incorrectas del programa (si consigue que falle la ejecución en el programa incorrecto). Nótese que con este criterio no demostramos la corrección de P, sino que generamos un conjunto de pruebas T que será capaz de detectar cualquier error en el programa. El problema que tiene este criterio es que no existe un procedimiento efectivo para generar un conjunto de casos de prueba *adecuado*. Por ello aparece la noción algo menos ambiciosa de *casi adecuación* (o *mutation adequacy*):

Definición 1.4.2 (*Mutation adequacy*) *Si P es un programa que implementa la función F de dominio D y Φ es una colección finita de programas, entonces un conjunto de pruebas $T \subset D$ es* adecuado para P con respecto a Φ *si para todo programa $Q \in \Phi$, si $Q(D) \neq F(D) \Rightarrow \exists t \in T$ tal que cumple que $Q(t) \neq F(t)$.*

Por lo tanto, con esta condición el conjunto de casos de prueba será capaz de detectar sólo los errores contemplados en los programas (incorrectos, mutados) de Φ.

Mothra es un sistema que analiza un conjunto de casos de prueba e indica si dicho conjunto es adecuado o no para un programa determinado. A este tipo de sistemas se les llama *acceptors*. Mothra usa 22 *operadores de mutación* (es decir, operadores que dado un programa generan otro programa casi idéntico al dado) para definir pruebas de programas escritos en Fortran y más de 70 operadores de mutación para programas escritos en C. Los operadores están definidos para generar los errores más frecuentemente cometidos por los programadores de esos lenguajes. Además, en el diseño de los operadores de mutación se tienen en cuenta criterios como el de cobertura de ramas, de valores límite, de perturbaciones del dominio, etc. de forma que se generan casos de prueba orientados a tener buena cobertura en esos aspectos.

Las variantes de los programas que se generan a partir de los operadores de mutación y que en nuestro ámbito serán prácticamente siempre programas *incorrectos*, son los llamados *mutantes* y serán los programas pertenecientes al conjunto Φ. Llamamos programa incorrecto a aquél que da un resultado distinto al esperado para algún dato de entrada. Una vez construido el conjungo Φ de programas incorrectos (mutantes) que representan los errores que queremos ser capaces de detectar, debemos definir un conjunto de casos de prueba que sea adecuado (en realidad casi-adecuado siguiento la Definición 1.4.2). Existe una interacción entre el sistema de generación de casos de prueba y un *tester* para determinar los datos que se ejecutan de forma correcta en el programa original pero que provocan que los mutantes fallen. Si el *tester* es una persona, la tarea de elegir los casos de prueba que hacen fallar a los mutantes puede resultar realmente dura, por ello la automatización de este proceso es importante. A continuación explicaremos el método con más detalle.

La estrategia

La estrategia seguida por este método es la de representar como restricciones algebraicas las condiciones bajo las que un mutante fallaría, es decir, condiciones de los daots de entrada que hacen que el resultado del programa original y del mutante sean diferentes. A partir de estas restricciones, automáticamente se generarán los datos que las satisfacen y que constituirán los casos de prueba. Dicho en otras palabras, dado un conjunto de mutantes (variantes del programa), se determina las condiciones o casos en los que estos programas no darían un resultado correcto tras su ejecución. Estos casos o condiciones se

expresan mediante restricciones y, a continuación, se genera un conjunto de casos de pruebas que satisface dichas restricciones. Esta metodología se llama *constraint-based testing* (CBT) y es una técnica *de aproximación*, así que los resultados obtenidos en ningún momento deben verse como absolutos. Lo que sí es posible es medir el grado de aproximación del resultado generado de forma que podemos saber lo cerca o lejos que está de ser un conjunto casi adecuado (Definición 1.4.2).

Vamos a describir de forma intuitiva el procedimiento iterativo que se sigue para la estimación de calidad de los casos de prueba generados a partir de los mutantes. En todo momento tendremos un conjunto de mutantes y un conjunto de casos de prueba generados (inicialmente vacío). Antes de añadir un caso de prueba al sistema de mutación, dicho caso de prueba se ejecuta en la versión original del programa para generar la *respuesta esperada*. Un oráculo (normalmente un usuario) determinará si dicha respuesta es correcta. Si no lo es, entonces el programa debe ser corregido y el proceso de análisis debe iniciarse de nuevo. Si por el contrario el resultado es correcto, entonces se ejecuta el caso de prueba con cada mutante comparándose así los resultados obtenidos por los mutantes con la respuesta esperada. Si la respuesta dada por el mutante no coincide con la esperada se puede decir que hemos generado un caso de prueba capaz de distinguir dicho mutante del original (y por lo tanto el error introducido en esa versión del programa). Esto quiere decir que podemos eliminar del conjunto de mutantes los mutantes cuya respuesta no coincida con el esperado (ya que ya tenemos un caso de prueba útil capaz de identificarlos).

Una vez terminado el proceso iterativo, se obtienen dos resultados. Uno es la cantidad de mutantes que han sido elminados, lo que nos indica lo buena que será la prueba del programa con ese conjunto de casos de prueba. Por otro lado, los mutantes que no hayan sido eliminados nos indican fallos en el conjunto de casos de prueba (y potencialmente fallos del programa que no serán detectados). Siempre es posible añadir a posteriori nuevos casos de prueba que consigan eliminar los mutantes restantes. El objetivo final del método es crear casos de prueba que *eliminen* todos mutantes del conjunto Φ pero como veremos, no siempre será posible hacerlo.

Ejemplo 1.4.3 (Función MAX) Supongamos que tenemos la función MAX mostrada en la Figura 1.1 que devuelve el entero más grande de los dos proporcionados como entrada a la función.

```
    FUNCTION MAX (M,N)
1   MAX = M
2   if (N > M) MAX = N
3   return
```

Figura 1.1: Pseudocódigo para la función *mayor*

A continuación mostramos cuatro mutantes generados mediante la modificación de una de las sentencias del programa original. El símbolo Δ representa la línea que ha sido mutada.

	FUNCTION MAX (M,N)		FUNCTION MAX (M,N)
1 Δ	MAX = N	1 Δ	MAX = abs(M)
2	if (N > M) MAX = N	2	if (N > M) MAX = N
3	return	3	return
	FUNCTION MAX (M,N)		FUNCTION MAX (M,N)
1	MAX = M	1	MAX = M
2 Δ	if (N < M) MAX = N	2 Δ	if (N ≥ M) MAX = N
3	return	3	return

Nótese, que el último mutante definido no introduce ningún cambio funcional del programa original, por lo que será imposible generar un caso de prueba para el que el mutante devuelva como resultado un valor distinto al devuelto por el programa original. ■

Generación de restricciones a partir de mutantes

Antes de evaluar la bondad de los casos generados, hay que obtenerlos a partir de los mutantes. Para ello la primera característica importante que tenemos que tener en cuenta es que, ya que un mutante consiste en un único cambio sobre el código original, la diferencia en el resultado en un mutanto ocurrirá siempre tras la ejecución del punto del código donde se ha introducido el cambio. Esta idea nos lleva a definir la noción de *condiciones de necesidad*.

Teorema 1.4.4 *Dado un programa P y un programa mutado M que consiste en cambiar la sentencia S de P, para que un caso de prueba t elimine el mutante M, es* necesario *que el estado de M inmediatamente posterior a la ejecución de S sea distinto del estado de P en ese mismo punto.*

En cualquier caso, aunque se cumpla esta condicion, puede que el mutante no sea eliminado (por eso le damos el nombre de condición necesaria y no de suficiente). Encontrar condiciones suficientes es una tarea inabordable en la práctica ya que implica el conocimiento de antemano el camino que tomará un programa. Existen trabajos que intentan solucionar este problema y generar condiciones suficientes pero no vamos a tratar este tema en este documento.

Analicemos más detalladamente el caso de tener un caso de prueba que sea necesario pero no suficiente. En este caso, la salida del mutante M bajo ese caso de prueba puede ser idéntica a la salida del programa original P. Esto significaría que el estado tras la ejecución de S en el mutante es distinto al estado en el mismo punto de P (recordemos que sí que es una condición necesaria) pero después, durante su ejecución, dicho estado convergerá a un estado que coincide con el de P.

Restricciones de necesidad

A continuación mostramos cómo pueden definirse las restricciones de necesidad a partir de los mutantes usando el ejemplo introducido. En Mothra existe una plantilla con la que es posible definir de forma automática las restricciones asociadas a cada operador de mutación. Recordemos que los operadores de mutación son aquéllos que introducen

las variaciones en el programa. En otras palabras, cada vez que se aplique un operador tendremos su correspondiente restricción de forma directa.

Ejemplo 1.4.5 Las restricciones identifican (y permiten detectar y eliminar por tanto) el mutante correspondiente. En la Figura 1.2 hemos puesto todas las variantes introducidas en los mutantes del Ejemplo 1.4.3 en un mismo código para poder ver claramente en qué consiste la definición de las restricciones. Nótese que se trata de encontrar el caso en el que el programa original y el mutado darían un resultado distinto.

	FUNCTION MAX (M,N)	
1	MAX = M	
Δ	MAX = N	M≠ N
Δ	MAX = abs(M)	M < 0
2	if (N > M) MAX = N	
Δ	if (N < M) MAX = N	(N>M)≠(N<M)
Δ	if (N ≥ M) MAX = N	(N>M)≠(N≥M)
3	return	

Figura 1.2: Restricciones asociadas a los mutantes

En la parte derecha de la tabla anterior se muestra la restricción correspondiente a la variación introducida en el mutante. Si nos fijamos en el primer mutante podemos ver que el único caso en el que el mutante daría el mismo resultado que el código original sería cuando N=M. Por tanto, M≠N será la restrocción que los datos de entrada deberán satisfacer para que el comportamiento de las dos versiones (mutante y original) sean distintos.

El último caso merece especial anteción ya que la restricción se satisfará sólo cuando N=M. Supongamos que se da este caso, entonces el programa original daría como resultado M mientras que el mutante daría N, pero al ser ambos datos iguales, el sistema no podría observar diferencia alguna (el resultado del mutante coincide con el esperado). Este es el motivo por el que hay veces que es imposible eliminar todos los mutantes (hemos obtenido un mutante funcionalmente equivalente al original). ■

La siguiente tabla presenta la plantilla de Mothra para la definición de las restricciones generadas para cada operador de mutación. La primera columna describe lo que hace el operador de mutación mientras que la segunda define las restricciones que se generan de forma automática cuando se usa el operador correspondiente.

sustitución de un *array* por otro	$A(e_1) \neq B(e_2)$
intercambio de *array*s comparables	$A(e_1) \neq B(e_2)$
introducción de un valor absoluto	$e_1 < 0$
	$e_1 > 0$
	$e_1 = 0$
sustitución de un *array* por una constante	$C \neq A(e_1)$
sustitución de un operador aritmético	$e_1 \rho e_2 \neq e_1 \phi e_2$
	$e_1 \rho e_2 \neq e_1$
	$e_1 \rho e_2 \neq e_2$
	$e_1 \rho e_2 \neq Mod(e_1, e_2)$
sustitución de un *array* por una variable	$X \neq A(e_1)$
sustitución de una constante por un *array*	$A(e_1) \neq C$
sustitución de una variable por un escalar	$X \neq C$
sustitución de una sentencia DO end	$e_2 - e_1 \geq 2$
	$e_2 \leq e_1$
sustitución de conector lógico por otro	$e_1 \rho e_2 \neq e_1 \phi e_2$
sustitución de operador relacional por otro	$e_1 \rho e_2 \neq e_1 \phi e_2$
sustitución de variables escalares	$X \neq Y$

Por ejemplo, se supone que en el primer caso hemos sustituido el nombre del *array* A por el nombre de otro *array* B y la expresión original e_1 por e_2.

Restricciones de predicados

Aunque no podamos calcular las condiciones suficientes de forma eficiente, podemos intentar aproximar el resultado lo máximo posible. Es decir, tener restricciones lo más precisas posible para detectar el mayor número de mutantes del conjunto generad. Para ello, además de las condiciones de necesidad, podemos definir unas condiciones de predicado. Estas condiciones tratarán de que predicados distintos puedan ser diferenciados. La técnica define para cada mutante e' de una expresión e una restricción de predicado de la forma $e \neq e'$.

Ejemplo 1.4.6 Supongamos que tenemos la expresión

$$e = \text{ if (I+K} \geq \text{J) then ...}$$

y definimos el siguiente mutante, que simplemente sustituye la variable I por una constante 3

$$e' = \text{if (3+K} \geq \text{J) then ...}$$

La restricción de necesidad definida a partir de las plantillas sería I≠3.

Supongamos ahora que tenemos el siguiente caso de prueba I=7, J=9 y K=7. Esto significaría que ambos resultados: I+K = 14 y 3+K = 10 son mayores o iguales que J=9. La restricción impuesta por el operador de mutación se satisface, por lo que no podríamos distinguir a priori el mutante.

Probemos ahora a añadir la restricción de predicado según lo indicado. Tendríamos que añadir $e \neq e'$, es decir, (I+K ≥ J)≠(3+K ≥ J). Es fácil observar que esta restricción ahora no se satisface ya que (I+K≥ J) = (3+K≥J) = *true*. De esta forma seguiremos

buscando un caso de prueba adecuado para la distinción de este mutante ya que habremos detectado que el actual no es válido (no satisface las restricciones impuestas). ■

1.4.3. Generación dinámica de casos de prueba

La idea principal en la generación dinámica de casos de prueba es la de utilizar información recogida durante una serie de ejecuciones del programa para generar un conjunto de casos de prueba que se acerque cada vez más a la condición de aceptación. Como puede intuirse, es un procedimiento iterativo al igual que el caso simbólico, pero a diferencia de aquél, para generar los casos de prueba se usarán las trazas de ejecución del programa como guía.

A continuación vamos a presentar algunos métodos dinámicos. Como se mencionó antes, existen tres aproximaciones principales para la definición de algoritmos dinámicos: uso de algoritmos genéticos, de recocido simulado y los basados en la búsqueda tabú.

TESTGEN. Método de Korel

Este método, definido en [Kor90] para la generación dinámica de casos de prueba está basado en tres conceptos fundamentales:

- la ejecución real del programa que se quiere probar,
- métodos de minimización de funciones, y
- análisis dinámico del flujo de datos del programa.

La idea intuitiva es la de monitorizar la ejecución del programa e ir variando los valores de los datos de entrada en función de esa monitorización. Cuando durante la ejecución de un programa se detecta un error (se considera un error cuando la ejecución no sigue el camino deseado o establecido en un principio), entonces un algoritmo de minimización de funciones identificará los valores de las variables de entrada que lo provocan, y por tanto serán éstos los que deberemos variar. El análisis dinámico del flujo de datos podrá decirnos cuál es la variable causante del mal funcionamiento del programa. Aunque en un principio esta metodología no consideraba que un programa pudiera tener estructuras de datos dinámicas, existen trabajos que muestran cómo extenderla para considerar esta capacidad.

En resumen, dados los caminos de ejecución de un programa, el generador de casos de prueba determina los valores de las variables de entrada que fuerzan que la ejecución siga precisamene esos caminos. Como hemos dicho al principio, necesitaremos el grafo de flujo del programa para determinar cuáles son las variables que provocan un error.

Definición 1.4.7 (Grafo de flujo) *El grafo de flujo de un programa Q será un grafo dirigido $C = (N, A, s, e)$ donde N es el conjunto de nodos, A es una relación binaria entre nodos y e y s son respectivamente los nodos de entrada y salida pertenecientes a N.*

Cada nodo representa la parte ejecutable más pequeña que tiene una única entrada y una única salida. Puede ser, por ejemplo, la asignación de valor a una variable, la parte de la condición de un *if-then-else*, etc.

Un arco $(n_1, n_2) \in A$ representa un cambio de control de la instrucción n_1 a la instrucción n_2. Puede tratarse de un simple cambio de control entre sentencias secuenciales.

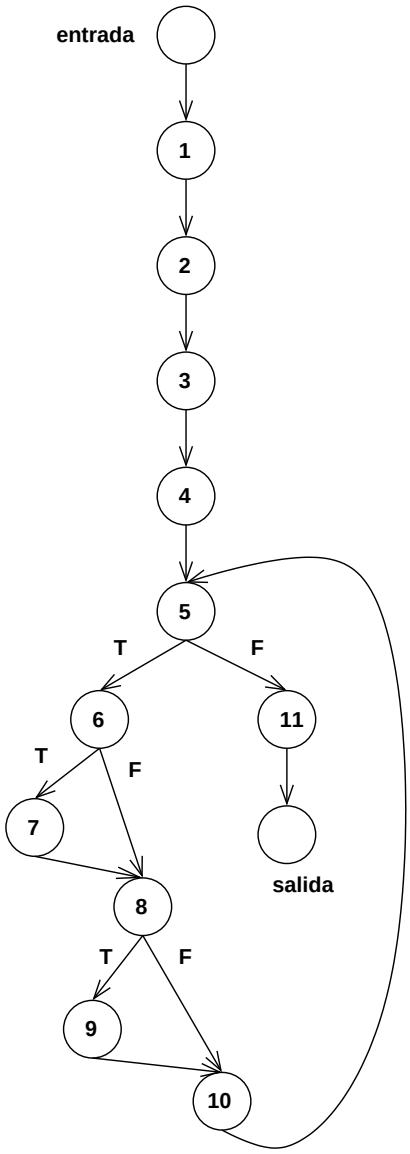
Si el cambio está relacionado con alguna condición, al arco se le llama *rama*. Un *camino* de un programa es una secuencia de nodos (instrucciones) entre las cuales existen arcos definidos.

A cada rama del programa (salida de punto de decisión), se le podrá asignar una función objetivo que la identificará. Así, para forzar la ejecución por ese punto en concreto, habrá que determinar los valores de entrada que minimicen dicha función objetivo. Para el cálculo se podrán usar los métodos tradicionales de minización de objetivos. A continuación veremos un ejemplo que aclarará el procedimiento.

Ejemplo 1.4.8 (Máximo y mínimo de un *array*) El siguiente programa calcula el valor máximo y mínimo de una matriz dada. Los números que aparecen a la izquierda de cada sentencia representan los nodos correspondientes en el grafo de flujo de control mostrado a la derecha.

```
var
A: array[1..100] of integer;
low, high, step: integer;
min, max: integer;
i: integer;

    begin
1   input(low,high,step,A);
2   min := A[low];
3   max := A[low];
4   i:= low+step;
5   while i<high do
        begin
6       if max < A[i] then
7         max:=A[i];
8       if min > A[i] then
9         min:=A[i];
10      i:=i+step;
        end
11  output(min,max);
    end;
```



A partir del grafo de flujo se determinan los caminos para los que se quieren generar casos de prueba. Idealmente deberían cubrirse todos los posibles caminos, pero a veces nos

conformamos con cubrir las ramas o las sentencias (dependerá del criterio de cobertura elegido).

Consideremos un camino particular del grafo, por ejemplo

$$P = \langle e, 1, 2, 3, 4, 5, 6, 8, 10, 5, 6, 8, 9, 10, 5, 11, s \rangle$$

Para definir un caso de prueba *bueno* para P debemos dar valores a los parámetros de entrada low, high, *step* y los elementos de *A* para los que la traza de ejecución sea exactamente el camino P.

Inicialmente todas las variables de entrada reciben un valor inicial cualquiera. Supongamos que damos los siguientes valores: low = 39, high = 93, step = 12, A[1] = 1, A[2] = 2, ..., A[100] = 100. Se ejecuta entonces el programa con los datos de entrada monitorizando en todo momento los nodos del grafo por los que va pasando la ejecución y comprobando que coinciden con los de P. En este caso en concreto, el resultado es que se ejecuta con éxito el subcamino $P_1 = \langle e, 1, 2, 3, 4, 5, 6 \rangle$ de de P. El siguiente paso de ejecución en el programa según P debería corresponder a la rama que va de 6 a 8 (es decir, la rama correspondiente a que la condición sea falsa), pero esto no tiene lugar porque la condición se cumple para los datos de entrada que se han dado.

La condición asociada a la rama de 6 a 8 es max < A[i] pero lo que queremos es que **no** se cumpla ya que queremos que tome la opción *else*. Nótese que tenemos max = A[low] = A[39] = 39. Por otro lado, A[i] = A[low+step] = A[39+12] = A[51] = 51, por lo que la condición max < A[i], 39 < 51 es **cierta**. ■

Cada vez que una ejecución queda bloqueada como en el ejemplo, es decir, cada vez que la traza de ejecución no coincide con el camino considerado P, el algoritmo debe arreglárselas para lograr superar el punto de ejecución donde falla. Para ello, se tendrán que modificar los valores de los datos de entrada ya que sólo de ellos depende la traza de ejecución del programa. Como hemos mencionado antes, cada punto de decisión en el programa tiene asociada una función objetivo que lo identifica. El algoritmo de generación de datos define siguiendo una plantilla la función asociada a la rama que ha fallado (en el sentido que es la rama donde se para el camino inicial) e intenta minimizarla explorando (y variando) por orden las variables de entrada.

Supongamos que $F_1(x)$ es la función asociada a la rama correspondiente (rama (6,8) en el ejemplo), el algoritmo buscará una variable x de entrada del programa que satisfaga la restricción $F_1(x) \leq 0$. Para cada tipo de condición asociado a las ramas, el algoritmo asocia un predicado siguiendo la plantilla siguiente:

Condición	Función	Relación
$E_1 > E_2$	$E_2 - E_1$	$<$
$E_1 \geq E_2$	$E_2 - E_1$	$\leq$
$E_1 < E_2$	$E_1 - E_2$	$<$
$E_1 \leq E_2$	$E_1 - E_2$	$\leq$
$E_1 = E_2$	$abs(E_1 - E_2)$	$=$
$E_1 \neq E_2$	$-abs(E_1 - E_2)$	$<$

Donde E_1 y E_2 son expresiones aritméticas y la última columna define la relación con 0, es decir, la función final será de la forma Función Relación 0. Siguiendo con el ejemplo:

Ejemplo 1.4.9 (Continuación) El predicado asociado (a partir del código) a la rama es max≥A[i]. Es decir, la negación de la condición que rige la sentencia ya que queremos que no se ejecute la parte then de la misma. Siguiendo la plantilla, la función asociada sería $F_1(x) =$ A[i]-max y la relación es (siempre según la plantilla) $\leq$. Una vez definida la función, intentamos modificar los datos de entrada. Si modificamos (ya sea incrementando o decrementando) el valor de los elementos del *array* empezando desde el primero, no observamos cambio alguno en el valor de la función. Al llegar al elemento A[39], el valor de la función cambia, lo que nos da una pista sobre el camino que debemos seguir. Si incrementamos el valor de este dato de entrada en una unidad, la función objetivo decrementa su valor. Si, por ejemplo, asignamos A[39] = 139, entonces la rama (6,8) será tomada y habremos satisfecho nuestro primer objetivo. ■

Una vez determinada la dirección buena (es decir, incrementar o decrementar una variable) hacia la que el valor de la función se acerca al objetivo hay que decidir el valor concreto que le asignamos a una variable. Esta decisión puede realizarse de varias formas y dependiendo de lo grandes que sean los incrementos o decrementos, tardaremos más o menos en alcanzar el objetivo. Por lo tanto, la eficiencia del algoritmo depende en gran medida de la elección del método de asignación de valor a la variable modificada.

Ejemplo 1.4.10 (Continuación) Así pues, con los nuevos datos de entrada, el subcamino recorrido es $P_2 = \langle e, 1, 2, 3, 4, 5, 6, 8\rangle$ pero vuelve a fallar en la rama que va del nodo 8 al 10 por lo que volvemos a tener que definir una función $F_2(x)$ asociada a la rama *min* $\leq$ *A[i]* para minimizarla. En este caso, $F_2(x) =$ *min-A[i]* y tendremos que determinar los valores de entrada para que $F_2(x) \leq 0$. La búsqueda de los valores apropiados se realiza de la misma forma, y de nuevo hasta el elemento número 39 del array no se produce cambio alguno en la función al variar los valores de entrada. Ahora detectamos que si decrementamos el valor de *A[39]*, el valor de la función $F_2(x)$ se decrementa. El algoritmo establece pues que si asignamos *A[39]=51*, se cumple el objetivo buscado (nótese que se debe cumplir también F_1).

Con los nuevos datos de entrada, ahora el subcamino recorrido se amplía y es $P_3 = \langle e, 1, 2, 3, 4, 5, 6, 8, 10, 5, 6\rangle$. El error ocurre en la misma rama que la primera función objetivo. En este caso se procede de forma similar definiendo la función $F_3(x)$ y realizando una búsqueda similar pero esta vez no encontramos una mejoría del resultado de la función hasta que no modificamos (decrementando su valor) el elemento *A[63]* del *array*. Asigmanos un nuevo valor *A[63]=-37*.

Ahora el camino recorrido es $P_4 = \langle e, 1, 2, 3, 4, 5, 6, 8, 10, 5, 6, 8, 9, 10, 5\rangle$ pero no podemos ejecutar la rama que va del punto 5 al 11. Ahora la función $F_4(x) =$ *high-i* y debemos encontrar un valor para los datos de entrada x que haga $F_4(x) \leq 0$. En este caso el cambio de valor de los elementos del *array* no implica cambio alguno en el valor de la función, pero el decremento de la variable *high* sí lo hace. El algoritmo asigna el valor *high = 67* lo que produce que con los nuevos datos de entrada, el camino entero P sea ejecutado. ■

Existen mejoras en cuanto a eficiencia temporal del algoritmo descrito en el ejemplo. Por ejemplo, en vez de tener un orden rígido en las variables de entrada de forma que siempre se intenta satisfacer la función objetivo modificando las entradas en el mismo orden, podría analizarse cuáles son las entradas con más posibilidades de éxito en este aspecto (en el ejemplo, en la función F_4 podríamos haber intentado directamente modificar la variable high en vez de probar con todos los elementos del *array*). Precisamente en

esta parte del algoritmo es donde un grafo de flujo de datos es muy útil ya que puede decirnos qué variables afectan a una determinada función. También existen extensiones del algoritmo que permiten tratar estructuras de datos dinámicas o punteros.

Metaheurísticas

Dentro del marco de la generación dinámica de casos de prueba a parte del método de Korel, encontramos una clase de métodos que hacen uso de metaheurísticas para definir casos de prueba de caja blanca. Existen varias aproximaciones, unas basadas en algoritmos genéticos, otras en el recocido simulado o en la búsqueda tabú. En esta sección veremos algunos de estos métodos heurísticos.

Entre las aproximaciones basadas en algoritmos genéticos tenemos los métodos de

- Jones, Stharner y Eyres de 1996
- McGraw, Michael y Schatz de 1998
- Pargas, Harrold y Peck de 1999
- Lin y Yeh de 2001

En general, los algoritmos genéticos son especialmente apropiados para ser usados en pruebas con una naturaleza iterativa. Por ejemplo casan muy bien con las pruebas de regresión ya que puedes partir de una población inicial basada en los casos de prueba de una versión anterior del programa e ir iterando a partir de ella.

En relación a las otras aproximaciones, Díaz y Tuya definieron un método basado en la búsqueda tabú mientras que Tracey, Clark y Mander en 1998 definieron un algoritmo basado en el recocido simulado (*simulated annealing*).

Como primera aplicación de los algoritmos genéticos a la generación automática de casos de prueba nos encontramos el trabajo *Automatic structural testing using genetic algorithms* de B. Jones, H. Stharner y D. Eyes publicado en *The Software Engineering Journal*, 11 en 1996. El objetivo en este trabajo es el de maximizar la cobertura de ramas usando el grafo de control de flujo del programa [JSE96].

El siguiente trabajo aparecido, *Generating software test data by evolution*, de G. McGraw, C. Michael y M. Schatz lo encontramos como Informe técnico de la RST Corporation en 1998 pero también en [MMS01]. El objetivo de esta técnica es la de maximizar la cobertura de ramas y la cobertura de condición. La generación automática de pruebas se realiza usando un sistema propio llamado GADGET (*Genetic Algorithm Data Generation Tool*), diseñado para trabajar con programas de gran tamaño escritos en C y C++. La metodología presentada no restringe el conjunto de construcciones de C que pueden usarse en los programas, es decir, considera todos los constructores de C, sin restricciones, y se basa en la idea de encontrar un camino hacia el punto en el código donde se quiere que el criterio de cobertura sea satisfecho, y convertir ese criterio en una función que pueda ser minimizada.

El trabajo de R.P. Pargas, M.J. Harrold y R.R. Peck titulado *Test data generation using genetic algoritms*, publicado en *The journal of software testing, verification and reliability*, describe el sistema TGEN que trata de maximizar la cobertura de objetivos usando el grafo de control de dependencias del programa, en el que los nodos representan sentencias y los arcos representan las dependencias entre sentencias (o predicados) [PHP99]. La función

objetivo compara el conjunto de predicados actual con el conjunto de predicados que tienen que cumplirse para llegar al nodo objetivo. Una solución que cubre la mayoría de los predicados tendrá una evaluación alta de la función objetivo.

Por último en cuanto a algoritmos genéticos, tenemos que hablar de la aproximación de J. Lin y P. Yeh *Automatic test data generation for path testing using GAs* publicado en *Information Sciences* donde se pretende alcanzar una máxima cobertura de caminos usando el grafo de control de flujo. Cada rama del grafo se etiqueta y cada rama del programa se instrumenta para generar una etiqueta al ejecutarse. Se selecciona un conjunto de caminos para las pruebas (normalmente los caminos más difíciles de cubrir mediante pruebas aleatorias) y se generan los casos de prueba. A continuación se evalúan los casos de prueba ejecutando el programa para determinar la satisfacción del criterio de prueba. La función de objetivo es una extensión de la distancia de Hamming[1] llamada NEHD (*Normaliza Extended Hamming Distance*) que mide la distancia entre dos caminos teniendo en cuenta, no sólo los elementos que son distintos, sino también la proximidad de los valores distintos. Por ejemplo, dada la cadena de bits `1000`, las dos cadenas `0100` y `0001`, estarían a la misma distancia de Hamming, pero usando la versión extendida, tendríamos que la segunda cadena está más cerca de la primera que la tercera. Las distancias permiten encontrar el camino conocido más cercano al camino objetivo.

Pasamos ahora a describir las ideas fundamentales de la técnica que usa recocido simulado. En su trabajo *Automated program flaw finding using simulated annealing* publicado en *International Symposium on software testing and analysis*, N. Tracey, J. Clark y K. Mander generan los casos de prueba en base a las especificaciones de un sistema. Esta técnica no se usa para la generación de pruebas que optimicen la cobertura de un programa, sino que la idea es probar las condiciones que producen excepciones en un programa. El recocido simulado es un método de búsqueda local que trata de encontrar un conjunto de posibles soluciones, reduciendo la posibilidad de quedarse atascado en un óptimo local *malo* mediante movimientos a soluciones inferiores controlados por un esquema aleatorio.

El último método mencionado es el basado en la búsqueda tabú. En *Tabu search part I,II*, F. Glover presentó una aproximación basada en el algoritmo de los k-vecinos que, junto con el mantenimiento de una lista tabú, evitaba repetir la búsqueda dentro de un área de espacio de soluciones. El trabajo fue publicado en 1989 en *Journal on Computing*.

Todos estos métodos tienen una ventaja con respecto al método de Korel y es el hecho de que pueden evitar los mínimos locales. El algoritmo de Korel descrito anteriormente puede quedarse en un mínimo local, no alcanzando así una solución óptima.

Algoritmos genéticos: Jones *et al.*

Para generar casos de prueba, siempre (excepto quizás en el caso aleatorio) se tiene en cuenta algún dato inicial como puede ser el propio código del programa o un grafo de flujo. Es más, es necesaria una métrica con la que distinguir una buena prueba de otra mediocre y saber si es necesario seguir probando el programa o podemos darnos por satisfechos. La estructura general del método genético es la de ir modificando sucesivamente un conjunto de casos de prueba de forma que se van obteniendo conjuntos mejores según un criterio de medida establecido previamente.

[1] La distancia de Hamming simplemente compara dos cadenas de elementos (bits, letras, etc.) elemento a elemento y cuenta el número de elementos distintos.

El método presentado por Jones y sus colegas tiene como objetivo principal el de obtener casos de prueba de buena calidad según el criterio de cobertura de ramas. Para generar los casos de prueba definieron un algoritmo genético.

Los **algoritmos genéticos** están basados (al menos intuitivamente y en general) en las ideas de Darwin sobre la evolución. La idea general de este tipo de algoritmos es la de combinar muestras o individuos de una población para así obtener mejores individuos que pasen a formar parte de dicha población. Cuando queremos definir un algoritmo genético tenemos que decidir ciertas cosas, como por ejemplo el modo en que representaremos un individuo. Según Jones y sus colegas, la forma más eficiente de representación es usar una única cadena de bits para cada variable de entrada. Asimismo, el conjunto de variables de entrada y por tanto la secuencia de cadenas representará un caso de prueba.

Una población de casos de pruebas estará formada por S tests (siendo S la longitud de la cadena de bits descrita arriba como representación de las variables de entrada). Es decir, normalmente habrá el mismo número de pruebas como número de bits en cada patrón de prueba (datos de entrada).

Usualmente, la población inicial se selecciona de forma aleatoria. A partir de ese momento y una vez elegida la función de optimización con la que se mide la bondad de los individuos (o casos de prueba en nuestro caso), dicha población se irá modificando en función de los propios individuos que la componen y de la función objetivo o de optimización elegida. En el método presentado por Jones *et al.*, se usan funciones basadas en las ramas de los predicados y en el número de iteraciones de los bucles.

Los algoritmos genéticos tienen dos operaciones básicas: el *cruce* y la *mutación*. La implementación de estos dos operadores puede ser muy variada y cada una de ellas da lugar a un algoritmo genético distinto, que de hecho puede tener un comportamiento radicalmente distinto a otro algoritmo genético en cuanto a poblaciones generadas. Aunque existen varias formas de implementar estas dos operaciones, la idea base es siempre la misma. Un cruce puede realizarse de la siguiente forma: se eligen aleatoriamente dos miembros de la población y un punto de las cadenas de bits. Entonces las colas de las dos cadenas de bits (a partir del punto elegido de forma aleatoria) se intercambian. Existen, como ya hemos dicho, variantes de este procedimiento, pudiéndose elegir por ejemplo dos puntos de la cadena e intercambiando la parte central, o intercambiar cada bit de las dos cadenas con una probabilidad cercana al 0.5.

Una *mutación* puede ser simplemente el proceso en el que el valor de cualquier bit en cualquier posición elegida de forma aleatoria es cambiado, pasando de 0 a 1 o de 1 a 0. Existen también en este caso numerosas variantes en cuanto a implementación de una mutación. Nótese que tanto las mutaciones como los cruces pueden dar lugar a datos que no tienen significado lógico en el programa, pero es importante probar el *software* también con estos datos *inválidos*, por lo que a priori no se suelen descartar o filtrar.

Siguiendo las ideas de Darwin, la definición de estos dos operdores no es arbitraria. El cruce modela el comportamiento natural de reproducción dentro de una población, mientras que la mutación modela el cambio o la evolución necesario para la supervivencia de cualquier especie a lo largo de los siglos. Dicho cambio tiene una naturaleza aleatoria, por ello en informática normalmente lo modelamos introduciendo una componente aleatoria en los valores de las variables. En términos matemáticos, esta componente de evolución impide que nuestro algoritmo se quede en un mínimo local de la función objetivo, problema que aparece muchas veces en los algoritmos de búsqueda.

Algunos aspectos de los algoritmos genéticos son especialmente importantes. Por ejem-

plo, la elección de si el número de miembros de la población es fijo o no. Es también importante determinar de forma adecuada la elección de los *padres* cuando se efectúa un cruce. Normalmente, una elección aleatoria ayuda a mantener la diversidad y salud de la población (eligiendo padres similares podríamos hacer que los miembros tuvieran valores similares, inferiores al óptimo). La definición de los valores concretos para todas las cuestiones mencionadas será completamente determinante del comportamiento del sistema (ya sea éste bueno o malo). En cada caso particular hay que valorar y estudiar qué valores nos interesa.

Por último, pero no menos importante, tenemos la elección de los *supervivientes*. Tanto la mutación como el cruce crean nuevos miembros de una población, pero sólo algunos de ellos (y de la población original) sobrevivirán y pasarán a la siguiente iteración. La elección del más cercano al óptimo no suele dar buenos resultados, ya que penaliza la diversidad de la población. Normalmente se hace una elección aleatoria o se usa un mecanismo híbrido entre la elección de los más cercanos al óptimo y la elección aleatoria.

La siguiente figura presenta el esquema general que sigue cualquier algoritmo genético sin entrar en detalles de cómo están definidos los operadores de cruce, de mutación, o el resto de parámetros significativos.

```
Inicializar poblacion
loop
    Determinar el ajuste de la poblacion
    Seleccionar los padres del cruce
    Obtener los miembros nuevos, mutar y determinar su ajuste
    si se alcanza el objetivo entonces salir is
    si se alcanzado limite de iteracion entonces salir is
    Seleccionar los supervivientes de la poblacion y los nuevos miembros
    Formar la nueva poblacion
end loop
```

A continuación se explicará cómo aplicar según Jones *et al.* la idea de los algoritmos genéticos a la generación automática de casos de prueba mediante un ejemplo intuitivo.

Ejemplo 1.4.11 Supongamos que tenemos que probar la siguiente estructura dentro del contexto de un programa:

if D=0 **then** C1 **else** C2 **end if**

D representa una función entera mientras que C1 y C2 corresponden a distintas secuencias de sentencias que se ejecutarán cuando D=0 sea verdadero o falso respectivamente. Supongamos que D es una función complicada que depende de las variabes $x_1, x_2, \ldots, x_n$. Está claro que para probar el comportamiento del programa, tenemos que probar tanto el caso en el que D=0 como en el que D$\neq$0.

La siguiente tabla muestra un ejemplo más concreto, donde D depende sólo de x_1 y x_2.

x_1	x_2	*ajuste*	*padres*	*hijos*	x_1	x_2	*ajuste*	*descripción*
5	4	3.8	`0101 0100`	`0100 1010`	4	10	9.1	cruce en posición 3
2	10	100.0	`0010 1010`	`0011 0100`	3	4	10.0	cruce en posición 3
6	1	2.5	`0110 0001`	`0110 0011`	6	3	2.6	cruce en posición 6
7	3	2.0	`0111 0011`	`0111 1001`	7	9	2.2	cruce en posición 3 y mutación en 5
		108.3					23.9	

Las cuatro primeras columnas son datos referentes a los padres elegidos. Nótese que el valor de la cadena de bits se corresponde con los valores de las entradas. El valor de ajuste de la tercera columna depende del valor de D. En principio no necesitaríamos conocer cuál es la función D, ya que nos bastaría saber el resultado de la función de la que depende la condición para cada caso de prueba. En este ejemplo, $D = x_1^2 - x_2 + 5$. La función objetivo (que nos proporciona el ajuste) en este caso es la inversa del valor absoluto, lo que significa que cuanto más cerca esté D de 0, mayor será el valor de ajuste.

Las siguientes cuatro columnas representan valores relacionados con los nuevos miembros obtenidos, mientras que la última columna describe el método por el que se ha obtenido el nuevo hijo (cruce o mutación).

El ajuste total de los hijos es inferior al de los padres, lo que nos hace ver que las mutaciones y cruces no garantizan en absoluto un mejor ajuste a la función objetivo. La nueva población, es decir, los supervivientes, se elegirán según su ajuste, pero también aleatoriamente. ■

El ejemplo mostrado refleja el caso de generación de un conjunto de casos de prueba para la sentencia condicional, pero cuando tenemos un programa completo se recurre al mantenimiento de un árbol o de un grafo de flujo de control. El algoritmo procede considerando cada vez un punto de decisión e intentando que todas las ramas del grafo (generado a partir del código) sean ejecutadas.

Algoritmos genéticos: GADGET, McGraw *et al.*

GADGET es una herramienta de generación automática de casos de prueba para programas *grandes* escritos en C o C++ basada en los algoritmos genéticos. El método usa el criterio de cobertura de *condición-decisión*. Una condición es una expresión que puede evaluarse a `true` o `false` pero que no contiene otras expresiones booleanas. Una decisión sin embargo es una expresión que influye en el flujo de control del programa. Una cobertura total significaría que todas las ramas en el código fueran ejecutadas al menos una vez, pero además que todas las condiciones que aparecen en el código se evaluen a `true` al menos una vez, y a `false` otra.

Una decisión, por ejemplo $X > 3 \wedge Y = 0$ puede evaluarse a cierto o a falso. Dos casos de prueba que cubrirían estas dos opciones serían, por ejemplo $\{X = 4, Y = 0\}$ y $\{X = 2, Y = 0\}$. Sin embargo, si nos fijamos en los valores de verdad de las expresiones que aparecen en la condición, vemos que para $X > 3$ sí que estamos cubriendo el caso de que sea cierta o falsa, pero para la expresión $Y = 0$ ambos casos de prueba son para el caso en el que se cumple, y no cuando la expresión es falsa. Este ejemplo muestra claramente que haría falta cambiar los casos de prueba, o bien añadir uno nuevo, para poder cubrir todas las posibles evaluaciones de las expresiones que componen la decisión.

El método presentado es un método dinámico que usa las técnicas basadas en algoritmos genéticos para optimizar los resultados (los conjuntos de casos de pruebas). Como ya sabemos, la ventaja de usar algoritmos genéticos para la búsqueda del óptimo radica en el hecho de que los algoritmos genéticos son menos sensibles a los *mínimos locales* gracias a su componente aleatoria.

Además del algoritmo genético tradicional descrito en la sección anterior, GADGET puede usar otro algoritmo genético llamado ***Differential Genetic Algorithm*** que describiremos a continuación: se genera una población inicial igual que en el caso tradicional pero el cruce de individuos se realiza una forma distinta. Para cada elemento de la población I, se eligen de forma aleatoria tres *compañeros* A, B y C (uno de ellos puede ser el mismo I y, además, pueden repetirse). A partir de las características de estos cuatro miembros de la población, se genera un nuevo individuo I' construido como sigue:

- por cada elemento I_i del individuo I, definimos $I'_i = I_i$ con una probabilidad p (siendo p un parámetro del algoritmo)
- Asignamos con probabilidad $1-p$ al elemento del nuevo individuo I'_i el valor resultado del cálculo $A_i + \alpha(B_i - C_i)$ siendo α otro parámetro del algoritmo a determinar

Si el individuo I' tiene una función objetivo mejor que I, lo introducimos en la población y eliminamos I.

Ejemplo 1.4.12 Supongamos que los individuos de la población están representados mediante una cadena de cinco números reales (representados por filas en la tabla de abajo). En este ejemplo el nuevo individuo mantiene tres de los cinco elementos del individuo inicial A ($I = A$). En particular el primero, el tercero y el cuarto. En este ejemplo la probabilidad puede que fuera $p = 2/3$ y se define α =0.4 así pues, el segundo elemento A_2 (encuadraro en la tabla) es el resultado de la función A_2+0.4$(B_2 - C_2)$ =3.0+0.4(1.0−3.0) =2.2 y A_5 se calcula de forma similar.

A	1.0	3.0	7.2	9.0	-1.0
B	8.0	1.0	4.1	0.0	4.0
C	9.0	3.0	6.0	-5.0	1.0
I'	1.0	2.2	7.2	9.0	0.2

■

Lo hemos mencionado anteriormente pero queremos recalcar que en la ejecución de estos métodos genéticos hay que tener en cuenta cuatro factores importantes: la población inicial, la función objetivo, las operaciones de cruce y mutación, y la probabilidad que tienen los individuos de ser seleccionados como padres que, normalmente, está relacionado con el grado de ajuste de cada individuo.

A continuación describiremos el algoritmo usado por GADGET para cumplir con su objetivo: maximizar la cobertura de condición-decisión del código. En primer lugar, para cada condición presente en el código analizado hay que introducir dos requerimientos: que se evalue a `true` al menos una vez, y que se evalúe a `false` al menos una vez también. Una vez definidos los requisitos, se ejecutará el programa una primera vez con una entrada aleatoria (determinada por una semilla) lo que permitirá inicializar la *tabla de cobertura*

donde se almacenarán los requisitos que hayan sido satisfechos hasta un momento determinado. Esta tabla servirá también para seleccionar qué requisito queremos cubrir en una determinada iteración del algoritmo.

Cada vez que se seleccione un requisito, el algoritmo genético iniciará de nuevo su ejecución, la cual terminará o bien dando un resultado, o bien porque ha alcanzado el número máximo de iteraciones permitido. Si el algoritmo genético ha generado una solución que satisface un requisito (ya sea el requisito que se está buscando u otro de la tabla), se almacena en la tabla de cobertura. Una vez alcanzado el requisito seleccionado, se vuelve a iterar seleccionando otro requisito que no haya sido satisfecho hasta el momento y este proceso continúa hasta que hayan sido satisfechos todos los requisitos, o bien se hayan intentado satisfacer todos. Nótese que puede que hayamos intentado satisfacer un objetivo sin éxito. En ese caso consideramos el requisito inalcanzable por el algoritmo genético.

Se puede medir la calidad del generador de casos de pruebas mirando el porcentaje de requisitos de la tabla que ha sido capaz de satisfacer. Podemos también mencionar algunos aspectos importantes en cuanto a la complejidad del algoritmo. Por ejemplo, sabemos que el coste más significativo de la ejecución del algoritmo completo es el coste de la ejecución del programa que se quiere probar. También es importante el coste del algoritmo de optimización de la función objetivo. Otros factores determinantes en la eficiencia del algoritmo son el número de individuos de una población, la probabilidad de mutaciones, el número máximo de iteraciones que puede realizar el algoritmo genético, etc.

Algoritmos genéticos: TGEN. Pargas *et al.*

El criterio de cobertura usado en este método es el de sentencia y el de ramas. Es una técnica guiada por el objetivo que usa un agoritmo genético, a su vez guiado por el control de dependencias del programa. El algoritmo puede implementarse en paralelo usando varios procesadores y balanceando la carga de proceso, obteniendo así un método más eficiente. La principal ventaja de esta aproximación con respecto a otras es su mayor eficiencia.

Antes que nada, vamos a definir qué es un *grafo de dependencia* de un programa, el cual se puede construir a partir del grafo de flujo de control. Ya hemos visto en qué consiste un grafo de flujo, así que nos centraremos en el de dependencias. Dado un grafo de flujo de control G, y sean V y W dos nodos de dicho grafo, podemos decir que W está *post-dominado* por V si todo camino dirigido que va desde W hasta la salida del programa, contiene el nodo V. Podemos también decir que dados dos nodos X e Y del grafo de flujo de control, Y es *control-dependiente* de X si, y sólo si:

1. existe un camino dirigido P que va de X a Y donde todo nodo Z perteneciente al camino P está post-dominado por Y, y

2. X no está post-dominado por Y

Podemos ver que en un grafo de dependencias, los nodos representan sentencias y los arcos representan las dependencias de control entre las sentencias.

Ejemplo 1.4.13 Dado el siguiente programa escrito en pseudocódigo que toma como entrada tres enteros:

```
        integer i, j, k
1       read i, j, k
2       if (i<j)
3         if (j<k)
4           i = k;
5         else k = i;
          endif
        endif
6       print i, j, k
```

El grafo de flujo asociado al programa es el que aparece a la izquierda de la Figura 1.3, mientas que el grafo de dependencias obtenido a partir del de flujo se muestra a la derecha. Los números en el interior de los nodos corresponden a los números asignados en las sentencias en el código del programa. En el grafo de flujo se puede ver por ejemplo, cómo el nodo etiquetado con el número 6 post-domina a todos los nodos del grafo excepto a sí mismo y al nodo de salida. En otras palabras, cualquier camino debe pasar por dicho nodo para llegar a la salida. Sin embargo, los nodos etiquetados con los números 3, 4 y 5 no post-dominan a ningún nodo del grafo porque existe un camino alternativo. El nodo 2 post-domina al nodo 1 y al nodo de entrada y, por último, el nodo 1 post-domina al nodo entrada únicamente.

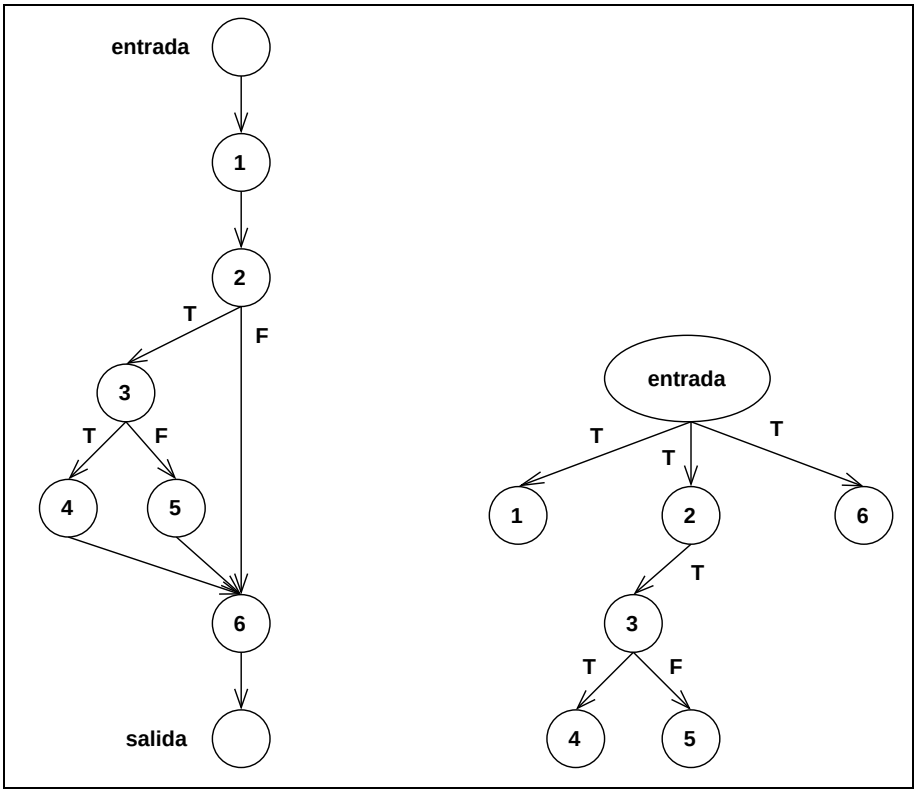


Figura 1.3: Grafo de flujo y de dependencias para el programa

En el grafo de dependencias, mostrado a la derecha de la figura, vemos además que el nodo 4 es dependiente de control del arco $3T$ (el que sale del nodo 3 con etiqueta

`true`) mientras que el nodo 5 es control-dependiente del arco $3F$. Los nodos 1, 2 y 6 son control-dependientes de la entrada en el programa.

■

Un camino acíclico del grafo de dependencias que vaya de la raiz a un nodo del grafo, contendrá el conjunto de predicados (arcos) que tienen que satisfacer las entradas del programa para que la sentencia asociada con el nodo destino sea ejecutada. Por ejemplo, el conjunto de predicados entradaT, 2T y 3T es un camino de predicados dependientes para la ejecución de la sentencia etiquetada con el número 4.

La idea del algoritmo de generación de datos es la de cubrir todos los predicados que aparecen en el grafo de dependencias. Esto quiere decir que la función que evalúa el grado de aceptación de una prueba medirá los predicados cubiertos por la misma. El algoritmo sigue la estructura tradicional de los algoritmos genéticos, guiando la elección de los padres en función del grado de aceptación de los mismos. Nótese cómo durante su ejecución (o la generación de casos de prueba), el objetivo final de cubrir todos los caminos va cambiando conforme se van cubriendo predicados.

Más concretamente, TGEN toma como entrada el programa que se quiere probar, el grafo de dependencias del mismo, un conjunto inicial de datos de prueba (puede ser el conjunto vacío), y un conjunto de requisitos de prueba que tienen que satisfacerse. Lo primero que se hace es inicializar las estructuras de datos auxiliares que usará el algoritmo. Se calculan los caminos acíclicos contenidos en el grafo de dependencias y los requisitos de prueba satisfechos hasta el momento (vacío inicialmente). Estos requisitos pueden estar basados en la cobertura de ramas y/o de sentencias.

El algoritmo entra entonces en un bucle iterativo donde, en cada iteración, obtiene un conjunto de casos de prueba. Para evitar caer en un bucle infinito cuando un requisito sea inalcanzable, se pone un límite en el número de iteraciones permitidas.

Dentro del bucle, como es ya tradicional, se calcula el grado de aceptación de la población y se ordenan los elementos de la misma. Se seleccionan entonces los padres de los nuevos individuos y se procede al cruce o mutación para obtener nuevos miembros. El método para el cruce usado en este algoritmo es el tradicional en algoritmos genéticos (elección de un punto de la cadena de representación en el que se corta e intercambian las colas de la cadena). El proceso de mutación lo que hace es asignar un valor aleatorio a uno de los elementos de la cadena elegido también de forma aleatoria. La relación entre cruces y mutaciones es del 90 % y 10 % de operaciones respectivamente.

Ejemplo 1.4.14 (Continuación) Continuando con el ejemplo descrito antes, vamos a suponer que en un momento dado de la ejecución del método tenemos una población como la descrita en la siguiente tabla:

Caso de prueba	Entrada (i, j, k)	Sentencias
$t1$	1, 6, 9	1, 2, 3, 4, 6
$t2$	0, 1, 4	1, 2, 3, 4, 6
$t3$	5, 0, 1	1, 2, 6
$t4$	2, 2, 3	1, 2, 6

A simple vista podemos ver que las sentencias cubiertas por los individuos de la población no contienen la sentencia 5, así que el algoritmo seleccionará dicha sentencia como objetivo inmediato para intentar definir un caso de prueba que la cubra. Viendo el grafo de

dependencias, vemos que la ejecución de la sentencia 5 depende de los predicados {entryT, 2T, 3F}. Ya dentro del bucle, el algoritmo lo primero que hace es determinar el grado de aceptación de cada individuo de la población en función del objetivo, obteniendo un resultado para dichos casos de prueba de 2, 2, 0 y 0 respectivamente (recordemos que el grado de aceptación se mide en función de la función objetivo que en este caso son las dependencias de la sentencia 5). La dependencia del nodo de entrada se ignora. Una vez calculada la aceptación, se asigna una probabilidad a cada elemento para ser elegido como padre de un cruce o mutación. En este caso se asigna una probabilidad de 0.5 a los dos primeros individuos y de 0.0 a los dos últimos.

Supongamos que se eligen como padres los elementos {(1,6,9), (0,1,4), (0,1,4), (0,1,4)} (podemos tener varias veces el mismo individuo) y que se hace un cruce entre los dos primeros elementos de este conjunto de padres y sendas mutaciones en los dos segundos. Obtendríamos los nuevos individuos $t5$ =(1,6,4), $t6$ =(0,1,9), $t7$ =(0,6,4) y $t8$ =(5,1,4). Tanto los casos de prueba $t5$ como el $t7$ satisfacen la función objetivo, por lo que el algoritmo terminaría ■

Recocido simulado: el método de Tracey *et al.*

El objetivo de este método es el de tener un marco de trabajo general para la generación de casos de prueba abarcando la generación tanto de pruebas de caja blanca como pruebas de caja negra (es decir, abarcar la prueba tanto de propiedades no funcionales como de funcionales).

Como en cualquier método dinámico, lo primero que se tiene que hacer es definir una función objetivo que irá guiando la búsqueda. La función objetivo recordemos que tiene que proporcionar un valor que indique el grado de cercanía de un elemento al objetivo, no basta con que diga si se satisface o no el objetivo.

En nuestro caso, la función objetivo estará definida en función de la precondición y postcondición de un determinado fragmento de código. Para calcular el valor devuelto por la función lo primero que se hará será convertir la precondición y la postcondición *negada* a la forma normal disyuntiva (DNF). Por ejemplo $A \rightarrow (B \vee (C \wedge D)$ se convertirá en $\neg A \vee B \vee (C \wedge D)$. Naturalmente, cualquier solución a cualquiera de las disyunciones será una solución a la expresión completa. A cada tipo de expresión se le asignará una función para calcular el coste, de forma que a cada error le estamos asignando de forma implícita un coste. Cuanto menor sea el coste, más cerca estaremos del objetivo.

El recocido simulado es una técnica de optimización basada en la búsqueda en la vecindad. El algoritmo selecciona soluciones candidatas que están en la vecindad de la solución actual. Siempre se aceptarán las soluciones que tengan un mejor valor de la función objetivo que la solución actual, sin embargo las soluciones que tengan un peor valor se podrán aceptar también aunque de una forma más controlada. El aceptar soluciones *peores* es un mecanismo que nos permite evitar quedar bloqueados en un mínimo local (cosa que puede ocurrir por ejemplo con el método de Korel). Inicialmente se pueden aceptar más soluciones peores, pero conforme se avanza en la búsqueda, la cantidad de peores aceptados va disminuyendo. La función que modera la aceptación de las soluciones que son peores se llama *cooling schedule*, ya que el modelo inicial habla de un parámetro *temperatura* que va disminuyendo conforme se va enfriando la posibilidad de aceptar a *malos* candidatos: a más temperatura mayor maleabilidad, y por lo tanto mayor probabilidad de introducir *malos* casos.

La implementación de este método implica la toma de varias decisiones que pueden afectar al comportamiento del algoritmo. En primer lugar es necesario decidir la representación que se usará para las soluciones (en nuestro caso son casos de prueba) y la vecindad. La vecindad puede darse como una función de cercanía, por ejemplo si tenemos una variable entera, los vecinos serán los que el valor de dicha variable entera esté comprendida en un rango determinado alrededor del valor actual.

2

Interpretación Abstracta

2.1. Introducción

La interpretación abstracta [CC77, Cou, Bru91] es una teoría de aproximación de estructuras matemáticas que puede ser aplicada a la construcción sistemática de métodos y algoritmos eficientes para aproximar problemas indecidibles o de alta complejidad en la informática. Por ejemplo, en los últimos años ha tenido mucho éxito el análisis estático de programas basado en interpretación abstracta que automáticamente infiere propiedades dinámicas de los sistemas [JN95]. El éxito se justifica porque que ha conseguido probar formalmente propiedades realmente complejas de sistemas empotrados, de tiempo real y/o críticos.

Las aplicaciones comunes [Cou01, CC92] de la interpretación abstracta aparecen en

- el diseño de semánticas de lenguajes de programación, lo que nos permite comparar programas a distintos niveles de observación,
- el diseño de métodos de prueba (y sistemas de tipado), que nos permite demostrar propiedades de programas, y
- el diseño de métodos de análisis de programas ya sea un análisis estático o dinámico.

Podemos preguntarnos por qué es necesaria la interpretación abstracta y la respuesta es bastante directa. Deseamos poder inferir, analizar o probar propiedades dinámicas en tiempo de compilación, es decir, estáticamente. La mayoría de problemas relacionados con este análisis son problemas indecidibles o demasiado complejos (NP-completos). Esta elevada complejidad provoca la necesidad de tener métodos que, aun siendo formales, sean parciales de forma que puedan dar una respuesta *correcta* en algunos casos. Con la interpretación abstracta podemos dar *respuestas parciales* a preguntas relacionadas, por ejemplo, con el análisis de flujo de datos. Es decir, un método basado en la interpretación abstracta será capaz de dar como respuesta un *sí*, un *no*, o incluso un *no sé*, en aquéllos casos en que no pueda asegurar una respuesta completamente fiable. Esta última posibilidad muestra claramente la parcialidad de la aproximación [Mar93, MS89].

Objetivos del capítulo

Este capítulo tiene como objetivo introducir la técnica de interpretación abstracta de forma que el lector sea capaz de ver cómo es posible aplicar la metodología para abordar problemas complejos de forma eficaz.

2.1.1. Intuición y puntos cruciales en interpretación abstracta

En esta sección se da la intuición tras la interpretación abstracta mediante la presentación de un ejemplo muy gráfico y simplificado, donde representaremos un sistema mediante sus trazas.

La semántica concreta de un programa puede verse como la formalización del conjunto de todas las posibles ejecuciones del programa en todos los posibles entornos. Sólo con esta definición se puede ver que estamos hablando de un conjunto de trazas o de estados muy grande. Imaginemos que representamos la evolución de los valores de un conjunto de variables a lo largo del tiempo mediante una serie de curvas. Podemos ver el diagrama que tendríamos en la siguiente figura (Figura 2.1), donde cada punto en la línea de tiempo (representada en la horizontal) representa un estado concreto del sistema. De esta forma vemos cómo el sistema evoluciona a lo largo del tiempo.

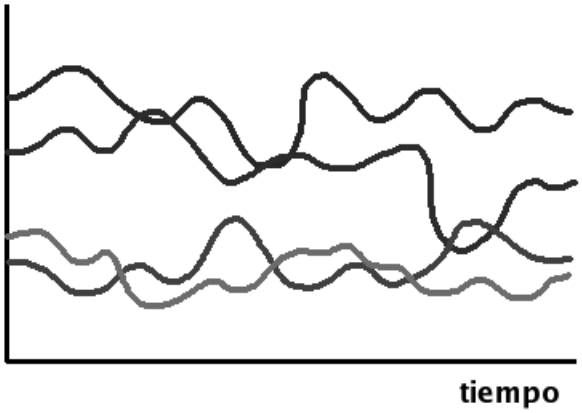


Figura 2.1: Representación de la semántica concreta de un sistema

Como puede imaginarse, la semántica concreta de un programa puede ser un *objeto matemático infinito*, en el sentido de que representa infinitos estados (o de trazas). En estos casos la semántica concreta **no es computable** ya que no es posible escribir un programa capaz de representar y computar todas las posibles ejecuciones de cualquier programa en todos los posibles entornos de ejecución.

Sin embargo, mediante el uso de la interpretación abstracta podemos analizar propiedades de sistemas cuyos estados no son computables. A continuaciíon se mostrarán los beneficios conseguidos al usar la interpretación abstracta para analizar propiedades de seguridad de un sistema. Una propiedad de seguridad (*safety*) expresa condiciones que queremos que nunca se lleguen a dar. Podríamos verlo como un "**nunca debemos llegar a un punto prohibido**". Gráficamente, podríamos definir zonas prohibidas en nuestro espacio de estados (de valores); Zonas donde debemos asegurar que nunca entrarán las curvas. En la Figura 2.2 las dos franjas grises (superior e inferior) y los óvalos en el centro del dibujo representan las zonas prohibidas.

En este contexto, el sistema satisface las propiedades de seguridad si la ejecución del mismo nunca pisa las zonas prohibidas. La prueba (o demostración) de propiedades de este tipo no consiste mas que en demostrar que la interesección entre la semántica del programa y las zonas prohibidas es el conjunto vacío, pero ya sabemos que este problema es indecidible debido a que la semántica concreta no es computable.

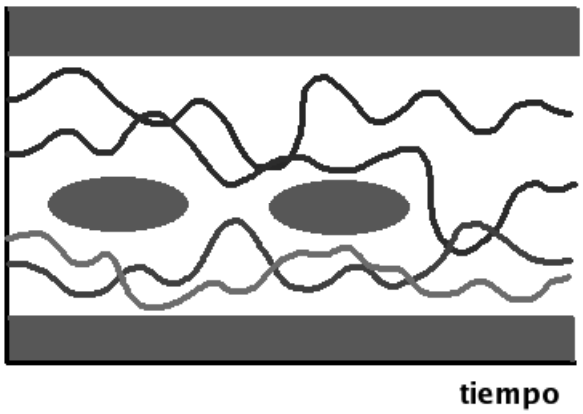


Figura 2.2: Representación de zonas *prohibidas*

En muchos casos, cuando nos encontramos con la situación anterior (una semántica no computable), se recurre a técnicas como el *software testing*, el cual considera y analiza únicamente un subconjunto de las posibles ejecuciones de la semántica concreta. Al considerar sólo un conjunto de las posibles trazas, en *software testing* puede darse el caso de que precisamente una de las ejecuciones no consideradas provoque un error. En la Figura 2.3 representamos con trazos continuos las ejecuciones consideradas y por tanto analizadas, y con trazos discontinuos las que se quedan fuera del análisis en *software testing*.

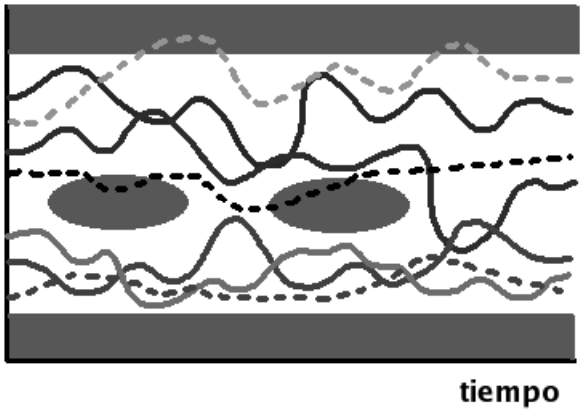


Figura 2.3: Representación de la *carencia de cobertura*

Bajo estas condiciones no se puede asegurar que el análisis sea capaz de detectar todos los *errores* del sitema, ya que aunque la intersección entre las ejecuciones consideradas y las zonas prohibidas es vacía, podemos ver cómo la intersección de la semántica real y las zonas prohibidas no es el conjunto vacío. Esta es la razón por la que el *software testing* no puede considerarse como una *prueba* (demostración) de propiedades. Este problema recibe conmúnmente el nombre de *carencia de cobertura*.

¿Qué puede arreglar la interpretación abstracta?. Si en vez de la semántica concreta, consideramos una semántica abstracta, el problema se transforma y puede verse de forma gráfica en la Figura 2.4.

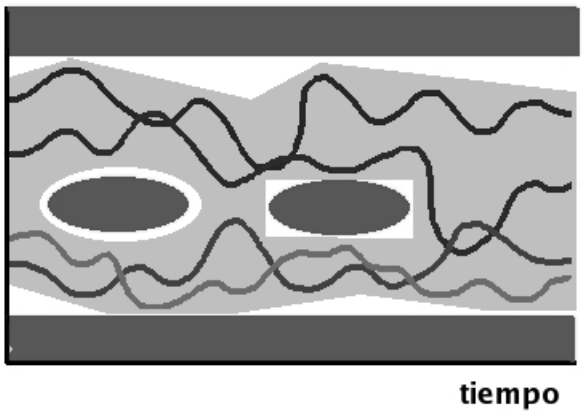


Figura 2.4: Representación de la semántica abstracta

En esta figura, la parte sombreada más clara representa la semántica abstracta del programa. Las franjas sombreadas oscuras siguen siendo las zonas prohibidas, mientras que las curvas representan la semántica concreta (una parte de ella). Hay que fijarse en que la semántica abstracta cubre *todas* las curvas de la semántica concreta. Así pues, la semántica abstracta abarca una zona más amplia que la concreta pero puede representarse de forma finita mediante áreas de polígonos. Es decir, la versión abstracta tiene una representación (simbólica) **computable**, y por tanto manejable. De esta forma, si la intersección de la semántica abstracta con las zonas prohibidas es el conjunto vacío, podemos estar seguros de que la semántica concreta también tendrá una intersección vacía ya que ésta está *incluída* en la abstracta.

Lo más importante cuando aplicamos la técnica de interpretación abstracta para verificar propiedades de seguridad de un sistema es que toda semántica abstracta tiene que ser **correcta**. Es decir, debe cubrir todas las trayectorias concretas, no puede dejarse ninguna trayectoria fuera ya que si no las abarca todas, el resultado positivo del análisis no podrá extrapolarse al mundo concreto. La importancia de este punto puede verse de forma gráfica en la Figura 2.5.

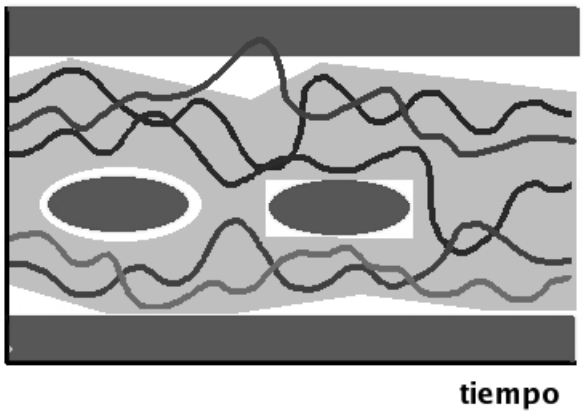


Figura 2.5: Representaciíon de una semántica abstracta no correcta

En la Figura 2.5, la semántica abstracta no cubre toda la concreta. El problema que acarrea esta situación es el siguiente: se puede observar que la intersección de la semántica abstracta con las zonas prohibidas es vacía, por lo que el análisis indicaría que el sistema cumple con las propiedades de seguridad requeridas. Sin embargo hay una curva concreta que sí pasa por una de las zonas prohibidas, y por tanto el sistema en realidad no cumple las propiedades especificadas. En otras palabras, en el mundo concreto - en la realidad - sí que hay un problema que con el análisis basado en interpretación abstracta no hemos sido capaz de detectar debido a que la semántica abstracta no era correcta (no cubría toda la concreta). La no detección del problema puede acarrear graves consecuencias en caso de estar analizando un sistema crítico, de ahí la importancia de trabajar con abstracciones correctas.

Podemos sacar una primera conclusión ya que una de las propiedades más importantes en interpretación abstracta es la corrección de la semántica abstracta, es decir, definirla lo suficientemente amplia como para considerar la semántica concreta al completo. Sin embargo, también es importante llegar a un compromiso entre la corrección y la eficiencia, ya que si abarcamos demasiado con la semántica abstracta podríamos llegar a considerar zonas que realmente nunca van a ser ocupadas por la semántica concreta y sin embargo hacen que la semántica abstracta tenga interesección con las zonas prohibidas. En estos casos hablamos de la introducción de *falsos errores* (o *falsas alarmas*) en la semántica, errores que en la realidad no existen pero que aparecen con la abstracción. En este caso hablamos de abstracciones demasiado **poco precisas**. Vemos un ejemplo gráfico en la Figura 2.6, donde la semántica abstracta considera todas las trazas concretas (por tanto es correcta) pero añade una zona (por la que nunca pasará una traza concreta) cuya intersección con las zonas prohibidas no es el conjunto vacío. El análisis abstracto nos indicará que el programa no satisface las propiedades de seguridad cuando en realidad sí que lo hace.

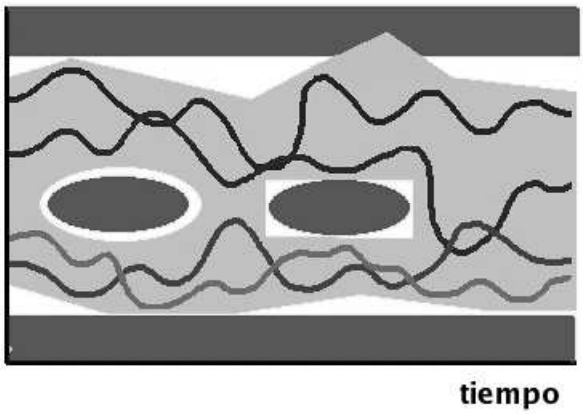


Figura 2.6: Representación de una semántica abstracta poco precisa

En la literatura encontramos algunos casos en los que las aproximaciones se definen en función de un intervalo de tiempo. Es decir, se pone un límite superior temporal que, una vez alcanzado, hace que el análisis se detenga. Esta es una forma trivial de obtener una semántica abstracta (en general) no correcta como se muestra de forma gráfica en la Figura 2.7:

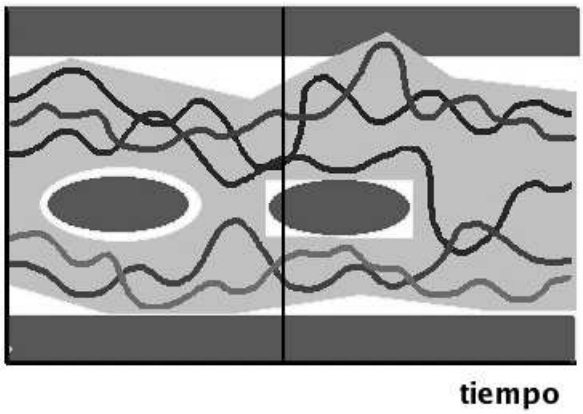


Figura 2.7: Representación de una semántica abstracta incorrecta

En este caso vemos que la abstracción sí que abarca toda la semántica completa, y que tanto la intersección de la semántica concreta como la de la semántica abstracta con la zona prohibida no es vacía. Sin embargo, si nos limitamos a analizar únicamente la semántica abstracta hasta la línea vertical trazada (límite temporal), el análisis no detecta el error. Por lo tanto, una abstracción que incluya un límite de tiempo será siempre peligrosa ya que por definición es no correcta y puede esconder errores más allá del límite impuesto.

Como consecuencia de todo el discurso anterior, tenemos que cuando se definen abstracciones hay que asegurar la corrección de las mismas, en el sentido de que la semántica abstracta tiene que ser siempre un superconjunto de la semántica concreta. Además se debe conseguir que la representación de la semántica abstracta sea lo suficientemente compacta y sencilla como para que lo maneje un computador o algoritmo. De nada nos sirve tener una semántica abstracta que sea no computable (como ocurría con la concreta). El hecho de no tener una abstracción precisa del todo hace que las técnicas de verificación o análisis que usan la interpretación abstracta no sean completas en el sentido de que habrá veces en las que no sea capaz de contestar, aunque sabemos que cuando da una respuesta, ésta es segura, es una demostración. Nótese que esta parcialidad a la hora de contestar es hasta cierto punto lógica ya que muchas veces los problemas que se intentan resolver son indecidibles.

2.1.2. Aspectos metodológicos

Ya hemos dicho que la interpretación abstracta proporciona una forma de diseñar análisis de programas, sistemas de tipos, semánticas, etc. A continuación remarcarmos algunas de sus particularidades.

La interpretación abstracta es capaz de simplificar, descomponer, componer o refinar técnicas de análisis y semánticas, por lo que suele ser una ayuda para **reducir la complejidad** de dichas herramientas, es decir, convertirlas en herramientas eficaces. Además, permite definir métodos formales para probar la **corrección** de programas. Los algoritmos basados en la interpretación abstracta son **universales**, en el sentido de que gracias a su alto nivel de abstracción, son fáciles de entender, reusar y modificar. Asociada de forma natural a la interpretación abstracta aparece una **taxonomía** o clasificación de los análisis,

semánticas o sistemas de tipos ya que éstos pueden compararse según su nivel de precisión (o abstracción). Por último mencionar que gracias al hecho de que la técnica proporciona un marco uniforme para el diseño de semánticas, sistemas de tipos y análisis, éstos son más fáciles de entender y de enseñar. De hecho, todos los métodos de análisis de flujos de datos, la mayoría de las semánticas y de los sistemas de tipos pueden especificarse, diseñarse y estudiarse bajo el marco de trabajo de la interpretación abstracta.

2.2. Fundamentos matemáticos

Vamos a repasar algunas notaciones y propiedades de conjuntos y funciones necesarios para la correcta definición y comprensión de abstracciones de programas y propiedades.

2.2.1. Conjuntos

Muchos conceptos de las matemáticas en realidad pueden verse como conjuntos, por ejemplo las relaciones entre elementos, las propiedades, los sistemas, etc. De ahí la importancia de este apartado que, aunque pueda parecer fuera de contexto, es la base de todo formalismo en la ingeniería del *software*. Empezamos definiendo la notación que se usará a lo largo del texto.

Escribiremos $a \in X$ cuando el objeto a pertenezca al (sea miembro del) conjunto X, mientras que $a \notin X$ expresa que a no pertenece al conjunto X. Por definición, $a \notin X \stackrel{\text{def}}{=} \neg(a \in X)$. Etiquetaremos el símbolo $=$ con las letras "def" ($\stackrel{\text{def}}{=}$) cuando estemos estableciendo una relación (más precisamente una equivalencia) por definición, es decir, una base de nuestra teoría.

Si P y Q son expresiones sobre conjuntos, podemos usar las siguientes abreviaturas: $P \wedge Q$ para la conjunción de conjuntos, $\neg P$ para la complementación de conjuntos, y $\forall x : P$ para la cuantificación sobre conjuntos.

A continuación damos otras definiciones de expresiones sobre conjuntos:

$$\begin{aligned}
P \vee Q &\stackrel{\text{def}}{=} \neg((\neg P) \wedge (\neg Q)) \\
P \Rightarrow Q &\stackrel{\text{def}}{=} (\neg P) \vee Q \\
P \Leftrightarrow Q &\stackrel{\text{def}}{=} (P \Rightarrow Q) \wedge (Q \Rightarrow P) \\
P \veebar Q &\stackrel{\text{def}}{=} (P \vee Q) \wedge \neg(P \wedge Q) \\
\exists x : P &\stackrel{\text{def}}{=} \neg(\forall x : (\neg P)) \\
\exists a \in S : P &\stackrel{\text{def}}{=} \exists a : a \in S \wedge P \\
\exists a_1, a_2 \ldots, a_n \in S : P &\stackrel{\text{def}}{=} \exists a_1 \in S : a_2 \in S, \ldots, a_n \in S : P \\
\forall a \in S : P &\stackrel{\text{def}}{=} \forall a : (a \in S) \Rightarrow P \\
\forall a_1, a_2, \ldots, a_n \in S : P &\stackrel{\text{def}}{=} \forall a_1 \in S : \forall a_2, \ldots, a_n \in S : P
\end{aligned}$$

Para comparar conjuntos se usa una relación que como veremos más adelante establece o define un orden entre los elementos. La siguiente tabla introduce la notación usada:

$$
\begin{array}{lcll}
X \subseteq Y & \stackrel{\text{def}}{=} & \forall a : (a \in X \Rightarrow a \in Y) & \textit{inclusion} \\
X \supseteq Y & \stackrel{\text{def}}{=} & Y \subseteq X & \textit{superconjunto} \\
X = Y & \stackrel{\text{def}}{=} & (X \subseteq Y) \wedge (Y \subseteq X) & \textit{igualdad} \\
X \neq Y & \stackrel{\text{def}}{=} & \neg(X = Y) & \textit{no igualdad} \\
X \subset Y & \stackrel{\text{def}}{=} & (X \subseteq Y) \wedge (X \neq Y) & \textit{inclusion estricta} \\
X \supset Y & \stackrel{\text{def}}{=} & (X \supseteq Y) \wedge (X \neq Y) & \textit{superconjunto estricto}
\end{array}
$$

Además se definen las siguientes operaciones sobre conjuntos:

$$
\begin{array}{lcll}
(Z = X \cup Y) & \stackrel{\text{def}}{=} & \forall a : (a \in Z) \Leftrightarrow (a \in X \vee a \in Y) & \textit{union} \\
(Z = X \cap Y) & \stackrel{\text{def}}{=} & \forall a : (a \in Z) \Leftrightarrow (a \in X \wedge a \in Y) & \textit{interseccion} \\
(Z = X \backslash Y) & \stackrel{\text{def}}{=} & \forall a : (a \in Z) \Leftrightarrow (a \in X \wedge a \notin Y) & \textit{diferencia}
\end{array}
$$

Para definir el conjunto vacío decimos que $\forall a : (a \notin \emptyset)$. El conjunto vacío es único y algunas leyes que lo rigen son:

$$
\begin{array}{ll}
x \backslash \emptyset = x & x \backslash x = \emptyset \\
x \cap (y \backslash x) = \emptyset & x \cap \emptyset = \emptyset \\
x \cup \emptyset = x & \emptyset \subseteq x
\end{array}
$$

Y para acabar, damos alguna notación adicional que usaremos a partir de ahora: $\emptyset$ representa el conjunto vacío tal y como se ha dicho arriba. $\{a\}$ representa un conjunto con un único elemento a. $\{a, b\}$ representa un conjunto con dos elementos y además $a \neq b$. $\{a_1, a_2, \ldots, a_n\}$ representa un conjunto finito de elementos (con n elementos distintos) mientras que $\{a_1, a_2, \ldots, a_n, \ldots\}$ representa un conjunto infinito. Podemos definir conjuntos de forma intensional: $\{a|P(a)\}$ es el conjunto de elementos a para los que la condición $P(a)$ es cierta. P será por tanto una función booleana.

Vamos a hablar ahora de pares, tuplas y otras operaciones sobre conjuntos. Definimos un par como $\langle a, b\rangle \stackrel{\text{def}}{=} \{\{a\}, \{a, b\}\}$, y a las proyecciones primera y segunda respectivamente serán $\langle a, b\rangle_1 = a$ y $\langle a, b\rangle_2 = b$.

Se puede generalizar la noción de tupla a cualquier número de elementos. Así pues, denotamos una tupla como $\langle a_1, \ldots, a_{n+1}\rangle \stackrel{\text{def}}{=} \langle\langle a_1, \ldots, a_n\rangle, a_{n+1}\rangle$. Estará constituida por un número cualquiera pero determinado de elementos n. La proyección i-ésima de la tupla la denotamos como $\langle a_1, \ldots, a_n\rangle_i \stackrel{\text{def}}{=} a_i$. Podemos decir que se cumple la siguiente ley entre tuplas: $\langle a_1, \ldots, a_n\rangle = \langle a'_1, \ldots, a'_n\rangle \Leftrightarrow a_1 = a'_1 \wedge \ldots \wedge a_n = a'_n$.

Producto cartesiano

El producto cartesiano de un par de conjuntos X e Y se define de la siguiente forma:

$X \times Y \stackrel{\text{def}}{=} \{\langle a, b\rangle | a \in X \wedge b \in Y\}$

De nuevo, al igual que ocurría con las tuplas, se puede generalizar el producto cartesiano a tuplas con más de dos componentes de la forma normal; $x_1 \times \ldots \times x_{n+1} \stackrel{\text{def}}{=} (x_1 \times \ldots x_n) \times x_{n+1}$, de forma que $x_1 \times \ldots \times x_n = \{\langle a_1, \ldots, a_n\rangle | a_1 \in x_1 \wedge \ldots \wedge a_n \in x_n\}$

El conjunto potencia $\wp$ de un conjunto X se define como $\wp(X) \stackrel{\text{def}}{=} \{Y | Y \subseteq X\}$. Es decir, es el conjunto que contiene todos los subconjuntos posibles de X (incluyendo el propio X).

La unión arbitraria de conjuntos la definimos $\bigcup Y \stackrel{\text{def}}{=} \{a|\exists X \in Y : a \in X\}$ mientras que la intersección la definimos $\bigcap Y \stackrel{\text{def}}{=} \{a|\forall X \in Y : a \in X\}$, siendo Y un conjunto de conjuntos.

Algunas leyes sobre conjuntos inferidas:

$$\begin{array}{ll} X \cup Y = \bigcup\{X, Y\} & X \cap Y = \bigcap\{X, Y\} \\ \bigcup\{X\} = X & \bigcap\{X\} = X \\ \bigcup \emptyset = \emptyset & \bigcap \emptyset = \{a|true\} \end{array}$$

Aprovechamos para incluir la noción de *familia*. Las familias de conjuntos no son más que conjuntos de conjuntos con un índice. En $X = \{Y_i|i \in I\}$ podemos decir que I es el conjunto que está indexando los elementos de X.

$\bigcup_{i \in I} Y_i \stackrel{\text{def}}{=} \bigcup X = \{a|\exists i \in I : a \in Y_i\}$
$\bigcap_{i \in I} Y_1 \stackrel{\text{def}}{=} \bigcap X = \{a|\forall i \in I : a \in Y_i\}$

Y finalmente, como leyes relacionadas con las familias tenemos

$\forall i \in I : (X_i \subseteq Y) \Rightarrow (\bigcup_{i \in I} X_i \subseteq Y)$
$\forall i \in I : (Y \subseteq X_i) \Rightarrow (Y \subseteq \bigcap_{i \in I} X_i)$

2.2.2. Relaciones

Vamos a ver algunas definiciones y notaciones necesarias para trabajar con relaciones sobre conjuntos.

En primer lugar, diremos que $R \subseteq X$ es una relación unaria sobre el conjunto X, mientras que $R \subseteq X \times Y$ es una relación binaria entre los dos conjuntos X e Y. Podemos leerlo también diciendo que R es un subconjunto del producto cartesiano de los dos conjuntos X e Y, siguiendo los conceptos introducidos en el apartado anterior. Generalizando, $R \subseteq X_1 \times \ldots \times X_n$ decimos que es es una relación n-aria sobre los conjuntos correspondientes.

Tenemos una notación alternativa para representar o trabajar con relaciones. Supongamos que tenemos la relación $R \subseteq X_1 \times \ldots \times X_n$, entonces podríamos hacer referencia a ella también de la siguiente forma:

$R(a_1, \ldots, a_n) \stackrel{\text{def}}{=} \langle a_1, \ldots, a_n \rangle \in R$

Pero para trabajar con relaciones binarias es muy común en el ámbito de la ingeniería del *software* usar una notación con un tinte más gráfico, ya que recuerda a los grafos o diagramas con los que normalmente representamos las relaciones:

$a\, R\, b \stackrel{\text{def}}{=} \langle a, b \rangle \in R$, o bien

$a \stackrel{R}{\rightarrow} b \stackrel{\text{def}}{=} \langle a, b \rangle \in R$

El **dominio** de una relación se define como $dom(R) \stackrel{\text{def}}{=} \{a|\exists b : \langle a, b \rangle \in R\}$ mientras que el **rango** (o codominio) se define como $rng(R) \stackrel{\text{def}}{=} \{b|\exists b : \langle a, b \rangle \in R\}$. Estos dos conceptos se usarán continuamente a lo largo del tema.

A continuación damos algunas leyes o propiedades que pueden tener las relaciones binarias. Sea $R \subseteq X \times X$ la relación binaria sobre el conjunto X (es decir, tanto la primera como la segunda componente de la relación pertenecerán al conjunto X),

$\forall a \in X : (a\, R\, a)$	*reflexividad*
$\forall a, b \in X : (a\, R\, b) \Leftrightarrow (b\, R\, a)$	*simetria*
$\forall a, b \in X : (a\, R\, b \wedge a \neq b) \Rightarrow \neg(b\, R\, a)$	*antisimetria*
$\forall a, b \in X : (a \neq b) \Rightarrow (a\, R\, b \vee b\, R\, a)$	*conexion*
$\forall a, b, c \in X : (a\, R\, b) \wedge (b\, R\, c) \Rightarrow (a\, R\, c)$	*transitividad*

Además, igual que pasa con los conjuntos, podemos operar también con relaciones, sobretodo si pensamos en ellas como conjuntos de tuplas. A continuación damos algunas operaciones sobre relaciones:

$\emptyset$	*relacion vacia*
$1_X \stackrel{\text{def}}{=} \{\langle a, a\rangle \mid a \in X\}$	*identidad*
$R^{-1} \stackrel{\text{def}}{=} \{\langle b, a\rangle \mid \langle a, b\rangle \in R\}$	*inversion*
$R_1 \circ R_2 \stackrel{\text{def}}{=} \{\langle a, c\rangle \mid \exists b : \langle a, b\rangle \in R_1 \wedge \langle b, c\rangle \in R_2\}$	*composicion*
$R_1 \cup R_2$	
$R_1 \cap R_2$	
$R_1 \backslash R_2$	

Para acabar con las operaciones entre relaciones, vamos a introducir el concepto de cierre. De las definiciones que proponemos, la del cierre transitivo y reflexivo de una relación será la de más interés en nuestro caso. El cierre se define a partir de las potencias de la relación como sigue. Dada una relación R,

$R^0 \stackrel{\text{def}}{=} 1_X$	*potencias*
$R^{n+1} \stackrel{\text{def}}{=} R^n \circ R (= R \circ R^n)$	
$R^* \stackrel{\text{def}}{=} \bigcup_{n \in \mathbb{N}} R^n$	*cierre reflexivo y transitivo*
$R^+ \stackrel{\text{def}}{=} \bigcup_{n \in \mathbb{N} \backslash \{0\}} R^n$	*cierre transitivo estricto*

2.2.3. Equivalencias y particiones

Directamente relacionadas con la noción de *relación* aparece los conceptos de equivalencia y de partición. Podemos decir que una relación binaria R sobre un conjunto X es una *relación de equivalencia* si, y sólo si, goza de las propiedades de reflexión, simetría y transitividad.

Siguiendo esta idea, y cumpliendo R con las propiedades mencionadas, $[a]_R \stackrel{\text{def}}{=} \{b \in X \mid a\, R\, b\}$ es una clase de equivalencia. Por otro lado, escribimos $X/_R \stackrel{\text{def}}{=} \{[a]_R \mid a \in X\}$ para denotar el cociente de X según R, es decir, la división de los elementos de X en las clases de equivalencia definidas por la relación.

Podemos decir que P es una partición del conjunto X si y sólo si P es una familia de conjuntos disjuntos que cubren X:

$\forall Y \in P : (Y \neq \emptyset)$
$\forall Y, Z \in P : (Y \neq Z) \Rightarrow (Y \cap Z = \emptyset)$
$X = \bigcup P$

Las particiones tienen una estrecha relación con las equivalencias ya que es sabido que si P es una partición del conjunto X, entonces $\{\langle a, b\rangle \mid \exists Y \in P : a \in Y \wedge b \in Y\}$ será una relación de equivalencia. Y en la dirección opuesta, si R es una relación de equivalencia sobre X, entonces $\{[a]_R \mid a \in X\}$ será una partición de X.

2.2.4. Órdenes sobre conjuntos

Podemos establecer relaciones entre conjuntos mediante la definición de órdenes. Un *orden parcial* sobre conjuntos se define como sigue:

Definición 2.2.1 (Orden parcial) *Una relación $\subseteq$ es un orden parcial sobre conjuntos si es reflexiva, antisimétrica y transitiva. Recordemos:*

$X \subseteq X$	*reflexividad*
$(X \subseteq Y \wedge Y \subseteq X) \Rightarrow (X = Y)$	*antisimetria*
$(X \subseteq Y) \wedge (Y \subseteq Z) \Rightarrow (X \subseteq Z)$	*transitividad*

Si eliminamos la reflexividad del conjunto de propiedades del orden, entonces obtenemos un orden parcial estricto tal y como se muestra en la siguiente definición:

Definición 2.2.2 (Orden parcial estricto) *La relación $\subset$ es un orden parcial estricto sobre X si cumple:*

$\neg(X \subset X)$	*irreflexividad*
$(X \subset Y) \wedge (Y \subset Z) \Rightarrow (X \subset Z)$	*transitividad*

Como ya hemos mencionado, un orden parcial es una relación reflexiva, antisimétrica y transitiva. Como ejemplos de órdenes parciales tenemos la relación $\leq$ sobre los naturales, sobre enteros, la relación de inclusión $\subseteq$ sobre conjuntos, el orden lexicográfico, alfabético, etc.

Las relaciones de orden parcial se suelen expresar en notación infija usando símbolos como $\leq$, $\sqsubseteq$, $\subseteq$, etc. con el significado $\leq = \{\langle a, b\rangle | a \leq b\}$. Los símbolos para las respectivas relaciones inversas suelen ser $\geq$, $\sqsupseteq$, $\supseteq$, etc. representando $\geq = \leq^{-1} = \{\langle b, a\rangle | a \leq b\}$. Por último, la negación o complementación suele denotarse como $\not\leq$, $\not\sqsubseteq$, $\not\subseteq$, etc. representando $\not\leq = \{\langle a, b\rangle | \neg(a \leq b)\}$. Con respecto a las relaciones de orden parcial estricto, los símbolos que suelen utilizarse son $<$, $\sqsubset$, $\subset$, etc. para decir $< = \{\langle a, b\rangle | a \leq b \wedge a \neq b\}$.

A continuación vamos a dar la definición del concepto de conjunto parcialmente ordenado, que como puede imaginarse estará relacionado con una relación de orden parcial. Este concepto es crucial en interpretación abstracta ya que es uno de los pilares de la estructura fundamental de la técnica.

Definición 2.2.3 (Poset) *Un conjunto parcialmente ordenado (poset) es un par $\langle X, \leq\rangle$ donde X es un conjunto, y $\leq$ es una relación de orden parcial sobre X.*

Se cumple es que si $\langle x, \leq\rangle$ es un poset y es cierto que $Y \subseteq X$, entonces $\langle Y, \leq\rangle$ también será un poset. Es decir, que la relación de orden parcial de un poset puede aplicarse a un subconjunto de elementos del mismo dando lugar a otro poset. Nótese que un orden parcial no exige que la relación goce de la propiedad de conexión.

Antes hemos mencionado la posibilidad de representar las relaciones binarias mediante diagramas. Más concretamente, ahora decimos que un diagrama de Hasse representa un poset de forma gráfica, definiendo vértices para cada elemento de X y arcos entre elementos *directamente* relacionados por $\leq$. Se representarán los elementos menores en la parte inferior del diagrama y los mayores en la parte superior, siempre según la relación de orden del poset.

Para definir formalmente los diagramas de Hasse necesitamos introducir la noción de *cobertura*. La relación de cobertura de un poset $\langle X, \leq \rangle$ se define como sigue:

$a - < b \stackrel{\text{def}}{=} (a < b) \wedge \neg(\exists c \in X : a < c < b)$

Es decir, decimos que b cubre a, o que a es cubierto por b cuando b sea estrictamente mayor que a, pero además no debe existir ningún elemento entre ellos en la relación. En otras palabras, deben estar directamente relacionados. Usando la relación de cobertura podemos dibujar diagramas de Hasse que representarán órdenes parciales.

A continuación daremos algunos resultados relacionados con los órdenes entre conjuntos. Estos resultados serán de utilidad para la formalización de las abstracciones. En particular sobre elementos de conjuntos. En primer lugar veremos la relación entre órdenes parciales y órdenes parciales estrictos desde un punto de vista constructivo:

Teorema 2.2.4 *Si $<$ es un orden parcial estricto sobre X, entonces $\leq$ definido como $a \leq b \Leftrightarrow a < b \vee a = b$ es un orden parcial sobre X.*

Teorema 2.2.5 *Si $\leq$ es un orden parcial sobre X, entonces $<$ definido como $a < b \Leftrightarrow a \leq b \wedge a \neq b$ es un orden parcial estricto sobre X*

Otro tipo de relación es la llamada *preorden*. Un preorden es una relación binaria que goza de las propiedades de reflexividad y transitividad, pero no tiene por qué ser antisimétrica.

Teorema 2.2.6 *Si $\preceq$ es un preorden sobre X, entonces $a \equiv b \stackrel{\text{def}}{=} (a \preceq b) \wedge (b \preceq a)$*

Podemos definir la restricción de un poset a un subconjunto determinado de elementos. Si R es una relación binaria sobre X y dado un subconjunto $Y \subseteq X$, entonces la restricción de R en función de Y es:

$R|_Y \stackrel{\text{def}}{=} \{\langle a, b\rangle \in R | a, b \in Y\}$

En otras palabras, se considerarán únicamente los elementos del subconjunto Y descartando el resto de elementos.

Teorema 2.2.7 *Si $\langle X, \leq \rangle$ es un poset e $Y \subseteq X$, entonces $\langle X, \subseteq |_Y \rangle$ también es un poset.*

Teorema 2.2.8 *El único orden parcial que es también una relación de equivalencia es la igualdad.*

Teorema 2.2.9 *El inverso de un orden parcial es también un orden parcial.*

Más adelante haremos uso de la noción de *cadena*. Una cadena de un poset $\langle X, \leq \rangle$ es un subconjunto $Z \subseteq X$ que cumple que $\forall a, b \in Z : (a \leq b) \vee (b \leq a)$. Es decir, que todo par de elementos del conjunto (de la cadena) está relacionado entre sí. Decimos que un poset (completo) es una cadena si X es una cadena, es decir, si Z coincide con X.

Por otro lado, una anti-cadena de un poset es un subconjunto $Z \in X$ en el que $\forall a, b \in Z : (a \leq b) \Rightarrow (a = b)$.

Dos nociones muy importantes dentro del contexto de los retículos que veremos en la siguiente sección son las nociones de maximales (minimales) y de máximos (y mínimos):

$MIN(X) \stackrel{\text{def}}{=} \{m \in X | \neg(\exists a \in X : a < m)\}$
$min(X) \in X \wedge \forall a \in X : min(X) \leq a$
$MAX(X) \stackrel{\text{def}}{=} \{M \in X | \neg(\exists a \in X : M < a)\}$
$max(X) \in X \wedge \forall a \in X : a \leq max(X)$

Lo que realmente nos interesa de estas nociones es el hecho de que un poset $\langle P, \leq \rangle$ tendrá un elemento *top*, supremo o máximo $\top$ si y sólo si $\top \in P \wedge \forall a \in P : a \leq \top$. De forma dual, $\bot$ será el elemento *bottom*, ínfimo, o mínimo de un poset si y sólo si $\bot \in P \wedge \forall a \in P : \bot \leq a$. Por la propiedad de antisimitría, si existen estos dos elementos son únicos.

Ejercicio 2.2.10 *Demostrar que la afirmación anterior es correcta. Es decir, que si existen el $\top$ y el $\bot$, son únicos.*

Hablaremos ahora de los *límites superior e inferior* de un poset. Estos elementos no tienen por qué pertenecer al conjunto de elementos del mismo. Para su definición vamos a usar la inclusión de conjuntos. Sea $\langle P, \leq \rangle$ un poset. Entonces $M \in P$ es un límite superior del subconjunto $S \subseteq P$ si, y sólo si $\forall x \in S : x \leq M$. Nótese que el elemento M pertenece al superconjunto P, de forma que puede que pertenezca o no al conjunto S. De forma dual, decimos que $m \in P$ es un límite inferior de $S \subseteq P$ si, y sólo si $\forall x \in S : m \leq x$.

Hilando más fino, ahora vamos a definir los conceptos de *mayor límite inferior* y *menor límite superior*. Es decir, los límites más "cercanos" al conjunto S. Sea $\langle P, \leq \rangle$ un poset y $X \subseteq P$. El menor límite superior de X, si existiera, es un elemento x que cumple que es un límite superior de X, y que además es el menor entre los límites superiores de X, es decir, que $\forall u \in P : (\forall y \in X : u \geq y) \Rightarrow (x \leq u)$. Al menor límite superior de X lo llamaremos $lub(X)$. Definimos el mayor límite inferior de X de forma dual al *lub*, y lo denotamos como $glb(X)$.

Los posets gozan de ciertas propiedades relacionados con los conceptos de límites introducidos. Por ejemplo,

- Sea $\langle P, \leq \rangle$ un poset y $X \subseteq P$. Si $lub(X)$ existe, entonces es único.
- Sea $\langle P, \leq \rangle$ un poset y $X \subseteq P$. Si $glb(X)$ existe, entonces es único

2.2.5. Retículos

La noción de retículo (*lattice* en inglés) es crucial para la interpretación abstracta. Los retículos definen la estructura de nuestros dominios, tanto abstractos como concretos y establecerán las relaciones entre elementos de nuestro dominio. Sin ellos no podríamos razonar sobre los sistemas, o sobre los estados del sistema. Empezaremos dando la definición de retículo, el cual está compuesto de dos semi-retículos con distintas características.

Definición 2.2.11 (Join semi lattice) *Un* Join semi lattice $\langle P, \leq, \sqcup \rangle$ *es un poset* $\langle P, \leq \rangle$ *tal que cualquier par de elementos $x, y \in P$ tienen un lub (menor límite superior) $x \sqcup y$.*

Es decir, que son posets con una condición adicional: que tengan un menor límite superior. Por otro lado definimos el otro tipo de retículo:

Definición 2.2.12 (Meet semi lattice) *Un* Meet semi lattice $\langle P, \leq, \sqcap \rangle$ *es un poset* $\langle P, \leq \rangle$ *tal que cualquier par de elementos* $x, y \in P$ *tienen un glb (mayor límite inferior)* $x \sqcap y$.

Un *meet semi lattice* puede verse como la noción dual al anterior, ya que en este caso lo que se exige al poset es que tenga un mayor límite superior. Uniendo estos dos conceptos duales obtenemos un retículo como muestra la siguiente definición:

Definición 2.2.13 (Lattice) *Un retículo se define como la tupla* $\langle P, \leq, \sqcup, \sqcap \rangle$ *donde* $\langle P, \leq, \sqcup \rangle$ *es un join semi lattice y* $\langle P, \leq, \sqcap \rangle$ *es un meet semi lattice.*

Los retículos cumplen con una serie de propiedades interesantes. Por ejemplo:

Teorema 2.2.14 *En un* join semi lattice $\langle P, \leq, \sqcup \rangle$ *tenemos para todo elemento* $a, b \in P$ *que* $a \leq b \Leftrightarrow a \sqcup b = b$

Teorema 2.2.15 *En un* meet semi lattice $\langle P, \leq, \sqcap \rangle$ *tenemos para todo elemento* $a, b \in P$ *que* $a \geq b \Leftrightarrow b = b \sqcap a$

Y uniendo estos dos resultados obtenemos el siguiente resultado para un retículo:

Teorema 2.2.16 *En un* lattice $\langle P, \leq, \sqcup, \sqcap \rangle$ *tenemos para todo elemento* $a, b \in P$ *que* $a \leq b \Leftrightarrow a \sqcup b = b \Leftrightarrow a = a \sqcap b$

En un *join semi lattice* se cumplen las siguientes propiedades algebraicas: asociatividad, conmutatividad e idempotencia. En un retículo tenemos, además de éstas, absorción. A continuación vamos a dar la definición de *complete lattice* que, intuitivamente, dice que para todo subconjunto del dominio ha de existir un límite superior menor dentro de P.

Definición 2.2.17 (Retículo completo) *Un retículo completo decimos que es un poset* $\langle P, \sqsubseteq \rangle$ *tal que cualquier subconjunto* $X \subseteq P$ *tiene un menor límite superior lub en* P .

Si cogemos cualquier conjunto finito S de elementos, $\langle \wp(S), \subseteq, \cup, \cap \rangle$ es siempre un retículo completo.

Todo retículo completo tiene un elemento $\top$ y un $\bot$, por lo que un retículo completo nunca será el conjunto vacío. Además, los retículos completos tienen siempre tanto menor límite superior (*lub*) como mayor límite inferior (*glb*).

2.2.6. Funciones

En esta sección introducimos la conexión que existe entre el concepto de función y el de relación. Una función de aridad n sobre un conjunto de elementos X es una relación R de aridad $n + 1$ sobre X para la que para cada $a \in dom(R)$, existe como mucho un elemento $b \in rng(R)$ para el que $\langle a, b \rangle \in R$. O dicho de otra forma:

$(\langle a, b \rangle \in R \wedge \langle a, c \rangle \in R) \Rightarrow (b = c)$

Es decir, es una relación donde cada elemento del dominio se relaciona únicamente con un elemento distinto del codominio (o rango). Si utilizamos la notación funcional en vez de la relacional, escribimos $R(a_1, \ldots, a_n) = b$ de forma equivalente a escribir la forma relacional $\langle a_1, \ldots, a_n, b \rangle \in R$.

Escribimos $f : X \mapsto Y$ para denotar el conjunto de funciones totales f para las cuales $dom(f) = X$ y $rng(f) \subseteq Y$. Es decir, que todo elemento de X forma parte del dominio de la función f mientras que puede que sólo una parte del conjunto Y sea miembro del rango de f. Por otro lado, $f : X \nmapsto Y$ denota el conjunto de funciones parciales f tales que $dom(f) \subseteq x$ y $rng(f) \subseteq y$. Es decir, que puede que haya elementos de X que no pertenezcan al dominio de la función f, lo que significaría que la función no estaría definida para dichos valores.

Igual que ocurría con los conjuntos y las relaciones, podemos definir operaciones que manejen funciones. Por ejemplo,

$f = \lambda a.k \stackrel{\text{def}}{=} \{\langle a, k\rangle \mid a \in dom(f)\}$	*funcion constante*
$1_X \stackrel{\text{def}}{=} \{\langle a, a\rangle \mid a \in X\}$	*funcion identidad*
$f \circ g \stackrel{\text{def}}{=} \lambda a.f(g(a))$	*composicion de funciones*
$f \restriction u \stackrel{\text{def}}{=} f \cap (u \times rng(f))$	*restriccion de funciones*
$f^{-1} \stackrel{\text{def}}{=} \{\langle f(a), a\rangle \mid a \in dom(f)\}$	*inversion de funcion*

Decimos que una función es *inyectiva* si distintos elementos del dominio tienen distintos elementos imagen: $\forall a, b \in X : a \neq b \Rightarrow f(a) \neq f(b) \Leftrightarrow \forall a, b \in X : f(a) = f(b) \Rightarrow a = b$. Nótese la diferencia de concepto con respecto a la definición de función total y parcial discutida arriba. Una función es sobreyectiva (o suryectiva) si todos los elementos del rango son imagen de algún elemento del dominio: $\forall b \in Y : \exists a \in X : f(a) = b$. Y para acabar, una función es biyectiva si es tanto inyectiva como sobreyectiva. También se dice que una función biyectiva es un isomorfismo y decimos que dos conjuntos son isomorfos si existe un isomorfismo entre ellos. La inversa de una función biyectiva se define: $f^{-1} = \{\langle b, a\rangle \mid \langle a, b\rangle \in f\}$.

2.2.7. Conexiones de Galois

En esta sección se define la noción de conexión de Galois. Las conexiones de Galois, definidas en función de poset's, son la base fundamental de la interpretación abstracta. Antes de entrar en la definición formal de este concepto, se definen los *operadores de cierre* por arriba y por abajo. Estos son operadores que trabajan sobre conjuntos y nos serán de gran utilidad a continuación.

De forma general, podemos decir que un operador sobre un conjunto de elementos P no es mas que un mapeo de P en P. De forma más precisa, un *operador de cierre por arriba* (uco) ρ sobre un poset $\langle P, \leq\rangle$ es un operador extensivo ($\forall x \in P : x \leq \rho(x)$), monótono ($\forall x, y \in P : (x \leq y) \Rightarrow (\rho(x) \leq \rho(y))$) e idempotente ($\rho(\rho(x)) = \rho(x)$). Un *operador de cierre por abajo* es el concepto dual al operador de cierre por arriba, es decir, que es un operador reductivo ($\forall x \in P : \rho(x) \leq x$), monotono e idempotente.

Teorema 2.2.18 *Un operador ρ sobre un poset $\langle P, \leq\rangle$ es un operador de cierre por arriba si, y sólo si*

$$\forall x, y \in P : x \leq \rho(y) \Leftrightarrow \rho(x) \leq \rho(y)$$

Podemos ver esta propiedad como el hecho de que el operador consigue que se mantenga la relación entre elementos del conjunto ya sea aplicando o no aplicando el operador a los mismos.

Una característica interesante de los operadores de cierre es que se pueden identificar usando sus puntos fijos. Recordemos que el conjunto de puntos fijos de un operador $f \in P \mapsto P$ sobre un conjunto P es $\{x|f(x) = x\}$.

Teorema 2.2.19 *Un operador de cierre está completamente representado por sus puntos fijos*

Y llegamos así a la clave de la interpretación abstracta. La noción de *conexión de Galois* es la más importante en el marco de la interpretación abstracta ya que todo gira alrededor suyo.

Definición 2.2.20 (Conexión de Galois) *Sean $\langle P, \leq\rangle$ y $\langle Q, \sqsubseteq\rangle$ posets, decimos que un par $\langle \alpha, \gamma\rangle$ de mapeos $\alpha \in P \mapsto Q$ y $\gamma \in Q \mapsto P$ es una conexión de Galois si, y sólo si:*

$\forall x \in P : \forall y \in Q : \alpha(x) \sqsubseteq y \Leftrightarrow x \leq \gamma(y)$

Es decir, que si aplicamos el operador α (de abstracción) a un elemento x, el resultado estará incluído en un elemento y del segundo conjunto (el dominio abstracto) si, y sólo si se cumple que el resultado obtenido de aplicar el operador (o función) γ (de concreción) al elemento y, es mayor que el elemento x (del dominio concreto).

Esquemáticamente representamos una conexión de Galois como sigue:

$$\langle P, \leq\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle Q, \sqsubseteq\rangle$$

Algunas propiedades de las conexiones de Galois son que, por ejemplo, dado $\langle P, \leq\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle Q, \sqsubseteq\rangle$, tanto α como β son monótonas. Además, la identidad $1_P \leq \gamma \circ \alpha$ y $\alpha \circ \gamma \sqsubseteq 1_Q$. Más propiedades: $\alpha \circ \gamma \circ \alpha = \alpha$ y $\gamma \circ \alpha \circ \gamma = \gamma$. $\alpha \circ \gamma$ es un operador de cierre por abajo sobre $\langle Q, \sqsubseteq\rangle$ mientras que $\gamma \circ \alpha$ es un operador de cierre por arriba sobre $\langle P, \leq\rangle$.

Usando otras palabras, podemos decir que $\alpha : \wp(P) \to Q$ tiene la propiedad de que $\forall y \in Q, y = \alpha(\gamma(y))$. Nótese que al aplicar esta operación no se produce ninguna pérdida de precisión, ya que obtenemos exactamente el elemento y del que se parte. Además, escribiremos que $\alpha(x) = y$ (leído "y es la abstracción de x") cuando y sea el *menor* elemento de Q que describa (o cubra) x.

Por otro lado, podemos decir que $\gamma : Q \to \wp(P)$ tiene la propiedad de que $\forall x \in \wp(P), x \subseteq \gamma(\alpha(x))$. Nótese que en este caso, al contrario de lo que ocurriía en el anterior, la aplicación del operador sí que acarrea una pérdida de precisión ya que el resultado no tiene por qué coincidir con el elemento x del que se ha partido. Además, escribiremos que $\gamma(y) = x$ (leído "x es la concreción de y") cuando x sea el *mayor* elemento de $\wp(P)$ al que describa (o cubra) y.

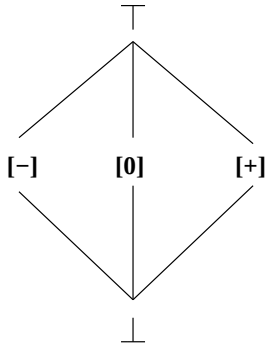
$$\boxed{(\wp(P), \subseteq) \text{ es un retículo completo.}}$$

A veces podemos hacer referencia a $\wp(P)$ escribiendo simplemente $\mathcal{C}$.

Vamos a clarificar todas estas nociones explicando un ejemplo sencillo.

Ejemplo 2.2.21 (Los signos) Imaginemos que tenemos una variable cuyo dominio son los números enteros y queremos hacer una abstracción según el signo (positivo, negativo o cero) de los valores del dominio.

El dominio abstracto podríamos describirlo mediante el siguiente diagrama de Hasse, el cual representa nuestro retículo completo:



La función de abstracción se definiría de la forma natural, siendo cada una de las clases de la figura la abstracción de los elementos negativos, del cero, y de los elementos positivos del dominio concreto respectivamente

Por otro lado, la función de concreción podría definirse mediante la siguiente especificación:

$$\gamma([-]) = \{x \in \mathbb{Z} | x < 0\}$$
$$\gamma([0]) = \{0\}$$
$$\gamma([+]) = \{x \in \mathbb{Z} | x > 0\}$$
$$\gamma(\top) = \mathbb{Z}$$
$$\gamma(\bot) = \emptyset$$

■

Analicemos las relaciones que existen entre las dos funciones que forman la conexión de Galois. Por ejemplo,

$$\alpha(X) \subseteq Y \Leftrightarrow X \subseteq \gamma(Y)$$

Es decir, que si la abstracción de un elemento concreto (o conjunto de elementos concretos) es Y, entonces se cumple que la concreción del conjunto Y contendrá el elemento concreto (o conjunto de elementos concretos) inicial.

Aunque ya se haya mencionado antes, repetiremos algunos conceptos importantes. Por ejemplo, sabemos que la abstracción de X ($\alpha(X)$) es la mejor aproximación para X ya que se cumple que

- X estará siempre contenido en la concreción de su abstracción:

  $X \subseteq \gamma(\alpha(X))$

- si X está en la concreción de cualquier conjunto abstracto Y (es decir, si Y es una aproximación cualquiera de X), entonces Y contendrá la abstracción de X:

  $X \subseteq \gamma(Y) \Rightarrow \alpha(X) \subseteq Y$

  Y esto implica que $\alpha(X)$ es más preciso que cualquier otra aproximación Y.

Se cumple también que las dos funciones, la de abstracción y la de concreción, son monótonas:

- $X \subseteq \gamma(\alpha(X))$ y
- $\alpha(\gamma(X)) \subseteq X$

- si $Q \subseteq Q'$, entonces $\gamma(\alpha(Q)) \subseteq \gamma(\alpha(Q'))$
- $\gamma\alpha \circ \gamma\alpha = \gamma\alpha$, o dicho con palabras: la composición de las dos funciones es idempotente

Teorema 2.2.22 *Una conexión de galois*
$\langle P, \leq\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle Q, \sqsubseteq\rangle \Leftrightarrow \alpha$ *es monotona,* γ *es monotona,* $\alpha \circ \gamma$ *es reductora y* $\gamma \circ \alpha$ *es extensiva.*

Además, se cumple un principio de dualidad entre conexiones de Galois, por lo que

Teorema 2.2.23 $\langle P, \leq\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle Q, \sqsubseteq\rangle$ *si y sólo si* $\langle Q, \sqsupseteq\rangle \underset{\gamma}{\overset{\alpha}{\leftrightarrows}} \langle P, \geq\rangle$

Y, por último, la composición de dos conexiones de Galois da como resultado otra conexión de Galois:

Teorema 2.2.24 *Si* $\langle P, \leq\rangle \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} \langle Q, \sqsubseteq\rangle$ *y* $\langle Q, \sqsubseteq\rangle \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} \langle R, \preceq\rangle$

entonces $\langle P, \leq\rangle \underset{\alpha_2 \circ \alpha_1}{\overset{\gamma_1 \circ \gamma_2}{\leftrightarrows}} \langle R, \preceq\rangle$

2.3. Abstracción de dominios

Una vez presentada la base matemática que sustenta y fundamenta cualquier análisis de sistemas o programas basado en interpretación abstracta, en esta sección se muestra cómo se deben aplicar las nociones introducidas dentro del marco del desarrollo de *software*.

A partir de ahora vamos a identificar el *mundo real* con el *mundo concreto*, mientras que el *mundo abstracto* será el *mundo aproximado*. De esta forma definiremos *conexiones de Galois* entre estos dos mundos que nos permitirán abstraer y concretar elementos de ambos mundos. Es decir, los objetos pertenecientes a cada uno de los dos mundos (real y aproximado) están relacionados mediante una *conexión de Galois*.

En la sección anterior hemos visto un ejemplo donde se definían las funciones α y γ para el dominio de los enteros en función del signo que tuvieran. Ahora vamos a ver un ejemplo más concreto y a recalcar algunas características del marco de trabajo.

Ejemplo 2.3.1 (Abstracción del producto) Supongamos que, como en el Ejemplo 2.2.21, el dominio P se instancia al dominio de los enteros $\mathbb{Z}$. Ahora añadamos a nuestro marco de trabajo el lenguaje $\mathcal{L}$ de expresiones que operan sobre los enteros (sobre P). En este ejemplo concreto, el lenguaje contendrá expresiones aritméticas con productos de enteros, adiciones, etc.

Estudiemos un poco más el caso del producto de enteros. El perfil de la función (u operación) producto lo podemos expresar de la siguiente forma: $* : \mathbb{Z}^2 \to \mathbb{Z}$.

Definamos el *dominio abstracto* ligeramente distinto al caso del ejemplo anterior. En este caso tendremos sólo dos elementos en el dominio, con los que distinguiremos los números positivos de los negativos:

$$D_\alpha = \{[-], [+]\}$$

Dicho de otra forma, los dos componentes de nuestro dominio abstracto representan dos clases de equivalencia para los enteros, las cuales distinguen los valores concretos de $\mathbb{Z}$ entre positivos y negativos..

Una vez definido el dominio abstracto y las funciones de abstracción y concreción de forma similar a como se ha hecho en el Ejemplo 2.2.21, tenemos que definir la versión abstracta de la función producto, y lo hacemos de la siguiente forma:

$$*_\alpha : D_\alpha^2 \to D_\alpha$$

$*_\alpha$	$[-]$	$[+]$
$[-]$	$[+]$	$[-]$
$[+]$	$[-]$	$[+]$

Nótese que el dominio del producto abstracto $*_\alpha$ ya no es el dominio de los enteros $\mathbb{Z}$, sino que será el dominio abstracto definido.

Leyendo la tabla anterior tenemos que el producto de dos elementos $[-]$ dará como resultado un elemento $[+]$. Lo mismo ocurrirá en caso de multiplicar dos elementos $[-]$ ya que obtendremos uno $[+]$. Sin embargo, si multiplicamos el elemento abstracto $[+]$ por $[-]$ o viceversa, obtendremos como resultado $[-]$.

A partir del nuevo dominio abstracto, podemos asegurar por ejemplo que x^2, es decir, cualquier entero multiplicado por sí mismo, dará como resultado un número positivo. ■

La idea que se persigue siempre es la de que una expresión concreta sea aproximada por las versiones abstractas de las mismas operaciones que aparecen en la expresión. Es decir, que si x e y son aproximados por x_α e y_α respectivamente, y $*_\alpha$ aproxima la operación $*$, entonces para aproximar la expresión compuesta $z = x * y$ no tendremos mas que calcular $z_\alpha = x_\alpha *_\alpha y_\alpha$.

Ejemplo 2.3.2 (Refinamiento) En caso de querer considerar el número 0 como un caso especial, podríamos refinar el dominio:

$$D'_\alpha = \{[-], [0], [+]\}$$

Ahora tenemos una nueva clase de equivalencia que representa el valor cero, a parte de las clases para los positivos y negativos.

Como el dominio abstracto ha cambiado y ahora tiene tres elementos en vez de dos, debemos redefinir la multiplicación abstracta:

$*_\alpha$	$[-]$	$[0]$	$[+]$
$[-]$	$[+]$	$[0]$	$[-]$
$[0]$	$[0]$	$[0]$	$[0]$
$[+]$	$[-]$	$[0]$	$[+]$

Esta nueva abstracción nos permite preguntar más cosas acerca del comportamiento de la multiplicación, por ejemplo podríamos saber que $z = y * (0 * x)$ siempre da como resultado 0. En función del marco definido (tanto los dominios como las operaciones) en cada caso, será posible analizar un tipo de propiedades u otro. Como ejemplo, el marco definido en el Ejemplo 2.3.1 no era capaz de analizar esta cuestión en concreto. ■

Para definir los dominios abstractos, aunque haya cierta libertad, tenemos que asegurarnos siempre de que sean abstracciones correctas (o seguras). Podemos definir distintas abstracciones para un mismo problema y compararlas según su precisión. En los dos ejemplos anteriores, podemos ver cómo la segunda aproximación es más precisa que la primera ya que nos permite hilar más fino en cuanto a las preguntas que es capaz de contestar.

Además, debemos pensar que podemos hacer abstracciones de un único elemento concreto, o de un conjunto de ellos. Es decir, que la abstracción de 3 será $[+]$, pero la de $\{3,4\}$ será también $[+]$. Pero ¿qué pasaría si intentáramos hallar la abstracción de $\{-2,2\}$?. Para estos casos es para los que usamos los valores $\top$ y $\bot$ ya que el elemento más preciso que representa a dicho conjunto en el dominio abstracto no es otro que $\top$ (observad el retículo del primer ejemplo).

Consideremos otro ejemplo pero esta vez vamos a trabajar abstrayendo el operador de suma de enteros.

Ejemplo 2.3.3 (Abstrayendo la suma de enteros) Como en los ejemplos anteriores tenemos que el dominio de P son los enteros $\mathbb{Z}$, pero ahora el dominio abstracto lo definimos como sigue:

$$D''_\alpha = \{[-], [0], [+], \top\}$$

El perfil de la operación de suma de enteros lo podemos escribir como $+ : \mathbb{Z}^2 \to \mathbb{Z}$. A continuación damos la especificación de la operación de suma abstracta:

$+_\alpha$	$[-]$	$[0]$	$[+]$	$\top$
$[-]$	$[-]$	$[-]$	$\top$	$\top$
$[0]$	$[-]$	$[0]$	$[+]$	$\top$
$[+]$	$\top$	$[+]$	$[+]$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

El elemento $\top$ representa todos los elementos de P, es decir, todos los enteros en $\mathbb{Z}$. Nótese que si no tuviéramos el elemento $\top$ en el dominio, tendríamos un problema ya que no podríamos encontrar una buena interpretación para la expresión $[+] +_\alpha [-]$ (la suma abstracta del elemento abstracto $[+]$ y el elemento abstracto $[-]$). Esto es debido a que en nuestro marco de trabajo no podemos saber de qué signo será el resultado de sumar un número positivo y un número negativo, ya que on se conocen sus valores absolutos.

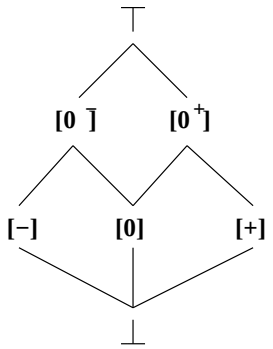
Con este nuevo marco abstracto ahora podríamos ver que $z = x^2 + y^2$ nunca da un resultado negativo. ■

Vamos a ver un último ejemplo de abstracción aún más refinada.

Ejemplo 2.3.4 Consideremos el dominio $D_\alpha = \{\bot, [-], [0^-], [0], [0^+], [+], \top\}$ donde la función de abstracción es la natural ($[-]$ representa los números negativos, $[0^-]$ representa a los negativos incluído el 0, etc.). La función de concreción se define de la siguiente forma:

$\gamma(\bot) = \emptyset$
$\gamma([-] = \{x \in \mathbb{Z} | x < 0\}$
$\gamma([0^-] = \{x \in \mathbb{Z} | x \leq 0\}$
$\gamma([0] = \{0\}$
$\gamma([0^+] = \{x \in \mathbb{Z} | x \geq 0\}$
$\gamma([+] = \{x \in \mathbb{Z} | x > 0\}$
$\gamma(\top) = \mathbb{Z}$

De forma gráfica, con las definiciones anteriores obtenemos como dominio abstracto el siguiente retículo dibujado como diagrama de Hasse, donde tenemos el elemento menor en la parte inferior del dibujo y el elemento mayor ($\top$) en la parte superior del dibujo.



Podemos ver de forma gráfica y muy intuitiva que en el retículo se cumple que $\sqcup\{[-],[0]\} = [0^-]$. Es decir, que el punto común por encima de todos los elementos del conjunto es $[0^-]$. También vemos que $\sqcup\{[+],[-]\} = \top$. Es decir, respectivamente son los elementos que representan a la conjunción de dos elementos de forma más precisa. ■

2.3.1. Abstracción por punto fijo

Vamos a pasar ahora a hablar de puntos fijos. El punto fijo de un operador $T : X \to X$ no es más que un elemento x del dominio X para el que $x = T(x)$, es decir, que aplicándole el operador, obtenemos como resultado el mismo elemento. A nosotros nos interesan porque existe una estrecha relación entre los puntos fijos, los operadores de cierre y los retículos completos. De hecho, todo retículo completo puede representarse mediante un operador de punto fijo.

Antes de empezar, vamos a recordar la notación introducida que vamos a usar aquí. Si $(X, \leq)$ es un poset, decimos que f es monótono siempre que $\forall x, y \in X, (x \leq y \Rightarrow f(x) \leq f(y))$. Si X es un retículo completo y f es monótono, entonces el conjunto formado por los puntos fijos de f también será un retículo completo. El menor elemento del retículo se denomina *menor punto fijo*, o en símbolos, $lfp(f)$. Para calcular las potencias de un operador monótono podemos acudir a las aplicaciones del operador:

$f \uparrow 0(x) = x$
$f \uparrow n(x) = f(f \uparrow (n-1)(x))$
$f \uparrow \omega(x) = \sqcup\{f \uparrow n(x) | n < \omega\}$

Escribiremos $f \uparrow \alpha(\bot)$ como $f \uparrow \alpha$ para abreviar. Gracias al resultado que nos dice que en un retículo existe siempre el menor límite superior, sabemos que siempre existirá algún ω en el que se llegue a un menor punto fijo: $f \uparrow \omega = lfp(f)$.

Ya sabemos que un subconjunto Y de un poset X, es una *cadena* si y sólo si $\forall y, y' \in Y, y \leq y' \vee y' \leq y$. Ahora añadiremos el hecho de que un retículo completo tiene *cadenas ascendentes* si y sólo si, para toda cadena no vacía $Y \subseteq X, \sqcup Y \in Y$. Con todo esto, podemos decir que si tenemos un retículo con cadenas ascendentes finitas, entonces la secuencia hasta llegar al menor punto fijo es siempre finita.

Los puntos fijos son interesantes porque podemos representar la semántica *collecting* de un programa mediante una caracterización por punto fijo. La semántica *collecting* de un programa no es mas que todos los posibles estados en los que puede encontrarse un determinado programa en un momento específico. Cada estado dependerá de la entrada del programa. Gracias a este hecho, podremos demostrar que todos los elementos del menor punto fijo del dominio concreto satisfacen cierta propiedad simplemente demostrando dicha

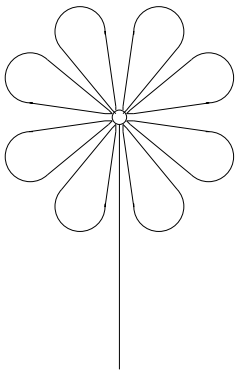
propiedad para todos los elementos de la concreción del menor punto fijo en el dominio abstracto (es decir, trabajando sobre el dominio abstracto directamente).

2.4. Abstracción de propiedades

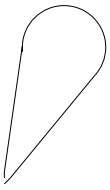
En esta sección vamos a ver de forma más o menos intuitiva en qué consiste el análisis de propiedades abstractas de programas.

Imaginemos que tenemos un pequeño lenguaje gráfico compuesto de objetos y de operaciones sobre objetos. Cada objeto estará computesto por dos valores, un origen (un punto de referencia) y un conjunto de píxeles negros (sobre una pizarra blanca).

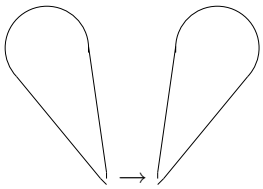
Por ejemplo podemos tener como objeto de nuestro dominio concreto una flor como la que vemos en la siguiente figura:



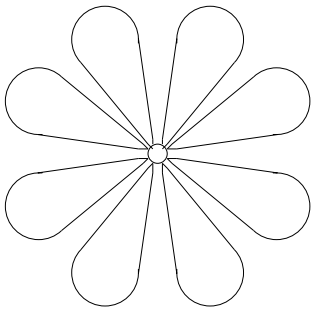
Sin embargo, una flor puede verse como un objeto compuesto, formado en realidad por otros objetos más sencillos como pétalos, un tayo, etc. Podemos generar objetos complejos a partir de otros más sencillos usando operaciones sobre estos últimos. Cojamos como punto de partida un pétalo al que consideraremos una constante de nuestro lenguaje gráfico.

pétalo = 

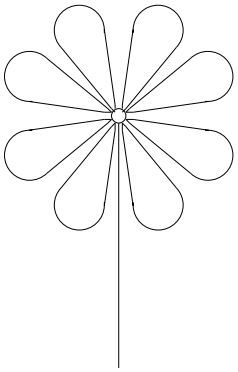
A parte de las constantes tendremos por tanto operaciones para transformarlas de alguna forma, componerlas uniendo distintas constantes, etc. Por ejemplo, para rotar objetos podemos tener la operación $r[a](o)$ que rota un objeto o un número determinado a de grados con respecto al punto de referencia de o. Gráficamente tendríamos que al aplicar una rotación al pétalo de la izquierda pasaríamos a tener el pétalo de la derecha en el siguiente esquema:



También podemos tener una operación que una (o componga) objetos. Es más, podría unir objetos que han sido previamente rotados, pudiéndose construir de esta forma la corola de una flor. Usaremos la notación $o_1 \cup o_2$ para unir dos objetos (o_1 y o_2). Vamos a ver de forma gráfica el resultado de unir ocho pétalos con sus respectivas y apropiadas rotaciones para formar una corola:



Por último, podríamos tener un operador más (*stem*), que tomara como entrada un objeto corola y le añadiera un tallo, de forma que definiría una flor completa:



stem(corola) =

Una forma alternativa de obtener la corola de la flor sería usando un operador de punto fijo. Este operador puede verse como una generalización de las uniones y rotaciones que deben aplicarse a nuestro ejemplo. Si elegimos bien el grado de rotación, hayaremos pronto un punto fijo tal y como se muestra en la Figura 2.8.

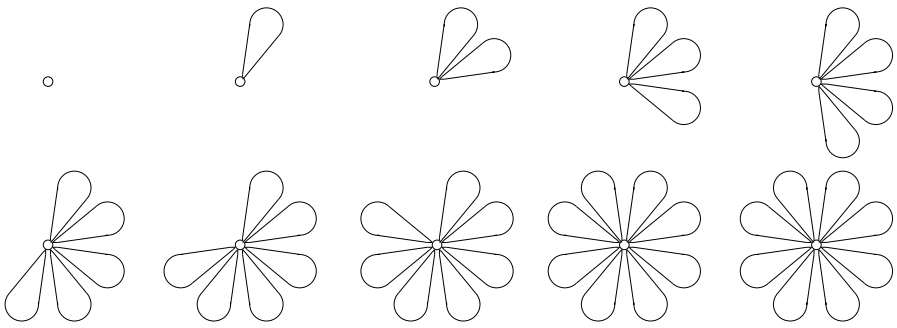


Figura 2.8: Secuencia de aplicaciones del operador de punto fijo del ejemplo

Si definiéramos un operador de punto fijo para las flores completas a las que se aplicara una cierta rotación en cada iteración, podríamos obtener un nuevo objeto *ramo*. También podemos obtener un ramillete uniendo tres flores con una ligera rotación.

2.4.1. Abstracción de objetos

En esta sección vamos a ver cómo abstraer un determinado objeto del lenguaje gráfico definido en el apartado anterior. Recordemos que la *sobre-aproximación* de un objeto es otro objeto que tiene el mismo origen y que tiene más píxeles negros que el objeto original (y además incluye a todos los del original). Esta inclusión es necesaria para tener una abstracción correcta. En el ejemplo, una sobre-aproximación podría ser desde el definir un trazo del objeto más gordo, hasta el trazar el perímetro del objeto y considerar todo su interior (incluído el borde), o trazar una serie de objetos más sencillos que cubran toda la superficie del objeto. Cada una de estas opciones es correcta aunque todas tienen una precisión diferente.

En la Figura 2.9 se muestran varias posibilidades de sobre-aproximaciones correctas del objeto flor. Además, puede verse que la alternativa que aparece más a la derecha es una sobre-aproximación de la del medio, que a su vez es una sobre-aproximación de la que aparece a la izquierda de la figura. Esta última se corresponde con la abstracción más precisa de las tres mostradas.

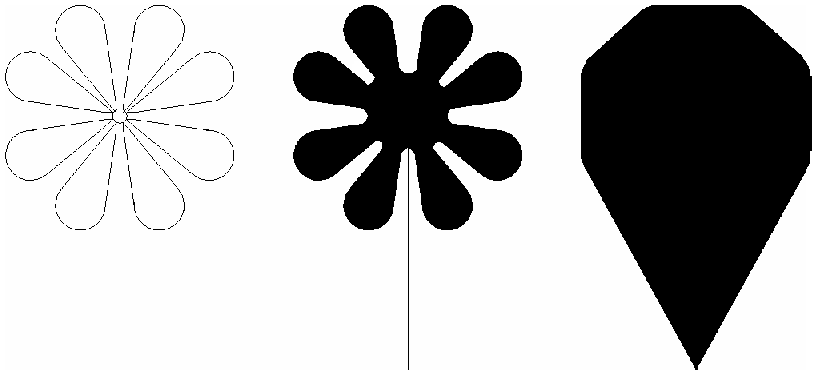


Figura 2.9: Tres posibles abstracciones del objeto flor

Aunque en el ejemplo de la Figura 2.9 no ocurre, puede darse el caso de que distintas abstracciones de un mismo objetos no sean comparables entre sí.

Como ya sabemos, un dominio abstracto es un conjunto de objetos que representan de alguna forma a los objetos concretos, más algunas operaciones abstractas sobre dichos representantes. Estas operaciones son aproximaciones de las correspondientes operaciones concretas. En el ejemplo del lenguaje gráfico, se tendrán que definir las respectivas abstracciones de las operaciónes de rotación y de unión de objetos.

No debemos olvidar nunca la función de concreción. En el ejemplo podemos verla de forma intuitiva de la siguiente forma: dado el perímetro de la flor (es decir, la abstracción), la concreción consistirá en rellenar todo ese perímetro, coloreando así de negro todos los píxeles interiores. Ya sabemos que una función de concreción γ mapea un objeto abstracto $\overline{o}$ a un objeto concreto $\gamma(\overline{o})$, es decir, obtiene la semántica concreta del objeto abstracto.

Un último apunte que daremos es que podemos usar la iteración por punto fijo para construir la corola abstracta a partir de ir abstrayendo las iteraciones concretas, pero es bueno definir operadores de punto fijo abstractos para no tener que manejar el dominio concreto, ya que esto último puede resultar demasiado costoso.

2.4.2. Abstracción de propiedades: dos alternativas

Hasta este momento, todos los ejemplos que hemos visto hablaban de abstracciones correctas para el análisis de propiedades de seguridad. En esta última sección del capítulo vamos a introducir qué otro tipo de abstracciones pueden definirse para analizar otro tipo de propiedades.

Cuando se quiere analizar o probar programas, en cierta forma podemos decir que queremos considerar o analizar objetos que representan cierta parte del estado de computación del sistema. Por ejemplo podemos analizar

- valores de variables: enteros, booleanos,... que representaremos por $\mathcal{V}$
- nombres de variables que representaremos como $\mathbb{X}$
- entornos (es decir, situaciones concretas) representados como $\mathbb{X} \mapsto \mathcal{V}$
- pilas de valores, es decir, asignaciones de valores a variables en el contexto de lenguajes de programación estructurados: $\bigcup_{n\geq 0}([1,n] \mapsto (\mathbb{X} \not\mapsto \mathcal{V}))$
- etc.

En un lenguaje de programación estructurado tradicional podemos encontrar otros objetos como *heaps* que representarán la memoria dinámica, tendremos puntos de control que podrán representarse mediante nombres de procedimientos, etiquetas, etc. y tendremos también estados del sistema que consistirán en un punto de control más el estado de la memoria del sistema. Los conceptos interesantes, es decir, los que se pretenden analizar, son los llamados *observables*. También podemos considerar otros observables centrados en aspectos dinámicos del lenguaje de programación como son los prefijos finitos de las trazas, las trazas maximales finitas (o infinitas) en el caso de programas deterministas, e incluso los conjuntos de trazas tanto finitas como infinitas en el caso de los programas no deterministas.

Una propiedad puede verse como un *conjunto de objetos*; en particular el conjunto de objetos que satisfacen dicha propiedad. Por ejemplo, podríamos tener las siguientes propiedades representadas por los respectivos conjuntos:

- números impares: $\{1, 3, 5, \ldots, 2n+1, \ldots\}$
- números pares: $\{2 * z | z \in \mathbb{Z}\}$
- los valores de las variables enteras en un programa: $\{z \in \mathbb{Z} | \text{minint} \leq z \leq \text{maxint}\}$
- valores de las variables enteras que puede que no hayamos inicializado: $\{z \in \mathbb{Z} | \text{minint} \leq z \leq \text{maxint}\} \cup \{\Omega_n | m \in \mathcal{M}\}$ donde $\mathcal{M}$ es el conjunto de mensajes de error
- igualdad de dos variables `X` e `Y`: $\{\rho \in \mathbb{X} \not\mapsto \mathcal{V} | \mathtt{X}, \mathtt{Y} \in \text{dom}(\rho) \wedge \rho(\mathtt{X}) = \rho(\mathtt{Y})\}$

- una propiedad invariante de un programa con estados en el conjunto Σ: $\mathcal{I} \in \wp(\Sigma)$
- una propiedad relacionada con una traza: $T \in \wp(\Sigma^{\vec{\infty}})$
- una propiedad de la semántica de trazas: $P \in \wp(\wp(\Sigma^{\vec{\infty}}))$

De hecho, el conjunto de propiedades concretas de un programa representadas como conjuntos de objetos forma un retículo completo. Sea el conjunto de propiedades $\wp(\Sigma)$ de los objetos en Σ, podemos definir el retículo completo

$$\langle \wp(\Sigma), \subseteq, \emptyset, \cup, \cap, \neg \rangle$$

donde una propiedad $P \in \wp(\Sigma)$ es el conjunto de objetos que satisfacen la propiedad P y $\subseteq$ es la implicación lógica ya que $P \subseteq Q$ significa que todo objeto con la propiedad P tiene la propiedad Q ($o \in P \Rightarrow o \in Q$). Además, podemos decir que el conjunto vacío representa la propiedad *false* mientras que Σ representa *true*. $\cup$ es la disyunción (es decir, representa los objetos que tienen o bien la propiedad P, o bien la Q - o ambas - en la expresión $P \cup Q$). $\cap$ representa la conjunción de propiedades entendida de forma análoga a $\cup$ mientras que $\neg$ es la negación, es decir, que los objetos en $\Sigma \backslash P$ son los objetos que no tienen la propiedad P.

Una abstracción sustituye *algo concreto* por una descripción esquemática que tiene en cuenta algunas de las propiedades (pero no todas) del concreto. O dicho de otra forma, el modelo abstracto que se inferirá describirá alguna de las propiedades del objeto concreto, pero no todas ellas. Por lo tanto, una abstracción de propiedades $\wp(\Sigma)$ de Σ es básicamente un subconjunto $A \subseteq \wp(\Sigma)$ de forma que:

- las propiedades presentes en A son las propiedades concretas que pertenecen al dominio abstracto, es decir, las consideradas por la función de abstracción sin pérdida de información
- las propiedades en $\wp(\Sigma) \backslash A$ son las propiedades que no describe exactamente la abstracción, es decir las no consideradas totalmente por la función de abstracción y a las que hay que hacer referencia por medio de aproximaciones expresadas en A.

En resumen, una *abstracción* de una computación sólo puede usar algunas propiedades $A \subseteq \wp(\Sigma)$ de los objetos en Σ. Además, las propiedades $P \in A$ que pueden usarse se denominan propiedades *abstractas* mientras que a las propiedades $P \in \wp(\Sigma)$ las llamamos propiedades *concretas*.

Asumamos que las *computaciones abstractas* trabajan sobre *aproximaciones correctas* o seguras. Entendemos que una aproximación es correcta porque

- Las propiedades concretas que son también abstractas, pueden usarse para razonar en la computación abstracta directamente, sin necesidad de modificarse y sin pérdida alguna de información
- Las propiedades concretas que no son abstractas $P \in \wp(\Sigma) \backslash A$ no pueden usarse en el razonamiento sobre la computación abstracta directamente, sino que deben ser *aproximadas* por alguna otra propiedad abstracta $\overline{P} \in A$ ($P \neq \overline{P}$) que constituirá una aproximación de la original.

Cuando aproximamos una propiedad concreta $P \in \wp(\Sigma)$ mediante una propiedad abstracta $\overline{P} \in A$ diferente a P, debemos establecer una relación entre ambas propiedades de forma que asegure la corrección del razonamiento en la computación abstracta con respecto a la concreta. Decimos entonces que $\overline{P} \in A$ *es una aproximación/abstracción de* $P \in \wp(\Sigma)$.

Existen dos formas básicas de aproximar una propiedad:

- Aproximación *por arriba* (o sobre-aproximación): $P \subseteq \overline{P}$
- Aproximación *por abajo* (o sub-aproximación): $P \supseteq \overline{P}$

En este texto, hasta ahora los ejemplos definían siempre aproximaciones por arriba (sobre-aproximaciones). Las dos nociones o alternativas para definir las abstracciones son duales. En la práctica, las sub-aproximaciones suelen ser de menor utilidad que las sobre-aproximaciones, y de hecho es más difícil encontrar una sub-aproximación que sea de utilidad real que una sobre-aproximación buena.

Veamos la dualidad de las dos técnicas con un ejemplo ilustrativo. Imaginemos que tenemos un conjunto de puntos bidimensionales que satisfacen una cierta propiedad P. En la Figura 2.10 representamos como puntos negros los puntos que satisfacen la propiedad. Es decir, la propiedad P será el conjunto de puntos negros.

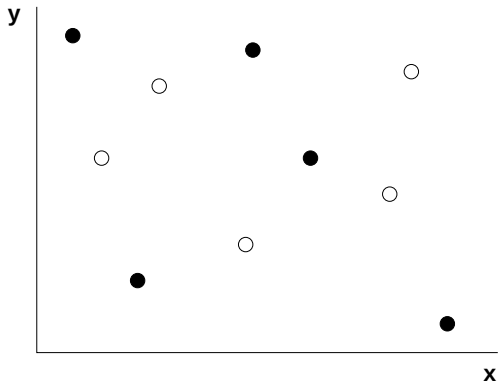
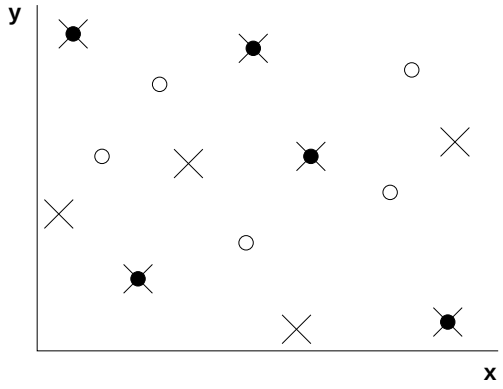


Figura 2.10: Representación de un conjunto de puntos

En primer lugar razonaremos sobre el caso sobre-aproximado, que es el que resultará más familiar al lector.

1. Caso sobre-aproximado.

   En este caso la propiedad abstracta $\overline{P}$ contiene al menos todos los puntos que originalmente satisfacían P, es decir, todos los puntos negros. Por lo tanto, todos los elementos de P están contenidos también en $\overline{P}$, pero pueden existir puntos de la propiedad abstracta $\overline{P}$ que en realidad no estaban en la concreta (se corresponden con las aspas sin puntos negros en la figura de abajo).



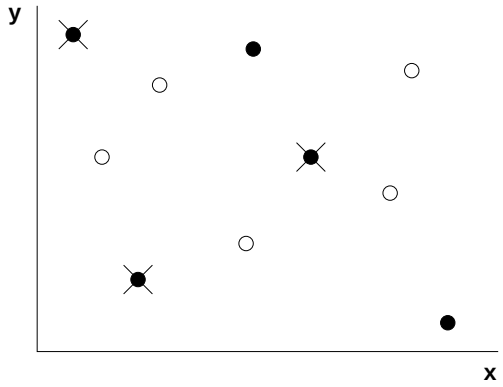
En la figura se han representado con aspas los puntos que representan la propiedad abstracta. Es decir, los puntos negros con un aspa superpuesta indican que dicho punto satisface tanto la propiedad P como su aproximación $\overline{P}$. Se puede observar que existen aspas sin punto negro, lo que como ya hemos mencionado se orresponde a puntos no considerados en el mundo concreto.

Esto quiere decir que ante la pregunta: *¿El punto (x,y) pertenece a P?*, si intentamos contestarla usando únicamente $\overline{P}$ (es decir, fijándonos únicamente en las aspas), tendremos que

- si $(x, y) \in \overline{P}$, entonces no sabemos nada seguro ya que puede que ese punto sea precisamente uno de los puntos que no están en P.
- si $(x, y) \notin \overline{P}$, entonces sabemos que la respuesta a la pregunta es **negativa** ya que todos los puntos de P están contenidos en la propiedad abstracta y por tanto han sido considerados.

2. Caso sub-aproximado.

   En este caso, la propiedad abstracta $\overline{P}$ es un subconjunto de la propiedad P. Es decir, que todos los elementos en la aproximada están también en la concreta pero existen algunos de la concreta que no están en la aproximada.



En la figura se ha vuelto a representar con aspas los puntos que cumplen la propiedad abstracta. Se puede observar que todas las aspas están superpuestas a puntos negros, pero hay puntos negros sin aspas.

Esto quiere decir que ante la pregunta: *¿El punto (x,y) pertenece a P?*, si intentamos contestarla usando únicamente $\overline{P}$ (las aspas), tendremos que

- si $(x, y) \in \overline{P}$, entonces sabemos que la respuesta es **afirmativa**, es decir, que $(x, y) \in P$
- si $(x, y) \notin \overline{P}$, entonces no podemos asegurar nada ya que puede que el punto en concreto sea uno de los que están en P pero no en su aproximación

Así pues, se puede ver que los dos tipos de abstracciones (o aproximaciones) posibles, tienen una utilidad diferente (y complementaria) ya que con ellas podemos responder a preguntas de naturaleza distinta. En el siguiente capítulo veremos cómo se puede aprovechar esta circunstancia para analizar un sistema usando una técnica concreta como es el *model checking*.

3

Model Checking Abstracto

3.1. Introducción

Ya sabemos que la interpretación abstracta es una técnica introducida por Cousot y Cousot en 1977 como forma unificada para el análisis de programas. Hemos visto que la interpretación abstracta describe computaciones en un universo diferente (abstracto) de objetos de forma que los resultados en el universo abstracto nos dan información sobre las ejecuciones en el universo original (real).

Todo elemento real debe estar relacionado (representado) por uno abstracto, y además todo elemento abstracto debe ser representante de alguno concreto.

En este capítulo estudiamos cómo hacer la técnica de *model checking* efectiva cuando el espacio de búsqueda es demasiado grande usando la interpretación abstracta. En el *model checking* abstracto, la idea fundamental que se persigue es conseguir probar una propiedad en el universo abstracto, garantizando que dicha propiedad se cumple también en el universo concreto. Si no es posible probar una propiedad en el abstracto, entonces no se asegura nada. Este tipo de resultados (preservación cuando la respuesta es positiva) son los que obtenemos cuando tenemos un marco de trabajo que cumple con la condición de *preservación débil*. Podemos llegar a tener *preservación fuerte* pero no suele ser frecuente. Con preservación fuerte tenemos que cuando una propiedad es falsa en el marco abstracto, se garantiza que es falsa también en el concreto.

Existen distintos tipos de sistemas muy diversos, pero podemos establecer dos grandes clases de sistemas: sitemas funcionales y sistemas reactivos. Los sistemas funcionales tienen un punto final concreto, donde podemos verificar una posible postcondición. Son los sistemas que toman una entrada y genera un resultado específico como salida. Para Razonar sobre estos sistemas tenemos lógicas como la de Hoare, de precondiciones y postcondiciones, etc. Además, la técnica de *testing*, aunque bajo determinados contextos se puede aplicar a sistemas reactivos, es adecuada para verificar sistemas funcionales, ya que analizan el resultado final, la relación de entrada/salida de los programas.

En los sitemas reactivos no hay un punto final concreto. Los clásicos sistemas reactivos son los sistemas empotrados (el *software* que controla las máquinas de café, los cajeros automáticos, etc.) pero también los sistemas operativos, los protocolos de comunicación, y en general todo sistemas que durante su ejecución interactúe con el usuario o algún otro *software*. Normalmente estos sistemas se especifican como sistemas concurrentes, por lo que su análisis suele ser complejo, sobretodo si se realiza de forma manual. Si a todo esto añadimos el hecho de que podemos tener sistemas no deterministas, vemos el motivo por el que las técnicas de verificación automática para sistemas reactivos son tan importantes

en la actualidad: hoy en día el *software* que se produce es cada vez más grande y complejo.

En un sistema reactivo, la ejecución podría continuar indefinidamente de forma que nunca habrá un sitio donde poder comprobar o verificar una postcondición. Por esto son necesarias otro tipo de lógicas, capaces de razonar y analizar las trazas de ejecución del sistema y no el resultado final. Además, las propiedades que normalmente queremos verificar o analizar de los sistemas reactivos suelen ser propiedades dinámicas, y lo queremos hacer de una forma estática (es decir, sin necesidad de ejecutar el programa). Esto hace que el problema se convierta en un problema NP-completo ya que aparece el ya famoso problema de la explosión del espacio de estados y de la falta de cobertura. Las lógicas que tenemos que usar en estos casos se han estudiado ya en otras asignaturas, pero en este texto daremos las nociones básicas de las lógicas que usaremos para que la documentación sea completa.

En los sistemas reactivos en general podemos estudiar cuatro tipo de propiedades. En primer lugar, podemos estudiar propiedades de seguridad (*safety*) o de viveza (*liveness*), pero también podemos dividir el tipo de propiedades a analizar entre *universales* o *existenciales*. Obviamente, normalmente tendremos combinaciones de estos dos criterios, podremos por tanto poder estudiar propiedades de seguridad universales, o de viveza universales, de seguridad existenciales o de seguridad universales. Las técnicas usadas para verificar propiedades de seguridad universales son distintas de las usadas para analizar propiedades existenciales o de viveza ya que en el caso de las de seguridad universales, suele resultar más sencillo buscar contraejemplos que analizar la satisfacción de la propiedad.

Objetivos del capítulo

El objetivo de este tema es, principalmente, ver la aplicación de la técnica de interpretación abstracta al *model checking*. Se considerará todos los aspectos de la metodología (abstracción de dominios, de relaciones, etc.) y se estudia el problema del refinamiento de la asbtracción, o la optimalidad de la abstracción.

3.2. Model Checking

La técnica de *model checking* [CES86, CGP99] consiste básicamente en la comprobación de que cualquier modelo de un determinado sistema satisface una propiedad dada. Es una técnica formal porque tiene una base fundamental matemática, y por lo tanto da una respuesta *segura*. Sin embargo, debido a que está basada en la búsqueda exhaustiva en el espacio de estados del sistema, a priori puede aplicarse únicamente a sistemas con un espacio de búsqueda finito (e incluso no demasiado grande). Ya se ha visto que existen muchas técnicas que permiten mejorar la eficiencia, y por lo tanto la aplicabilidad, de la técnica. Por ejemplo, el *model checking* simbólico [CMCHG96, BCM$^+$92] está basado en la representación mediante funciones booleanas del modelo y la propiedad. También existen otro tipo de optimizaciones como son las técnicas *on-the-fly* [Cou99], basadas en lenguajes regulares [BJNT00], en métodos composicionales, en evaluación parcial, en simulaciones y órdenes parciales, y como veremos en este tema, en interpretación abstracta [Dam96, DGG97, CGL94].

Recordaremos que el problema del *model checking* se formula de la siguiente forma:

dado el modelo del sistema $\mathcal{M}$ y una propiedad que queremos verificar ϕ, se establece si

$$\mathcal{M} \models \phi$$

El algoritmo puede dar una respuesta positiva, en cuyo caso sabemos que el sistema satisface la propiedad, o bien una respuesta negativa, y en este caso se dará también un contraejemplo: la traza en la que la propiedad ha fallado. Recordemos que el enunciado nos dice que comprobemos si para todo estado inicial del modelo, la propiedad se satisface:

$$\forall s \in \mathit{init}(\mathcal{M}) \;\; \mathcal{M}, s \models \phi$$

3.2.1. El modelo

La estructura clásica usada para la representación del sistema (el modelo), suele ser algún tipo de *estructura de Kripke*:

Definición 3.2.1 (Estructura de *Kripke*) *Una estructura de Kripke es una tupla* $\langle S, I, R, L\rangle$ *donde*

- S *es un conjunto de estados (todos los estados del sistema)*
- $I \subseteq S$ *es el conjunto de estados iniciales del sistema*
- $R \subseteq S \times S$ *es una relación (total) binaria definida entre estados, y*
- $L : \wp(\Sigma) \rightarrow S$ *es una función de etiquetado que define, para cada estado, qué expresiones son ciertas. Dicho de otra forma, para cada conjunto de expresiones del lenguaje, define en qué estados se satisfacen.*

El hecho de exigir que la relación de transición sea total se debe a que estamos interesados en las trazas infinitas, y una forma de garantizarlo es pidiendo que todo estado tenga un sucesor. Sin embargo existen otras alternativas con las que no es necesario tener una relación total, ya que podemos eliminar las trazas finitas a nivel de semántica de la lógica (capturando únicamente las trazas infinitas).

La estructura de Kripke aparece en su origen para dar una interpretación a la *lógica modal* clásica, en la que cada nodo representaba un mundo posible, y las transiciones se correspondían con el paso de un mundo a otro en función del conocimiento adquirido hasta ese momento.

Estas estructuras pueden representarse de forma gráfica mediante un grafo dirigido. En nuestro contexto (el de la ingeniería del *software*), cada nodo del grafo representará un estado del sistema, mientras que la relación R de transición entre estados simboliza los pasos de ejecución y se representa mediante arcos entre los nodos. La función de etiquetado se representará definiendo en cada nodo el conjunto de expresiones o proposiciones que se satisfacen.

Ejemplo 3.2.2 En la Figura 3.1 vamos a representar un ejemplo típico de estructura de Kripke que especifica el comportamiento de un microondas. El ejemplo, aunque parecido, no se corresponde exactamente con el presentado en la literatura clásica [MP95].

■

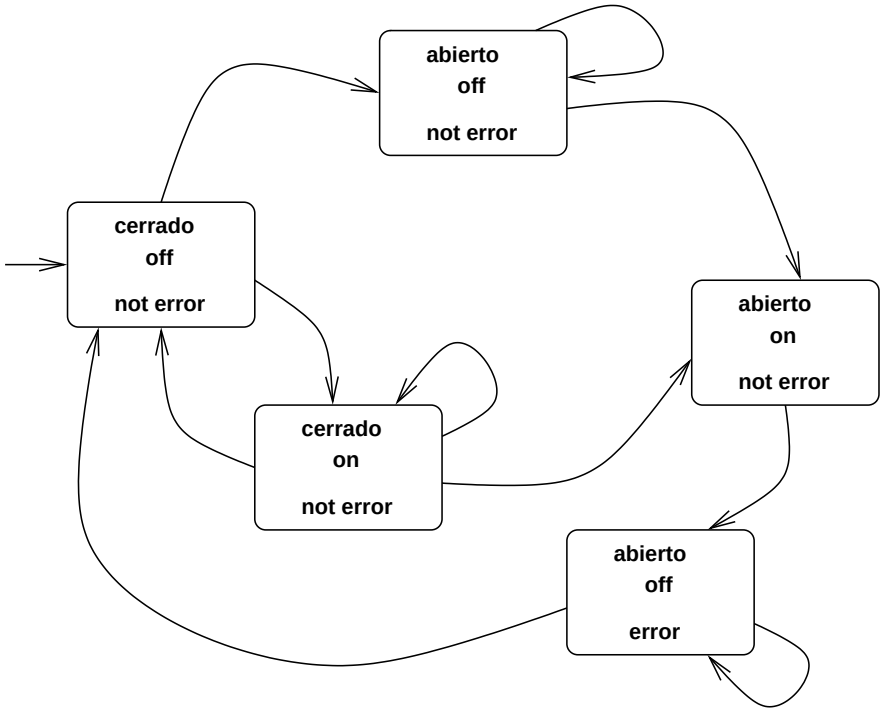


Figura 3.1: Modelo simplificado de un microondas

Un modelo como el del ejemplo puede ser calculado a partir de un programa o especificación de alto nivel de formas distintas. En primer lugar, la técnica más frecuente es la de generarlo a mano. El problema de esta alternativa es que podríamos introducir errores adicionales (que no se corresponden con el programa real) sin darnos cuenta. Desde luego, es preferible hacerlo de forma automática aunque no todas las herramientas contemplan esta posibilidad y muchas veces nos vemos obligados a fiarnos de nuestra destreza a la hora de definirlo. Cuando se detecta un error en el modelo, se debe ser capaz de identificarlo en el propio código del programa.

3.2.2. La propiedad

La forma más usual de especificar la propiedad que queremos verificar suele ser mediante algún tipo de lógica temporal. En esta sección vamos a definir y describir las tres lógicas temporales que encontramos más frecuentemente en la literatura relacionada con el *model checking*.

Las lógicas temporales son una clase particular de la lógica modal clásica. En particular, proporcionan los mecanismos formales necesarios para describir de forma cualitativa aspectos dinámicos de los sistemas. De esta forma permiten razonar sobre la evolución del valor de verdad de las afirmaciones a lo largo del tiempo. Las modalidades temporales clásicas son *eventualmente* y *siempre*.

Las lógicas temporales fueron rescatadas de los libros en el año 83 y siguientes debido a la necesidad de razonar sobre trazas y no sobre resultados finales de los programas. A partir de entonces, dejaron de ser un tipo de lógica olvidada y empezaron a aplicarse a la

verificación de sistemas reactivos.

Clasificación

Podemos clasificar las lógicas temporales (TL) para analizar sistemas concurrentes según distintos criterios: lógicas proposicionales frente a de primer orden, globales frente a composicionales, ramificadas frente a lineales, puntuales frente a de intervalos, discretas frente a continuas o lógicas del pasado frente a lógicas del futuro.

Las lógicas temporales proposicionales toman la lógica proposicional clásica y le añaden una serie de operadores temporales. Así pues, las lógicas temporales proposicionales se basan en un conjunto de proposiciones atómicas que expresan hechos atómicos sobre los estados del sistema. Por otro lado, las lógicas temporales *de primer orden* se inspiran en la lógica de primer orden tradicional.

Las lógicas globales se interpretan en un único universo que se corresponde, en nuestro contexto, con un único programa. Por otro lado, la sintaxis de las lógicas composicionales nos permite expresar propiedades sobre distintos programas (o fragmentos del programa) en una misma fórmula.

La naturaleza del tiempo nos divide las lógicas en dos tipos: las ramificadas y las lineales, dependiendo de si estamos considerando un tiempo en el que puede haber distintos futuros (natural en sistemas no deterministas), o un tiempo en el que sólo hay un posible futuro. En el primer caso tenemos un tiempo en forma de árbol y los operadores de la lógica nos permiten cuantificar las ramas del árbol.

La mayoría de los formalismos lógicos están basados en el análisis del valor de verdad de una fórmula en un instante determinado. Son las lógicas de punto. Existen otras lógicas que tienen operadores que permiten ser evaluados en un intervalo de tiempo, lo que a veces puede simplificar la especificación de algunas propiedades.

Otra característica importante del tiempo nos proporciona la clasficación en cuanto a continuo o discreto. En la mayoría de sistemas, el tiempo se asume que es discreto: el estado actual es el presente y el siguiente momento se corresponde con el siguiente estado del programa. También existen las lógicas *densas* en las que las fórmulas se interpretan según un tiempo continuo como pueden ser los reales o los racionales. Este segundo tipo de lógica es importante para expresar propiedades de sistemas híbridos, donde ciertas variables pueden tener una naturaleza continua (por ejemplo expresando valores de variables relacionadas con temperaturas, volúmenes, etc.).

Por último, una lógica temporal puede tener operadores de tiempo que hacen referencia al pasado, que hacen referencia al futuro, o de ambos tipos. Hoy en día el uso de operadores del pasado está limitado al hecho de que puedan simplificar la especificación de ciertas propiedades. En realidad, la introducción de operadores pasados añade expresividad cuando manejamos una noción de equivalencia global, si por el contrario usamos una noción de equivalencia dependiendo de un instante inicial, entonces las dos versiones de la lógica son equivalentes.

3.2.3. La lógica temporal lineal (LTL)

La estructura en la que se basa la *lógica temporal lineal* (LTL) es un conjunto totalmente ordenado. Normalmente se asume el orden $(\mathbb{N}, <)$. Esto quiere decir que hace uso de una noción de tiempo discreto, con un punto inicial que no tiene predecesores (un mínimo

en el orden) e infinito. Esta es una noción apropiada para razonar sobre programas en general ya que la ejecución de un programa es discreta (paso de estado a estado) y siempre comienza en un estado inicial. Los valores de verdad de las fórmulas no se interpretan sobre un estado único, sino sobre secuencias de estados (computaciones o trazas).

La lógica temporal lineal proposicional

Los operadores básicos de la lógica temporal lineal proposicional (PLTL) son

- $\Diamond\phi$ leído *en algún instante en el futuro ϕ es cierta*. En algunos textos encontramos este mismo operador escrito como $F\phi$;
- $\Box\phi$ leído *en todos los instantes futuros ϕ es cierta*. Algunas veces aparece escrito como $G\phi$;
- $\bigcirc\phi$ leído *en el siguiente instante de tiempo ϕ es cierta*. Lo podemos encontrar escrito como $X\phi$; y
- $\phi\mathcal{U}\psi$ leído como *es cierta ϕ hasta el instante futuro donde ψ es cierta*.

Dado un conjunto de proposiciones atómicas Ω, la sintaxis de la lógica PLTL consiste en el menor conjunto de fórmulas generadas siguiendo las siguientes reglas:

1. toda proposición atómica Ω es una fómula PLTL
2. dadas dos fórmulas PLTL ϕ y ψ, las fórmulas $\phi\wedge\psi$ y $\neg\phi$ también son fórmulas PLTL
3. dadas dos fórmulas PLTL ϕ y ψ, las fórmulas $\phi\mathcal{U}\psi$ y $\bigcirc\phi$ también son fórmulas PLTL

Se pueden definir los demás operadores en función de los anteriores como $\phi\vee\psi$ que abrevia $\neg(\neg\phi\wedge\neg\psi)$, $\phi\rightarrow\psi$ abrevia $\neg\phi\vee\psi$, o $\phi\equiv\psi$ abrevia $(\phi\rightarrow\psi)\wedge(\psi\rightarrow\phi)$. Además, *true* abrevia $\neg\phi\vee\phi$ mientras que *false* abrevia $\neg true$. En cuanto a los operadores temporales, $\Diamond\phi$ abrevia $true\,\mathcal{U}\phi$ y $\Box\phi$ abrevia $\neg\Diamond\neg\phi$.

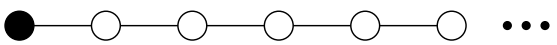
La semántica de las fórmulas PLTL la definiremos basándonos en secuencias de estados en un modelo $M=(S,I,R,L)$ donde, como siempre S es un conjunto de estados, R es una relación entre estados y L es una función que etiqueta los estados con las fórmulas que se satisfacen en cada uno de ellos. Definimos una secuencia temporal de estados como $s=s_0,s_1,\ldots,s_n,\ldots$ donde para cada s_i y s_{i+1} se cumple que existe un par $(s_i,s_{i+1})\in R$. Diremos también que s^i es el sufijo $s_i,s_{i+1},s_{i+2},\ldots$ de s.

Una vez definida la nomenclatura que usaremos, definimos la relación de satisfacción $\models$ con respecto a la secuencia s. Esta definición se puede extender de forma obvia a modelos considerando las secuencias cuyo primer estado es un estado inicial del mismo. Abusaremos de notación y escribiremos $\models$ cuando hablemos tanto de satisfacción de secuencias como de satisfacción de modelos.

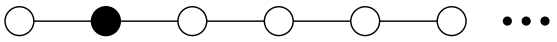
(1)	$s\models\phi$	sii $\forall\phi$ proposición atómica de la lógica, $\phi\in L(s_0)$
(2)	$s\models\phi\wedge\psi$	sii $s\models\phi$ y además $s\models\psi$
(3)	$s\models\neg\phi$	sii $s\not\models\phi$
(4)	$s\models(\phi\mathcal{U}\psi)$	sii $\exists j$ tal que $s^j\models\psi$ y además $\forall k<j$ se cumple que $s^k\models\phi$
(5)	$s\models\bigcirc\phi$	sii $s^1\models\phi$

Ejemplo 3.2.3 En la siguiente figura, podemos ver representado el comportamiento de los distintos operadores y modalidades. Representaremos como una secuencia de puntos la traza s, y como puntos negros el momento en el que se satisface p.

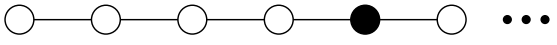
- $s \models p$



- $s \models \bigcirc p$



- $s \models (true\, \mathcal{U}\, p)$



■

Por último daremos la caracterización de punto fijo de los operadores temporales:

$$\begin{array}{lcl} \models \Diamond\phi & \equiv & \phi \vee \bigcirc\Diamond\phi \\ \models \Box\phi & \equiv & \phi \wedge \bigcirc\Box\phi \\ \models \phi\mathcal{U}\psi & \equiv & \psi \vee (\phi \wedge \bigcirc(\phi\mathcal{U}\psi)) \end{array}$$

3.2.4. La lógica temporal ramificada (CTL* y CTL)

La estructura de tiempo que hay debajo de la lógica temporal ramificada, es una estructura de árbol infinito. De hecho asumiremos que cada camino en el árbol es isomorfo a $\mathbb{N}$ y, además, cada nodo del árbol puede tener un número infinito de sucesores inmediatos. Además todos los nodos del árbol tendrán como mínimo un sucesor.

Esta lógica, al igual que la lógica LTL, se interpreta sobre una estructura de Kripke $M = (S, I, R, L)$ donde de nuevo S es un conjunto de estados, R una relación binaria entre estados y L una función que etiqueta los estados con las fórmulas que se cumplen en cada uno de ellos.

La lógica temporal ramificada proposicional

Existen dos lógicas principales de este tipo: la *Computational Tree Logic* (CTL) y la versión enriquecida CTL*. Estas lógicas, además de los operadores temporales tienen dos operadores que cuantifican los caminos del árbol. En particular, el operador A cuantifica universalmente los caminos mientras que E los cuantifica existencialmente.

A continuación daremos la sintaxis de la lógica CTL*. En esta lógica tenemos dos tipos de fórmulas: *state*-fórmulas y *path*-fórmulas cuyos valores de verdad se definen sobre estados o sobre caminos del árbol respectivamente. Las *state*-fórmulas se definen según las siguientes tres reglas:

(S1) las proposiciones atómicas son *state*-fórmulas
(S2) si ϕ y ψ son *state*-fórmulas, entonces $\phi \wedge \psi$ y $\neg\phi$ también son *state*-fórmulas
(S3) si ϕ es una *path*-fórmula, entonces $A\phi$ y $E\phi$ son *state*-fórmulas

Las siguientes reglas definen la sintaxis de las *path*-fórmulas:

(P1) toda *state*-fórmula es una *path*-fórmulas
(P2) si ϕ y ψ son *path*-fórmulas, entonces $\phi \wedge \psi$ y $\neg\phi$ también son *path*-fórmulas
(P3) si ϕ y ψ son *path*-fórmulas, entonces $\bigcirc\phi$ y $\phi \mathcal{U} \psi$ son también *path*-fórmulas

La sintaxis de la lógica CTL se obtiene imponiendo algunas restricciones a las reglas anteriores de CTL*. Formalmente se sustituyen las reglas P1-P3 por la regla:

(P0) si ϕ y ψ son *state*-fórmulas, entonces $\bigcirc\phi$ y $\phi \mathcal{U} \psi$ son *path*-fórmulas.

Informalmente, lo que se hace es impedir que puedan aparecer de forma consecutiva dos o más operadores temporales. En la práctica podemos pensar que tenemos los siguientes operadores temporales: $A\Box$, $E\Box$, $A\Diamond$ y $E\Diamond$ ya que toda fórmula dominada por un operador temporal debe estar precedida de un cuantificador para convertirse en una *state*-fórmula.

Como nota importante y aunque no se demuestre en este texto, diremos que CTL* subsume tanto a LTL como a CTL. Sin embargo, LTL y CTL son incomparables entre sí ya que ni LTL subsume a CTL ni CTL subsume a LTL.

Necesitamos ahora dar significado a las fórmulas y lo haremos definiendo su semántica. CTL* se interpreta sobre un modelo de Kripke $M = (S, I, R, L)$. Un camino completo (*fullpath*) sobre este modelo es una secuencia *infinita* $s = s_0, s_1, \ldots, s_n, \ldots$ de forma que $\forall i (s_i, s_{i+1}) \in R$. Igual que antes, s^i denotará el sufijo $s_i, s_{i+1}, \ldots$ de la secuencia.

Escribiremos $M, s_0 \models \phi$ para decir que una *state*-fórmula ϕ es cierta según el modelo M en el estado s_0. De forma similar, $M, s \models \phi$ denota que una *path*-fórmula ϕ es cierta según el modelo M para la secuencia s.

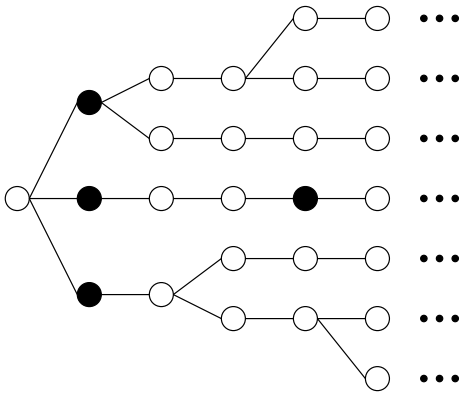
$\models$ se define de forma inductiva a partir de las siguientes reglas:

(S1)	$M, s_0 \models \phi$	sii $\phi \in L(s_0)$
(S2)	$M, s_0 \models \phi \wedge \psi$	sii $M, s_0 \models \phi$ y además $M, s_0 \models \psi$
(S3)	$M, s_0 \models \neg\phi$	sii $M, s_0 \not\models \phi$
(S4)	$M, s_0 \models E\phi$	sii existe un *fullpath* $s = s_0, s_1, \ldots$ en M tal que $M, s \models \phi$
(S5)	$M, s_0 \models A\phi$	sii para todo *fullpath* $s = s_0, s_1, \ldots$ en M tal que $M, s \models \phi$
(P1)	$M, s \models \phi$	sii $M, s_0 \models \phi$
(P2)	$M, s \models \phi \wedge \psi$	sii $M, s \models \phi$ y además $M, s \models \psi$
(P3)	$M, s \models \neg\phi$	sii $M, s \not\models \phi$
(P4)	$M, s \models \phi \mathcal{U} \psi$	sii existe un i tal que $M, s^i \models \psi$ y para todo j menor que i, $M, s^j \models \phi$
(P5)	$M, s \models \bigcirc\phi$	sii $M, s^1 \models \phi$

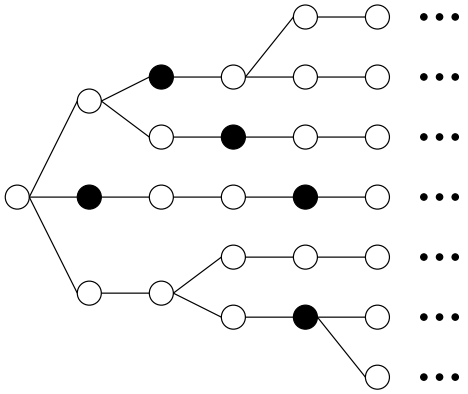
Decimos que una *state*-fórmula (*path*-fórmula) es válida si para todo modelo M y todo estado s_i (camino s) de M, se cumple que $M, s_i \models \phi$ ($M, s \models \phi$). Además, una *state*-fórmula (*path*-fórmula) es satisfacible si existe un modelo M y existe un estado s_i (camino s) de M donde se cumple que $M, s_i \models \phi$ ($M, s \models \phi$). Podemos usar esta misma semántica para interpretar fórmulas CTL.

Ejemplo 3.2.4 En las siguientes figuras, podemos ver representado el comportamiento de los distintos operadores y modalidades. Representaremos como puntos negros el momento en el que se satisface p y grises el momento en el que se satisface q.

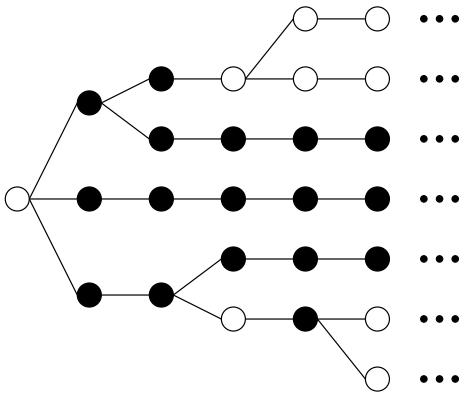
- $s \models A \bigcirc p$



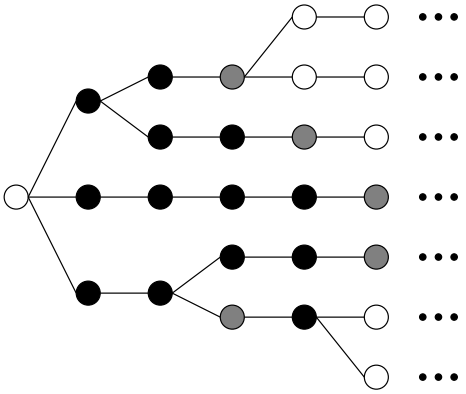
- $s \models E\Diamond p$ y $s \not\models A\Diamond p$



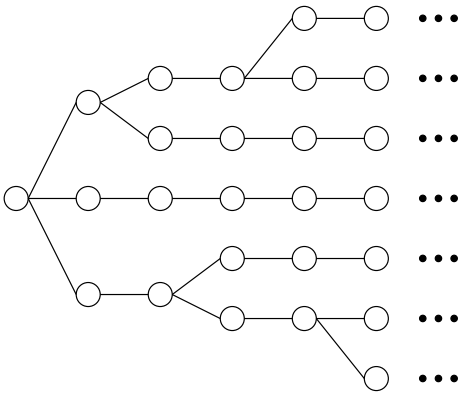
- $s \models A \bigcirc (E\Box p)$



- $s \models A(p \mathcal{U} q)$



- $s \not\models E(true \mathcal{U} p)$



■

3.3. Model Checking Abstracto

En el Capítulo 2 vimos que para abstraer un dominio debemos definir un conjunto de elementos abstractos con una serie de características, una relación entre el conjunto de elementos abstractos y concretos, y debemos definir también una abstracción de las operaciones y expresiones del lenguaje considerado. En esta sección veremos cómo abstraer un modelo. Más adelante veremos cómo definir la semántica abstracta de un lenguaje para evitar así tener que construir el modelo concreto.

3.3.1. La lógica

Vamos a concretar un poco la notación que usaremos a partir de ahora. Sintácticamente, una propiedad estará formada por un conjunto de *proposiciones atómicas Prop*. Llamaremos *literales* al conjunto de expresiones

$$Lit = Prop \cup \{\neg p | p \in Prop\}$$

A partir de los literales podemos definir las *state*-fórmulas:

$$\phi ::= p|\phi \wedge \phi|\phi \vee \phi|A\psi|E\psi$$

y el conjunto de *path*-fórmulas:

$$\psi ::= \phi|\psi \wedge \psi|\psi \vee \psi| \bigcirc \psi|\psi \,\mathcal{U}\, \psi|\psi \,\mathcal{V}\, \psi$$

Nótese que no se define la negación como operador básico. De hecho la negación podrá aparecer únicamente delante de las proposiciones atómicas. Esto hace que sea necesario el uso del operador dual al $\mathcal{U}$ que se define como sigue:

$$(\psi_1 \,\mathcal{V}\, \psi_2) \equiv \neg(\neg\psi_1 \,\mathcal{U}\, \neg\psi_2)$$

La semántica que se usará es similar a la presentada en la sección anterior. Recordemos que las *state*-fórmulas deben su nombre al hecho de que la propiedad toma como punto de referencia un punto determinado, mientras que las *path*-fórmulas analizan la propiedad para un camino. Sin embargo, un único punto puede verse también como un camino, por ello las *state*-fórmulas son en realidad también *path*-fórmulas.

Para hacer la especificación de propiedades más sencilla e intuitiva, normalmente se usan una serie de abreviaturas. Nótese que dichas abreviaturas no son aleatorias, sino que se ha probado la equivalencia con respecto a los operadores básicos. Escribimos $\neg\psi$ para representar la forma normal negada de la fórmula ψ, y para hallar dicha forma normal podemos aplicar las siguientes reglas:

- **R1:** $\neg(\psi_1 \wedge \psi_2) \rightarrow \neg\psi_1 \vee \neg\psi_2$
- **R2:** $\neg(\psi_1 \vee \psi_2) \rightarrow \neg\psi_1 \wedge \neg\psi_2$
- **R3:** $\neg A\psi \rightarrow E\neg\psi$
- **R4:** $\neg E\psi \rightarrow A\neg\psi$
- **R5:** $\neg \bigcirc \psi \rightarrow \bigcirc\neg\psi$
- **R6:** $\neg(\psi_1 \,\mathcal{U}\, \psi_2) \rightarrow \neg\psi_1 \,\mathcal{V}\, \neg\psi_2$
- **R7:** $\neg(\psi_1 \,\mathcal{V}\, \psi_2) \rightarrow \neg\psi_1 \,\mathcal{U}\, \neg\psi_2$

Otras abreviaturas son las cásicas $\mathit{true} \equiv \psi_1 \vee \neg\psi_1$, $\mathit{false} \equiv \psi_1 \wedge \neg\psi_1$, $\psi_1 \rightarrow \psi_2 \equiv \neg\psi_1 \vee \psi_2$, $\Diamond\psi \equiv \mathit{true}\,\mathcal{U}\,\psi$, $\Box\psi \equiv \mathit{false}\,\mathcal{V}\,\psi \equiv \neg(\mathit{true}\,\mathcal{U}\,\neg\psi)$,

El model checking abstracto introducido en [Dam96, DGG97] tiene en cuenta dos fragmentos de la lógica CTL* introducida: $\forall$CTL* y $\exists$CTL*. Estas dos lógicas se definen como la CTL*, con la condición añadida de que en la forma normal negada de las fórmulas sólo puede aparecer cuantificadores universales (respectivamente existenciales). Así pues, es fácil identificar cuando una fórmula pertenece a uno de estos fragmentos siempre y cuando nos fijemos en la forma normal negada, ya que de otra forma podríamos confundirlos.

3.3.2. Sistemas de transiciones

Un sistema de transiciones sobre un conjunto S de estados es un par (S, R) donde R es una relación $R \subseteq S \times S$. Un camino π es una secuencia infinita $\pi = s_0 \cdot s_1 \cdot s_2 \ldots$ de estados, para los que se cumple que para todo $i \in \mathbb{N}, R(s_i, s_{i+1})$. Al elemento i-ésimo del camino lo denotaremos como $\pi(i)$. π^n denotará el sufijo de π cuyo primer elemento es $\pi(n)$.

Esta es la definición más general de sistema de transición, sin embargo un sistema de transición puede tener algunos atributos adicionales, como el conjunto de estados iniciales $I \subseteq S$. En función de este conjunto I, se define la noción de *alcanzabilidad*. Decimos que un camino π es un t-camino si su primer estado es t. Podemos decir ahora que un estado s es alcanzable si existe un t-camino (siendo $t \in I$) que pase por s.

Al sistema de transiciones se le puede añadir también una *interpretación* que asocie a cada elemento del conjunto de literales el conjunto de estados donde dicho literal es válido. A esta función de interpretación le imponemos una restricción, ya que no permitimos que la versión negada de una proposición atómica esté asociada a un estado donde ya estaba asociada dicha proposición atómica (sin negar). Pueden existir estados donde ni la proposición atómica p ni el literal $\neg p$ estén, permitiendo de esta forma la existencia de estados donde el valor de una determinada proposición sea *desconocido*.

Es sencillo asociar la noción de sistema de transiciones con estos dos atributos a una estructura de Kripke. El etiquetado de la estructura da la interpretación de los literales sobre los estados. A continuación vamos a definir la noción de simulación entre estructuras de Kripke. Esta noción es importante porque con ella podremos relacionar modelos y asegurar que ciertas propiedades se preservan cuando pasemos de uno a otro. Las simulaciones se definen a partir de una relación entre estados de las dos estructuras de Kripke. Es decir, se establece una relación entre estados de un sistema y estados del otro cumpliendo una serie de requisitos.

Definición 3.3.1 *Dadas dos estructuras de Kripke* $K_1 = (S_1, I_1, R_1, L_1)$ *y* $K_2 = (S_2, I_2, R_2, L_2)$. $\sigma \subseteq S_1 \times S_2$ *es una relación de simulación entre* K_1 *y* K_2 *si cumple que para todo* $s \in S_1$ *y* $t \in S_2$, *si* $\sigma(s, t)$ *entonces:*

1. $L_1(s) = L_2(t)$,
2. *para todo* s' *tal que* $R(s, s')$, *existe un* t' *tal que* $R_2(t, t')$ *y además* $\sigma(s', t')$.

Vamos a introducir un ejemplo que iremos desarrollando a lo largo del resto del capítulo para ilustrar la técnica de abstracción.

Ejemplo 3.3.2 Supongamos que tenemos queremos representar un protocolo de acceso mutuamente excluyente por parte de dos procesos que se ejecutan en paralelo. Representamos el sistema como dos matemáticos que se encuentran sentados a la mesa listos para comer. Los dos matemáticos deben turnarse para comer, de forma que su estado será *pensante* o *comiendo* de forma alternativa.

Para comer, cada matemático tendrá que consultar una variable del sistema (n) de forma que, si el valor de la variable es *par*, entonces el primer matemático podrá comer, mientras que si es *impar*, el que comerá será el segundo matemático. En cualquier caso, una vez terminado de comer, los matemáticos asignaran un valor nuevo a la variable n, cada uno de forma distinta.

Inicialmente, los dos matemáticos están pensando y la variable n tiene un valor arbitrario.

■

El objetivo final es verificar las siguientes propiedades:

1. los matemáticos tienen acceso mutuamente excluyente al comedor
2. los matemáticos no se bloquean, es decir, si uno está comiendo, tarde o temprano el otro matemático también comerá

La descripción dada para el ejemplo es intuitiva y clara para nosotros, pero no podemos usarla como entrada de un algoritmo. Por ello es necesario codificar el problema de forma que sea manejable de forma automática. Normalmente se recurre a los lenguajes de programación, pero también se pueden usar modelos como los introducidos para el *model checking*.

Ejemplo 3.3.3 (Continuación) Las variables m_0 y m_1 representan el estado en el que se encuentra el primer y segundo matemático respectivamente. Estas variables pueden tomar dos valores: *pensando* o *comiendo*. El conjunto de estados del sistema serán configuraciones con tres componentes: el estado del primer matemático, el estado del segundo y el valor de la variable de control n. Así pues, los estados Σ serán tuplas del conjunto

$$\{pensando, comiendo\}^2 \times \mathbb{N}\backslash\{0\}$$

y se representarán de la forma $\langle m_0, m_1, n\rangle$.

Para definir las posibles transiciones que pueden darse durante la ejecución del sistema, definimos cuatro acciones. La primera (Acción 1) nos dice que si el primer matemático está pensando y n es impar, entonces el primer matemático pasará a estar comiendo:

$$m_0 = pensando, odd(n) \rightarrow m_0 := comiendo$$

De forma similar se definen la otras tres acciones (Acción 2, Acción 3 y Acción 4 respectivamente):

$$m_0 = comiendo \rightarrow m_0 := pensando, n := 3 * n + 1$$
$$m_1 = pensando\, even(n) \rightarrow m_1 := comiendo$$
$$m_1 = comiendo \rightarrow m_1 := pensando, n := n/2$$

La Figura 3.2 muestra el comportamiento de cada uno de los matemáticos de forma gráfica. Nótese que **no** se trata de una estructura de Kripke.

■

Ahora debemos codificar también las propiedades que hemos mencionado antes. Para ello usaremos la lógica temporal CTL*.

Ejemplo 3.3.4 (continuación) La primera propiedad que queremos especificar dice que *nunca podrán estar los dos matemáticos comiendo al mismo tiempo*. Teniendo en cuenta que A cuantifica sobre todos los posibles caminos, y que $\Box$ significa *siempre*, la propiedad la podemos escribir de la siguiente forma:

$$\forall\Box\neg(m_0 = comiendo \wedge m_1 = comiendo)$$

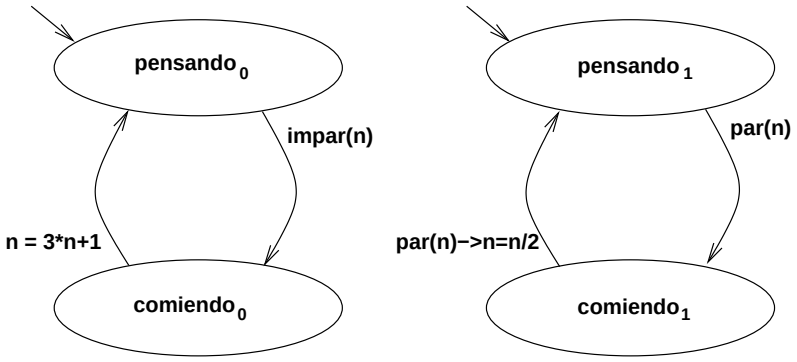


Figura 3.2: Representación gráfica de los matemáticos

La segunda propiedad mencionada en la descripción se puede modelar con dos afirmaciones: *que siempre ocurre que si un matemático está comiendo, entonces eventualmente, independientemente del camino que se tome, el otro matemático comerá* y viceversa.

$$\forall\Box(m_0 = comiendo \rightarrow \forall\Diamond m_1 = comiendo)$$
$$\forall\Box(m_1 = comiendo \rightarrow \forall\Diamond m_0 = comiendo)$$

■

3.3.3. Abstracción del modelo

Para abstraer los modelos de forma que podamos garantizar la preservación de resultados entre el análisis abstracto y el sistema real, hay que cumplir ciertas condiciones a la hora de abstraer el modelo. Por ejemplo, las propiedades de seguridad se preservan entre los modelos abstractos y concretos siempre y cuando el resultado de los cálculos sobre la concreción de los objetos abstractos esté incluído en la concreción del resultado abstracto calculado (usando las versiones abstractas de los operadores).

En el *model checking* abstracto, contamos con las mismas herramientas y estructuras que en el clásico. Empezamos definiendo el modelo abstracto, que será una versión abstracta de la estructura de Kripke tradicional. La idea es la de definir una función ξ que relacione (establezca una simulación entre) la estructura de Kripke concreta con la versión abstracta: $\xi \subseteq KS \times KS_\alpha$ (KS es el conjunto de estructuras de Kripke mientras que KS_α es el conjunto de estructuras de Kripke abstractas), pero teniendo en cuenta que la relación entre ellas debe asegurar que las propiedades CTL* se preservan, al menos de forma débil:

$$\forall K \in KS, A \in KS_\alpha, (\xi(K, A) \Rightarrow \forall\phi \in CTL^*(K \models \phi \Leftarrow A \models \phi))$$

Para definir la relación ξ, se define una relación ρ entre estados (concretos) de K y estados (abstractos) de A.

La idea es obtener una definición constructuva de la relación entre modelos, por ello transformamos la propiedad de preservación en términos de estados en vez de usar modelos

$$\forall c \in S, a \in S_\alpha, (\rho(c, a) \Rightarrow \forall\phi \in CTL^*((K, c) \models \phi \Leftarrow (A, a) \models \phi))$$

siendo S el conjunto de estados del modelo concreto K y S_α el conjunto de estados del modelo abstracto A. Esta relación se integra dentro de una conexión de Galois (α, γ), por

lo que asumimos que S_α es un retículo completo con un orden de aproximación $\sqsubseteq$ y que tenemos una inserción de Galois (α, γ) de $(\wp(S), \subseteq)$ a $(S_\alpha, \sqsubseteq)$. Recordemos que $a \sqsubseteq b$ si y sólo si $\gamma(a) \subseteq \gamma(b)$. El hecho de que consideremos como universo concreto $\wp(S)$ y no S es un aspecto meramente técnico. En realidad el universo continuará siendo S, pero en la relación usamos el conjunto potencia para de esta forma tener de forma automática un retículo completo bajo inclusión de conjuntos como vimos en el tema anterior.

Observemos ahora el ejemplo que hemos introducido. Debido al uso de la variable de control entera, el espacio de estados del sistema es infinito. Una de las formas de manejar dicho espacio de estados de forma efectiva es mediante la interpretación abstracta. Vamos a definir el dominio abstracto que usaremos para este ejemplo concreto.

Ejemplo 3.3.5 (Definición del dominio) Vamos a representar el dominio de las variables de estado de los matemáticos de forma similar a como se hace en el mundo concreto, es decir, tendremos dos valores: *comiendo* y *pensando*.

Los valores de la variable de control vamos a agruparlos en dos valores abstractos *par* e *impar*. De esta forma pasamos de un universo de estados infinito, a uno finito ya que el dominio de la variable entera que era la responsable del gran tamaño del conjunto de estados ahora se limita a dos valores posibles: *par* e *impar*. Además de los valores par e impar, añadimos $\top$ y $\bot$ (ver Capítulo 2).

En la Figura 3.3 se muestran los dominios abstractos de los dos tipos de variables del sistema.

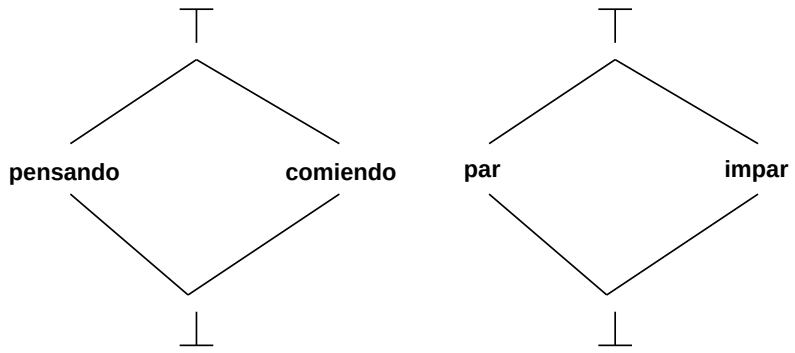


Figura 3.3: Dominios abstractos

Para relacionar lo abstracto y lo concreto, se definen las funciones de abstracción y concreción α y γ:

$$\alpha(n) \begin{cases} par & \text{si } n \, mod \, 2 = 0 \\ impar & \text{si } n \, mod \, 2 \neq 0 \end{cases}$$

$$\gamma(par) = \{2, 4, 6, 8, \ldots\}$$
$$\gamma(impar) = \{1, 3, 5, 7, \ldots\}$$

■

Ahora se puede ver claramente cómo tenemos un espacio de estados finito:

$$\Sigma_\alpha = \{\bot, pensando, comiendo, \top\}^2 \times \{\bot, par, impar, \top\}$$

Preservación de resultados

Tradicionalmente, el *model checking* abstracto se ha centrado en la preservación de propiedades de seguridad universales. Sin embargo, es posible construir modelos que preserven propiedades algo más ambiciosas, o al menos con una naturaleza distinta.

La preservación de resultados depende de la relación que exista entre los modelos. Si tenemos dos estructuras isomorfas, satisfarán las mismas propiedades de la lógica de primer orden. La correspondencia entre las estructuras será una relación de equivalencia.

En particular, en el *model checking* abstracto podemos tener preservación *débil* o preservación *fuerte*. En la preservación débil toda descripción de un elemento concreto (es decir, su abstracción) debe proporcionar información fiable acerca del elemento concreto cuando la respuesta es positiva.

$$\forall c \in C, a \in A\ (\alpha(c) = a \Rightarrow \forall \phi \in L(c \models \phi \Leftarrow a \models^{\alpha} \phi))$$

siendo L la lógica en la que se expresan las propiedades.

Por otro lado, con la preservación fuerte, además de conservar las propiedades que se satisfacen, también se conservan las propiedades que son falsas:

$$\forall c \in C, a \in A\ (\alpha(c) = a \Rightarrow \forall \phi \in L(c \models \phi \Leftrightarrow a \models^{\alpha} \phi))$$

Existen dos versiones para la preservación fuerte. La anterior denota una relación *buena*, mientras que

$$\forall c \in C, a \in A\ (\alpha(c) = a \Leftrightarrow \forall \phi \in L(c \models \phi \Leftrightarrow a \models^{\alpha} \phi))$$

denota una relación *adecuada* entre modelos.

Si queremos tener preservación fuerte, el nivel de simplificación (abstracción) del modelo dependerá en gran medida del conjunto de propiedades de L ya que sólo las propiedades que no estén en dicho conjunto podrán ser *abstraidas*.

Estructura de Kripke abstracta

Supongamos que tenemos un conjunto S_α de estados abstractos, junto con una inserción de Galois (α, γ) que especifica la conexión entre el mundo concreto y el mundo abstracto. Para definir una *Estructura de Kripke abstracta* necesitamos además:

- Una función L_α que especifique la interpretación de los literales sobre los estados abstractos
- Un conjunto I_α de estados iniciales abstractos
- Una relación de transición R_α entre estados abstractos

Estas tres componentes definen ξ, y de ellas depende en parte la precisión de nuestro modelo abstracto. Antes de entrar en más detalles, vamos a introducir el concepto de concreción de una secuencia de estados en una estructura de Kripke.

Definición 3.3.6 *Para una secuencia de estados $s_\alpha = a_0 \cdot a_1 \ldots$ con $a_i \in S_\alpha$, definimos* $\gamma(s_\alpha) = \{c_0 \cdot c_1 \ldots | \forall i R(c_i, c_{i+1}) \wedge c_i \in \gamma(a_i)\}$

Interpretación (L_α). Para satisfacer la condición de preservación débil, se tiene que cumplir que para todo literal p, $(A, a) \models p \Rightarrow (K, \gamma(a)) \models p$. Es decir, que si el literal se satisface en el modelo abstracto, entonces la concreción correspondiente a dicho modelo satisface también dicho literal. Para aumentar la precisión debemos tener el mayor número de literales que se satisfagan en cada estado del modelo abstracto. Por ello se defome la función L_α como:

Definición 3.3.7 *Para* $p \in Lit, L_\alpha(p) = \{a \in S_\alpha | \gamma(a) \subseteq L(p)\}$

La intuición tras esta definición es que el conjunto de estados abstractos en los que un literal p será satisfecho, será aquéllos cuyas concreciones satisfagan dicho literal p en el modelo concreto.

A partir de esta definición se define la relación $\models_\alpha$ de forma que para todo $a \in S_\alpha$ y $p \in Lit$, $(A, a) \models_\alpha p \Leftrightarrow (K, \gamma(a)) \models p$.

Como último apunte haremos notar que, si $\gamma(a)$ contiene estados concretos, algunos que satisfacen p y otros que satisfacen $\neg p$, entonces se dará el caso en el que ni $a \models_\alpha p$, ni $a \models_\alpha \neg p$. De hecho, $a \not\models_\alpha p$ **no implica** que $a \models_\alpha \neg p$. Es más, cuanto más precisa sea una descripción, más literales satisfará, o dicho al contrario, cuanto más abstracta sea una descripción, menos literales satisfará:

Lema 3.3.8 *Sean* $a, a' \in S_\alpha$, *si* $a' \sqsupseteq a$, *entonces para todo* $p \in Lit$, $a' \models_\alpha p \Rightarrow a \models_\alpha p$.

Estados iniciales. Pasemos ahora a la definición de los estados iniciales del modelo abstracto. Como antes, en esta fase también debemos tener en cuenta las condiciones de preservación débil de resultados. Cuanto más pequeño sea el conjunto de estados iniciales más efectivo será el método, pero con la condición de que al menos deberá incluir al conjunto de estados iniciales del modelo concreto. La situación ideal sería la de definir el conjunto de estados iniciales abstracto I_α de tal forma que $I = \cup\{\gamma(a) | a \in I_\alpha\}$, es decir, que el conjunto de concreciones de los estados en I_α coincida con el conjunto de estados iniciales del modelo concreto. Esto en general no es posible pero sí que podemos definir I_α de forma que obtengamos un conjunto de estados lo suficientemente preciso:

$$I_\alpha = \{\alpha(c) | c \in I\}$$

El resultado de esta operación **no** es equivalente a hacer la abstracción del conjunto I ($\alpha(I)$) ya que la primera nos da en general un conjunto más pequeño. El siguiente ejemplo ilustra esta característica:

Ejemplo 3.3.9 (Selección de conjunto inicial) En la Figura 3.4 los elementos d_0 a d_3 denotan objetos del mundo concreto, mientras que b_1 a b_3 son los objetos abstractos que están relacionados entre sí como muestra la figura. Además, asumimos que el conjunto de estados iniciales del dominio concreto es $I = \{d_3, d_2, d_1\}$.

Con la definición dada, tenemos $I_\alpha = \{b_1, b_3\}$. La concreción de este conjunto I_α es $\{d_3, d_2, d_1\}$. Si calculamos $\alpha(I) = \{b_2\}$ cuya concreción es $\{d_3, d_2, d_1, d_0\}$ y es, por lo tanto, menos preciso que el primer resultado ya que incluye el elemento d_0 que no era parte del conjunto de estados iniciales. ■

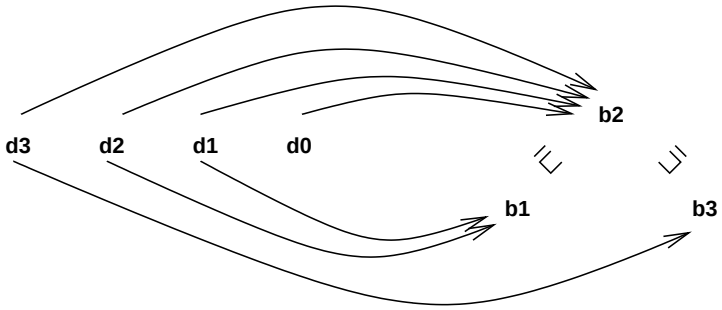


Figura 3.4: Correspondencias entre el modelo concreto y el abstracto

Relación de transición abstracta La tercera y última tarea para obtener la abstracción de una estructura de Kripke es la definición de la relación de transición abstracta. Este es un punto crucial ya que de él depende en gran medida el éxito o fracaso de la técnica de *model checking* abstracto. En este apartado debemos dar respuesta a la pregunta

¿Cuándo debe existir una transición entre el estado abstracto a y el estado abstracto b de nuestro modelo?.

Para que el modelo abstracto preserve las propiedades existenciales del concreto, la relación de transición abstracta debe definirse de forma que la existencia de una transición desde el estado abstracto a, implique que para todo estado $c \in \gamma(a)$, exista un sucesor de c en el modelo concreto que satisfaga la propiedad. Por otro lado, para tener preservación de propiedades universales, el hecho de que el sucesor de a satisfaga cierta propiedad y por tanto que exista esa transición, debe implicar que existe un $c \in \gamma(a)$ cuyos sucesores satisfagan la propiedad.

Para que estas dos condiciones (preservación de propiedades existenciales y de propiedades universales) se cumplan, se puede exigir que la relación R_α sea una *bisimulación* entre los modelos, lo que es un requisito demasiado costoso y que puede implicar que la reducción perseguida del espacio de búsqueda sea muy pequeña. En cualquier caso, la bisimulación implicaría de forma automática que el modelo preservaría las fórmulas CTL* de forma fuerte.

En [Dam96, DGG97], en lugar de imponer esta restricción tan exigente, lo que se hace es definir **dos** relaciones de transición en el modelo. Una de las relaciones preservará las condiciones existenciales, mientras que la otra preservará las condiciones universales. Por este motivo hemos definido los fragmentos de la lótiga temporal $\forall$CTL* y $\exists$CTL*. Podemos distinguir las propiedades según esta clasificación para saber cuál de las dos transiciones del modelo abstracto debemos usar para determinar el valor de verdad de la propiedad. Sin embargo, esta solución es algo más débil que la mencionada anteriormente donde exigíamos bisimulación ya que no se obtiene preservación fuerte para CTL*.

En las dos siguientes subsecciones vamos a describir con más detalle las dos relaciones de transición abstractas del modelo abstracto: la relación de transición *vinculada* y la relación de transición *libre*.

La relación de transición vinculada

Consideremos un estado abstracto $a \in S_\alpha$ que tiene un sucesor $b \in S_\alpha$ para el que se satisface la propiedad $\phi \in CTL^*$. Es decir, que $a \models \exists \bigcirc \phi$. Como tenemos que garantizar la preservación de la propiedad, tiene que ocurrir que todo estado concreto perteneciente al conjunto $\gamma(a)$, tiene que tener un sucesor para el que se satisfaga ϕ.

Vamos a definir una serie de relaciones que serán útiles en esta sección:

Definición 3.3.10 *Las relaciones* $R^{\exists\exists}, R^{\forall\exists} \subseteq \wp(A) \times \wp(B)$ *se definen como:*

- $R^{\exists\exists} = \{(X,Y) | \exists x \in X \exists y \in Y . R(X,Y)\}$
- $R^{\forall\exists} = \{(X,Y) | \forall x \in X \exists y \in Y . R(X,Y)\}$

Hay que notar que estas dos relaciones están definidas sobre superconjuntos, es decir, que devolverá un par de conjuntos que pertenecerán a la relación si existe un (para todo) elemento en X que está relacionado mediante R con un elemento de Y.

Usando esta definición y volviendo a la definición de la relación de transición vinculada, podemos decir que b será un sucesor de a sólo si $R^{\forall\exists}(\gamma(a), Y)$ para algún $Y \subseteq S$ cuya descripción (abstracción) es b, es decir, que $Y \subseteq \gamma(b)$. Con esta definición vemos que estamos garantizando que todo elemento descrito por a estará relacionado con al menos un elemento de Y, que resulta ser un conjunto de elementos descritos por b, y por tanto que satisfacen la propiedad ϕ.

Esta condición garantiza seguridad en el sentido de que las propiedades existenciales son conservadas entre ambos modelos. En cualquier caso, nos interesa también que a tenga cuantos más sucesores mejor, por lo que cada uno de ellos debe ser la descripción más precisa posible de Y, por lo que elegiremos siempre el conjunto Y mínimo y *la mejor* descripción b de Y. Resumiendo, tenemos que

Definición 3.3.11 *La relación de transición abstracta vinculada* R^C_α *se define como*

$$R^C_\alpha(a,b) \Leftrightarrow b \in \{\alpha(Y) | Y \in \min\{Y' | R^{\forall\exists}(\gamma(a), Y')\}\}$$

Ejemplo 3.3.12 (Relación de transición vinculada) En la Figura 3.5 se representa la relación de transición vinculada (a la derecha en el dibujo) definida a partir de un determinado mundo concreto (a la izquierda en el dibujo). Los arcos dibujados con líneas continuas representan la relación de transición tanto en el mundo concreto como en el abstracto, mientras que los arcos dibujados con un trazo discontinuo representan las abstracciones de cada uno de los elementos concretos.

Obsérvese que según el dibujo, $\gamma(a) = \{c_1, c_2\}$, $\alpha(\{d_3\}) = b3$, $\alpha(\{d_2\}) = \alpha(\{d_1\}) = \alpha(\{d_2, d_1\}) = b1$ y que $\alpha(\{d_3, d_1\}) = \alpha(\{d_0\}) = b_2$. Del estado a podemos definir transiciones vinculadas a b_1 y a b_2. Desde el punto de vista de la preservación de propiedades, sería suficiente la transición a b_1. ■

Se cumple que si a es un estado abstracto, y $c \in \gamma(a)$. Si s_α es un camino en la estructura abstracta sobre la relación vinculada R^C_α, entonces existe un camino concreto s en $\gamma(s_\alpha)$. Es decir, que si tenemos una traza en el modelo abstracto que sigue un camino con la relación vinculada, uno de los caminos del conjunto de caminos formado por la concreción de dicha traza abstracta se corresponde con una traza en el modelo real.

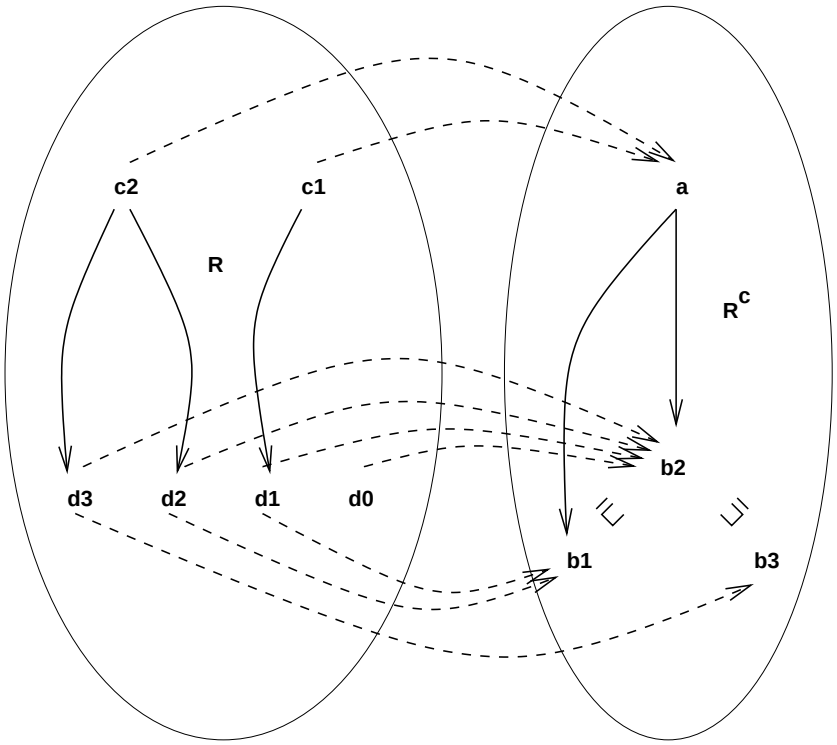


Figura 3.5: Relación de transición vinculada

La relación de transición libre

Ahora consideremos un estado abstracto $a \in S_\alpha$ tal que todo sucesor suyo b satisfaga ϕ. Es decir, que $a \models \forall \bigcirc \phi$. Para satisfacer las restricciones sobre preservación, todo sucesor de todo estado concreto de $\gamma(a)$ debería satisfacer ϕ. Esta cuestión puede girarse de forma que podemos decir que si algún elemento $c \in \gamma(a)$ tiene un sucesor que satisfaga ϕ, entonces a debe tener un sucesor que satisfaga ϕ.

Usando la terminología que hemos empleado para el caso anterior, podemos imponer la condición de que b debe ser un sucesor de a si $R^{\exists\exists}(\gamma(a), Y)$ y b es una descripción (abstracción) de Y. Esta condición hace que las propiedades universales se preserven. Además, en este caso nos gustaría que a tuviera cuantos menos sucesores mejor pero que a la vez, cada uno de ellos sea una descripción de Y lo más precisa posible. Para obtener este resultado elegiremos el mínimo Y que satisfaga la condición y b la mejor representación de dicho Y mínimo.

Definición 3.3.13 (Relación de transición abstracta libre) *Decimos que la relación de transición libre se define como*

$$R^F_\alpha(a, b) \Leftrightarrow b \in \{\alpha(Y) | Y \in \min\{Y' | R^{\exists\exists}(\gamma(a), Y')\}\}$$

Ejemplo 3.3.14 (Relación de transición libre) En la Figura 3.6 se representa la relación de transición libre (en la parte derecha del dibujo) definida a partir de un determinado mundo concreto (a la izquierda en el dibujo). Como en el ejemplo anterior, los arcos

dibujados con líneas continuas representan la relación de transición tanto en el mundo concreto como en el abstracto, mientras que los arcos dibujados con un trazo discontinuo representan las abstracciones de cada uno de los elementos concretos.

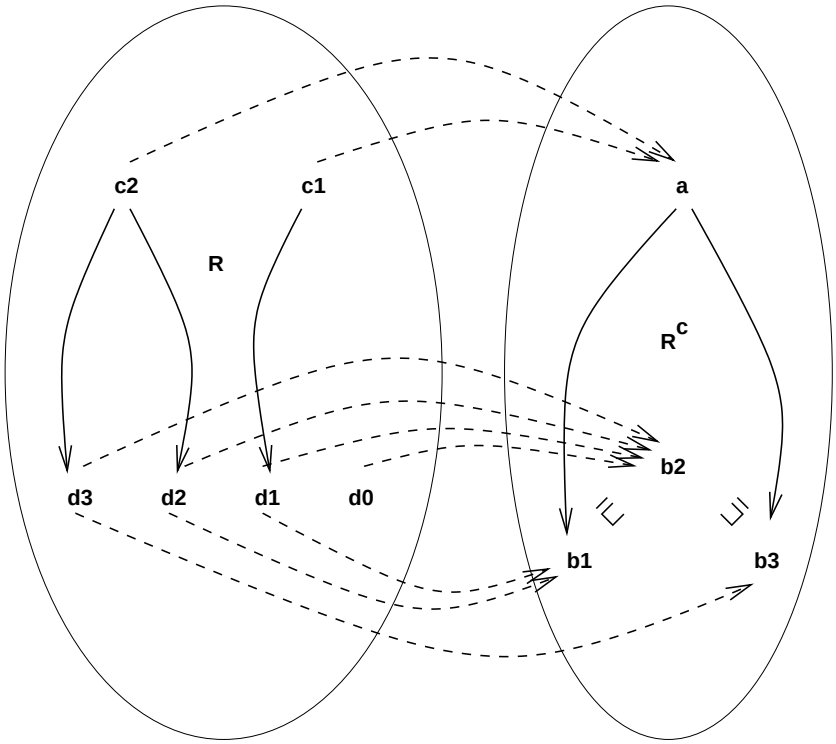


Figura 3.6: Relación de transición libre

En la figura podemos observar que no existe una relación entre el elemento a y b_2. Esto es debido a la exigencia de minimalidad de conjunto. Para cualquier estado abstracto $a \in S_\alpha$, el conjunto mínimo para el que $R^{\exists\exists}(\gamma(a), Y')$ se cumple es un conjunto unitario. En el ejemplo, los Y' son $\{d_3\}$, $\{d_2\}$ y $\{d_1\}$, siendo $\alpha(Y')$ respectivamente b_3, b_1 y b_1. ■

Igual que ocurriía con la relación vinculada, también aquí tenemos una correspondencia entre caminos concretos y abstractos. Si $a \in S_\alpha$ y $c \in \gamma(a)$ y s es un camino del modelo concreto, entonces existe un camino en la relación libre del modelo abstracto en cuya concreción está incluida la traza s.

Para terminar diremos que debido a la condición de minimalidad exigida para estas dos relaciones de transición, en general no tiene por qué satisfacerse que $R^C_\alpha \subseteq R^F_\alpha$.

Modelo abstracto final

En la definición de la estructura de Kripke abstracta se integran los conceptos definidos a lo largo de las secciones anteriores:

Definición 3.3.15 (Sistema de transición mixto) *Un sistema de transición mixto es una tripla (S, F, C) que consiste en un conjunto de estados S y dos relaciones de transición F y C llamadas* libre *(*free*) y* vinculada *(*constrained*) respectivamente.*

Un camino libre *será un camino cuyas transiciones están incluídas en la relación F, mientras que un* camino vinculado *tendrá sus transiciones en la relación C.*

La interpretación de fórmulas CTL sobre este tipo de estructura cambia ligeramente, ya que ahora decimos que:*

- $s \models \forall\psi$ *si y sólo si para todo camino libre* π, $\pi \models \psi$
- $s \models \exists\psi$ *si y sólo si existe un camino vinculado* π *tal que* $\pi \models \psi$

A partir del sistema de transición mixto, se define la noción de estructura de Kripke abstracta:

Definición 3.3.16 (Estructura de Kripke abstracta) *Una estructura de Kripke abstracta es una quintupla $(S_\alpha, F, C, J, L_\alpha)$ donde (S_α, F, C) es un sistema de transición mixto, con estados iniciales J y función de interpretación L. La noción de alcanzabilidad se define en principio según la unión $F \cup C$.*

Dada la estructura de Kripke $K = (S, R, I, L)$ y el conjunto de estados S_α, la función de abstracción $\alpha^M : KS \to KS_\alpha$ mapea la estructura K a la estructura abstracta $K_\alpha = (S_\alpha, R^F_\alpha, R^C_\alpha, I_\alpha, L_\alpha)$ donde R^C_α, R^F_α, I_α y L_α se definen como hemos visto en las secciones anteriores.

Gracias al modo en que se ha definido la estructura de Kripke abstracta, tenemos el siguiente resultado:

Teorema 3.3.17 *Para toda fórmula $\phi \in CTL^*$, $\alpha^M(K) \models_\alpha \phi \Rightarrow K \models \phi$*

3.4. Abstracción de programas

En este capítulo se ha descrito cómo generar un modelo abstracto a partir de uno concreto, pero en realidad a nosotros nos interesa construir un modelo abstracto directamente, sin tener que construir previamente el concreto. Para ello recurrimos a la abstracción de semánticas. De esta forma, en vez de ejecutar un programa de forma tradicional, se ejecutará *de forma abstracta*, obteniendo así una traza abstracta. Querremos definir una semántica abstracta que imite de la forma más precisa posible el comportamiento de la semántica concreta.

La idea fundamental es la de definir una semántica *no estándar* que imite (o represente) la semántica estándar. La semántica abstracta que podamos definir en cada momento dependerá principalmente del lenguaje de programación que se esté usando. Si la semántica concreta de un programa la representamos por la función $\mathcal{I}(P)$, tendremos que definir una relación σ que obtenga la correspondiente semántica concreta $\mathcal{I}_\alpha(P)$. Si hablamos de la semántica de un programa definida por sus puntos fijos, podemos aprovecharnos del siguiente resultado:

Lema 3.4.1 *Sea $(C, \sqsubseteq)$ un cpo, y $f : C \to C$ una función monótona. Sea $(A, \leq)$ un poset, y $f_\alpha : A \to A$ monótona y (α, γ) una conexión de Galois de $(C, \sqsubseteq)$ a $(A, \leq)$. Supongamos que*

$$\alpha \circ f \leq f_\alpha \circ \alpha$$

entonces $\alpha(\mathrm{lfp}(f)) \leq \mathrm{lfp}(f_\alpha)$

Para ilustrar la forma de definir la semántica abstracta de un lenguaje de programación vamos a centrarnos en un lenguaje de programación concreto. En particular el lenguaje usado en el ejemplo de los matemáticos, donde se expresan las acciones que se toman cuando determinadas condiciones se satisfacen.

A continuación se define el lenguaje de programación usado de forma más formal. Un programa será un conjunto de acciones de la forma $c_i(\overline{x}) \rightarrow t_i(\overline{x}, \overline{x}')$, donde y es un valor de un conjunto indexado J, $\overline{x}$ representa el vector de variables de un programa, c_i es una condición que depende de los valores de las variables del programa, y t_i representa la actualización $\overline{x}'$ de los valores de las variables del sistema. Un programa de esta forma se ejecuta eligiendo de forma no determinista entre una de las acciones para las que la condición c_i se cumple, y actualizando los valores como viene especificado por t_i.

Llamaremos *Val* al conjunto de valores que puede tomar el vector $\overline{x}$ e $IVal \subseteq Val$ será el conjunto de valores que pueden tener inicialmente. Así pues, cada c_i es un predicado sobre *Val* y t_i es una relación sobre $Val \times Val$.

Para este lenguaje, podemos definir la semántica estándar (concreta) como sigue

Definición 3.4.2 *La función de interpretación concreta $\mathcal{I} : Lang \rightarrow KS$ se define como sigue. Sea $P = \{c_i(\overline{x}) \rightarrow t_i(\overline{x}, \overline{x}') | i \in J\}$ en Lang, $\mathcal{I}(P)$ es el sistema de transición (S, I, R) donde:*

- $S = Val$
- $I = IVal$
- $R = \{(\overline{v}, \overline{v}') \in Val^2 | \exists i \in J. c_i(\overline{v}) \wedge t_i(\overline{v}, \overline{v}')\}$.

Ahora asumamos que tenemos un dominio abstracto Val_α relacionado con el concreto mediante una conexión de Galois (α, γ) de $(\wp(Val, \subseteq)$ a $(Val_\alpha, \sqsubseteq)$. Definimos entonces dos tipos de interpretaciones abstractas de cada c_i y t_i que se corresponderán con las transiciones vinculadas y libres del sistema.

Definición 3.4.3 *Para $i \in J$, sea c_i^F, c_i^C las condiciones sobre Val_α y t_i^F y t_i^C las transformaciones sobre $Val_\alpha \times Val_\alpha$:*

- *$c_i^F(a)$ es una interpretación abstracta libre de c_i si, y sólo si, para todo elemento $a \in Val_\alpha$,*
$$c_i^F(a) \Leftrightarrow \exists \overline{v} \in \gamma(a). c_i(\overline{v})$$
- *$t_i^F(a)$ es una interpretación abstracta libre de t_i si, y sólo si, para todo par de elementos $a, b \in Val_\alpha$,*
$$t_i^F(a, b) \Leftrightarrow b \in \{\alpha(Y) | Y \in \min\{Y' | t_i^{\exists\exists}(\gamma(a), Y')\}\}$$
- *$c_i^C(a)$ es una interpretación abstracta vinculada de c_i si, y sólo si, para todo elemento $a \in Val_\alpha$,*
$$c_i^C(a) \Leftrightarrow \forall \overline{v} \in \gamma(a). c_i(\overline{v})$$
- *$t_i^C(a)$ es una interpretación abstracta vinculada de t_i si, y sólo si, para todo par de elementos $a, b \in Val_\alpha$,*
$$t_i^C(a, b) \Leftrightarrow b \in \{\alpha(Y) | Y \in \min\{Y' | t_i^{\forall\exists}(\gamma(a), Y')\}\}$$

Para acabar, definimos la interpretación abstracta $\mathcal{I}_\alpha(P)$ de la semántica de un programa P como el sistema $\widehat{A^M} = (S_\alpha, \widehat{R^F_\alpha}, \widehat{R^C_\alpha}, I_\alpha)$ donde:

- $S_\alpha = Val_\alpha$
- $\widehat{R^F_\alpha} = \{(a,b) \in Val^2 | \exists i \in J. c_i^F(a) \wedge t_i^F(a,b)\}$.
- $\widehat{R^C_\alpha} = \{(a,b) \in Val^2 | \exists i \in J. c_i^C(a) \wedge t_i^C(a,b)\}$.
- $I_\alpha = \{\alpha(\overline{v}) | \overline{v} \in IVal\}$

$\widehat{R^F_\alpha}$ y $\widehat{R^C_\alpha}$ son las llamadas relaciones de transiciones *computadas* libre y vinculada respectivamente.

Ejemplo de los matemáticos: Modelo final

Resumiendo lo que tenemos hasta ahora, tenemos la especificación del sistema y la definición del dominio abstracto, así como la inserción de Galois. Vamos a definir ahora las relaciones de transición del sistema mediante la definición de los operadores del lenguaje.

Nuestro dominio de estados abstractos es, recordemos

$$\Sigma_\alpha = S_\alpha = \{pensando, comiendo, \top\}^2 \times \{par, impar, \top\}$$

Los estados iniciales del sistema serán

$$I_\alpha = \{\langle pensando, pensando, par\rangle, \langle pensando, pensando, impar\rangle\}$$

Notese que este conjunto de estados se corresponde con la definición formal dada: $I_\alpha = \{\alpha(c) | c \in I\}$.

Las Tablas 3.1, 3.2 y 3.3 nos dan la interpretación abstracta libre para el ejemplo. Tenemos que considerar tanto las actualizaciones como las posibles comprobaciones de guardas que se hacen en el programa. Es decir, debemos definir la versión abstracta de la multiplicación por tres, de la suma de uno y de la división por dos, además de las comprobaciones de que un número sea par, impar, o una variable de estado sea igual a pensando o comiendo.

Como ejemplo de interpretación de las tablas, diremos que la entrada *false* en la fila $+1^F$, columna *(par,par)* de la Tabla 3.1, significa que $+1^F$(par,par) es falso, es decir, que para cualquier conjunto minimal Y tal que $+1^{\exists\exists}(\gamma(par), Y)$, tenemos que $\alpha(Y) \neq par$.

Tabla 3.1: Versión abstracta libre de los operadores binarios

FREE	(*par*,*par*)	(*par*,*impar*)	(*par*,$\top$)	(*impar*,*par*)	(*impar*,*impar*)	(*impar*,$\top$)	($\top$,*par*)	($\top$,*impar*)	($\top$,$\top$)
$3*^F$	*true*	*false*	*false*	*false*	*true*	*false*	*true*	*true*	*false*
$+1^F$	*false*	*true*	*false*	*true*	*false*	*false*	*true*	*true*	*false*
$/2^F$	*true*	*true*	*false*	*false*	*false*	*false*	*true*	*true*	*false*

La entrada *true* de la Tabla 3.2, fila par^F, columna *par* nos indica que $par^F(par)$ es cierta, es decir, que $\exists n \in \gamma(par).par(n)$.

Tabla 3.2: Versión abstracta libre de la paridad

FREE	*par*	*impar*	$\top$
par^F	*true*	*false*	*true*
$impar^F$	*false*	*true*	*true*

Tabla 3.3: Versión abstracta libre de los estados

FREE	*pensando*	*comiendo*	$\top$
$(=pensando)^F$	*true*	*false*	*true*
$(=comiendo)^F$	*false*	*true*	*true*

Con este ejemplo vemos también que cuando abstraemos funciones, éstas pueden dejar de ser funciones pasando a ser relaciones como ocurre para $/2^F$: ante una misma entrada podemos obtener dos resultados distintos (ver las dos primeras columnas de la operación).

Usando las tablas definidas para las abstracciones libres, podemos dibujar el modelo abstracto libre a partir del programa. Es decir, construiremos la semántica no estándar definida por las funciones de las tablas (no a partir del modelo concreto). La Figura 3.7 muestra el modelo obtenido con las tablas anteriores:

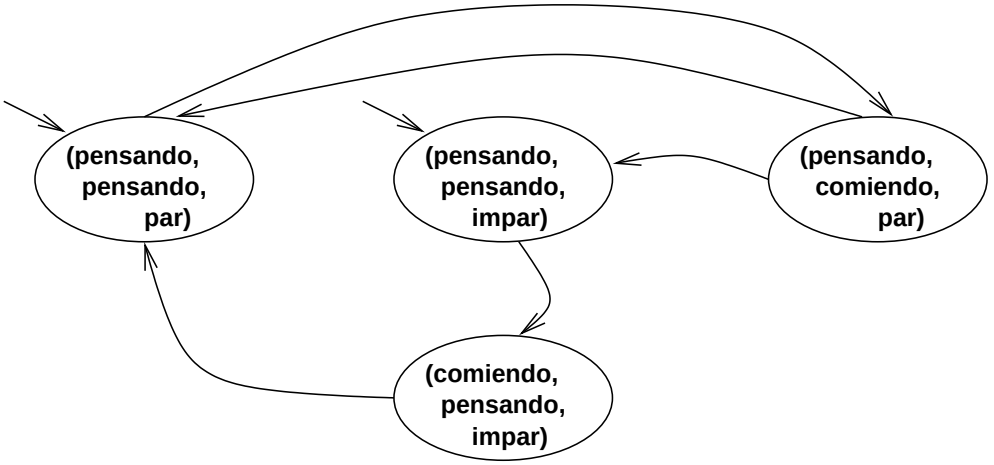


Figura 3.7: Modelo *libre* de los matemáticos

Para ver la utilidad de las versiones vinculadas de abstracción tenemos que modificar ligeramente el problema. Vamos a añadir un tercer proceso concurrente que puede *resetear* el sistema asignando el valor 100 a la variable n. Esta acción puede llevarse a cabo sólo cuando los dos matemáticos estén pensando. Para especificar dicho comportamiento, añadimos al programa la siguiente acción:

$$m_0 = pensando, m_1 = pensando \rightarrow n := 100$$

En este nuevo programa, queremos comprobar que a lo largo de todo camino, en todo

estado existe un camino sucesivo que alcanza un estado de *reseteo*. Si denoramos como *reset* la condición $l_0 = pensando \wedge l_1 = pensando \wedge n = 100$, la propiedad en CTL* se escribiría como:

$$\forall\Box\exists\Diamond reset$$

Se tiene que extender el dominio abstracto con el valor 100, de forma que $\gamma(100) = \{100\}$. Como la fórmula que queremos verificar no es del fragmento $\exists$CTL* ni de $\forall$CTL*, sino que pertenece a CTL*, necesitaremos un sistema de transición mixto.

En las Tablas 3.4, 3.5 y 3.6 se dala interpretación vinculada de los operadores del sistema. Damos sólo una parte de las tablas, la parte necesaria para la construcción del modelo que representa la semántica abstracta. Las tablas libres 3.1, 3.2 y 3.3 deben ser extendidas para considerar el nuevo valor abstracto 100, pero la extensión es trivial, por lo que no incluiremos dicha información.

Tabla 3.4: Versión abstracta vinculada de los operadores binarios

CONSTR.	*(par,100)*	*(par,par)*	*(par,impar)*	*(par,⊤)*	*(100,100)*	*(100,par)*	*(100,impar)*	*(100,⊤)*
$/2^C$	*false*	*false*	*false*	*true*	*false*	*true*	*false*	*false*

CONSTR.	*(impar,100)*	*(impar,par)*	*(impar,impar)*	*(impar,⊤)*
$3*^C$	*false*	*false*	*true*	*false*
$+1^C$	*false*	*true*	*false*	*false*

Tabla 3.5: Versión abstracta vinculada de la paridad

CONSTR.	*100*	*par*	*impar*	⊤
$even^C$	*true*	*true*	*false*	*false*
odd^C	*false*	*false*	*true*	*false*

Tabla 3.6: Versión abstracta vinculada del estado

CONSTR.	*pensando*	*comiendo*	⊤
$= pensando^C$	*true*	*false*	*false*
$= comiendo^C$	*false*	*true*	*false*

La Figura 3.8 muestra la estructura de Kripke abstracta *relevante*, que representa el sistema del protocolo de mutua exclusión de los matemáticos. La estructura relevante se limita a considerar los estados alcanzables desde el estado inicial del sistema.

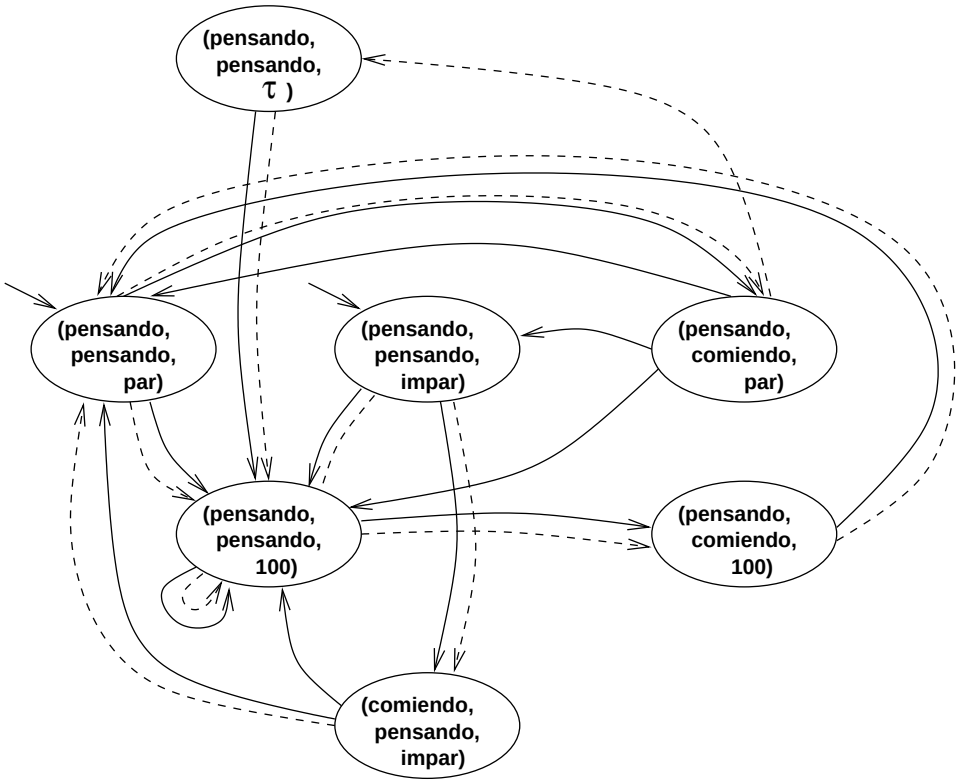


Figura 3.8: Modelo abstracto de los matemáticos

3.4.1. Ejemplo de los matemáticos: Verificación

Recordemos cuáles eran las propiedades que debíamos comprobar de nuestro sistema original (sin la opción de *reset*). Por un lado teníamos una propiedad de seguridad:

$$\forall\Box\neg(m_0 = \mathit{comiendo} \wedge m_1 = \mathit{comiendo})$$

La segunda propiedad, formada por dos fórmulas de viveza, pretendía garantizar que ambos matemáticos iban a nutrirse de forma más o menos frecuente:

$$\forall\Box(m_0 = \mathit{comiendo} \rightarrow \forall\Diamond m_1 = \mathit{comiendo})$$
$$\forall\Box(m_1 = \mathit{comiendo} \rightarrow \forall\Diamond m_0 = \mathit{comiendo})$$

La primera propiedad pertenece al fragmento $\forall$CTL*. Esto nos permite poder verificarla usando únicamente las transiciones libres del modelo (es decir, el modelo que aparece en la Figura 3.7. Observando el modelo libre, podemos ver que ningún estado alcanzable cumple la propiedad $m_0 = \mathit{comiendo} \wedge m_1 = \mathit{comiendo}$. Esto nos garantiza la satisfacción de la propiedad.

Consideremos ahora la primera propiedad de viveza: si el primer matemático come, el segundo en un futuro también lo hará. En este caso, la propiedad vuelve a estar en el fragmento $\forall$CTL*. Mirando de nuevo el modelo de la Figura 3.7, podemos ver que todo camino desde el estado donde $m_0 = \mathit{comiendo}$, en dos pasos de ejecución llega al estado donde $m_1 = \mathit{comiendo}$.

Ninguna de las dos fórmulas anteriores nos causan problemas graves que impidan su demostración. Sin embargo, si cogemos ahora la última fórmula, la que constituye la segunda parte de la segunda propiedad, podemos ver que puede ocurrir que, una vez en el estado donde $m_1 = comiendo$, puede ocurrir que no se alcance el estado donde $m_0 = comiendo$ ya que existe un bucle en la estructura entre el primer estado mencionado y el que aparece más a la derecha en la figura. Este bucle representa los valores que son potencia de 2.

Para resolver el problema podríamos intentar probar la propiedad contraria, es decir, negar la fórmula e intentar probarla pero esto tampoco nos daría el resultado esperado, ya que existen valores para los que el primer matemático sí que llega a comer. El problema además es aún más grave ya que ni siquiera refinando el dominio podemos probar esta propiedad, al menos con un dominio finito, ya que deberíamos considerar de forma independiente cada valor potencia de 2. Existen métodos que permiten verificar esta propiedad que pueden consultarse en caso de interés en [CGL94, DGG97].

Consideremos ahora el modelo modificado, donde habíamos introducido un estado de reinicio (*reset*). Debido a que la propiedad

$$\forall\Box\exists\Diamond reset$$

no pertenece ni al fragmento $\forall$CTL*, ni al $\exists$CTL*, para verificarla tendríamos que usar el modelo completo (donde aparecen los dos tipos de transiciones) de la Figura 3.8. Sobre dicho modelo, tendríamos que interpretar el cuantificador universal sobre los caminos libres, es decir, sobre los arcos discontinuos y el cuantificador existencial sobre la relación vinculada. Observando el modelo podemos ver que efectivamente la propiedad se satisface. Esto quiere decir que la propiedad se satisface también en el modelo concreto.

3.5. Abstracción óptima

Cuando definimos la abstracción de un lenguaje de programación determinado, es decir, su semántica abstracta, puede que no estemos dando una abstracción óptima. Existen métodos para comprobar si una abstracción es óptima o no, pero estos métodos suelen ser demasiado costosos. Una forma de ver si una abstracción **no** es óptima es comprobar si no se cumple que para cualquier guarda del lenguaje de programación, y para todo estado abstracto alcanzable, o bien todos los elementos concretos representados por él satisfacen la guarda, o bien no lo hacen. Esta es una condición necesaria pero no suficiente para la optimalidad del la abstracción. Si algún estado no satisface esta propiedad, directamente sabemos que nuestra abstracción no es óptima.

Como hemos dicho, el determinar la abstracción óptima, o incluso comprobar si una abstracción es óptima o no, es una tarea demasiado compleja en general. Por ello solemos acontentarnos con aproximaciones que, aunque no son óptimas son lo suficientemente precisas para ser útiles, y en caso de que no lo fueran, podrían refinarse para mejorar la precisión.

Existen dos formas básicas de obtener un marco óptimo. Una es obteniendo un dominio abstracto óptimo, y la segunda forma consiste en especificar la semántica abstracta de forma precisa. No vamos a detallar todas las técnicas existentes para el refinamiento de dominios abstractos y de semánticas, pero diremos que puede hacerse tanto usando un

refinamiento de la partición del dominio de las variables, como usando un contraejemplo *falso* como guía. Podemos seguir la traza del programa con respecto al contraejemplo hasta llegar a un punto en el que el dominio de las variables es *incompatible* con el dominio en un estado anterior. Esto nos determina que ha habido algún tipo de *inconsistencia* o *contradicción* en el dominio concreto que no ha sido capaz de detectar el dominio abstracto.

4

Evaluación Parcial

4.1. Introducción

La evaluación parcial es una técnica de optimización de programas basada en la especialización de programas [JGS93, CD93]. El punto fuerte de esta técnica es que puede ser completamente automatizada, y que es capaz de obtener generadores de programas.

La idea fundamental de la técnica es la siguiente: a partir de una función con dos parámetros, especializarla para obtener una función con un solo parámetro; es decir, fijar el valor de uno de los dos parámetros de la función original. La idea no es nueva, ya que aunque no fuera aplicada a programas, surgió a mediados del siglo pasado. Por aquél entonces, la idea no tenía en cuenta la eficiencia de las funciones especializadas que se generaban, argumento que hoy en día sí debe ser considerado.

Así pues, un evaluador parcial tomará como entrada un programa (el que queremos especializar) y alguna entrada de dicho programa. Como salida dará la versión especializada del programa de entrada, que aceptará como entrada a su vez el resto de entradas (las que no dimos como parámetro al evaluador parcial). La salida de dicho programa especializado debe ser la misma que la que hubiéramos obtenido pasando las mismas entradas al programa original.

Ejemplo 4.1.1 El siguiente programa calcula la potencia n de un número x:

$$\begin{aligned} \mathsf{f(n,x)} = \ & \mathsf{if\ n = 0\ then\ 1} \\ & \mathsf{else\ if\ even(n)\ then\ f(n/2,x)\uparrow 2} \\ & \quad \mathsf{else\ x\ *\ f(n\text{-}1),x)} \end{aligned}$$

La versión especializada si le pasamos al evaluador parcial el valor de $n = 5$ sería:

$$\mathsf{f5(x)} = \mathsf{x\ *\ ((x\uparrow 2)\uparrow 2)}$$

Obviamente, si en vez de especializar según el valor de n, especializáramos según el valor de x, la evaluación parcial no conseguiría mejorar nada, ya que el flujo de control del programa no depende completamente de la variable x, sino de n. ■

Ahora bien, ¿cómo se realiza la transformación del programa original al especializado?. Vamos a ver que existen algunas técnicas básicas como son las computaciones simbólicas, el *unfolding* de llamadas de procedimiento, o la *memoization*.

4.1.1. Definición formal de la evaluación parcial

Vamos a asumir que $[\![p]\!]$ denota el significado del programa p:

$$\mathsf{salida} = [\![p]\!]_L[\mathsf{in}_1, \ldots, \mathsf{in}_n]$$

Si la lista de entradas pasadas al programa es completa, es decir, incluye todos los parámetros de p, entonces el resultado será la semántica de entrada/salida asociada a los valores de entrada. Si en la lista faltara algún parámetro, entonces daría como resultado un *programa especializado*. El subíndice L de la ecuación anterior indica que estamos usando L como lenguaje de implementación del programa.

Cogiendo un ejemplo de un programa con dos entradas, tendríamos que la computación tradicional en un solo paso se representaría como:

$$\mathsf{out} = [\![p]\!][\mathsf{in1}, \mathsf{in2}]$$

La computación en dos pasos que hace uso del evaluador parcial *mix* sería:

$$\begin{array}{l} \mathsf{p}_{in1} = [\![mix]\!][p, \mathsf{in1}] \\ \mathsf{out} = [\![\mathsf{p}_{in1}]\!][\mathsf{in2}] \end{array}$$

Automáticamente obtenemos una definición de evaluación parcial combinando las ecuaciones anteriores:

$$[\![p]\!][\mathsf{in1}, \mathsf{in2}] = \underbrace{[\![[\![mix]\!][p, \mathsf{in1}]]\!]}_{\text{evaluador parcial}}\ \mathsf{in2}$$

4.1.2. Motivación

La velocidad o eficiencia de los programas es la principal motivación para el uso de la evaluación parcial como optimización de programas. Un programa especializado es normalmente más eficiente que el original, aunque para asegurar esta propieda debemos definir el algoritmo de evaluación parcial con cuidado. Nótese que un programa especializado es menos general que el original, es decir, la semántica cambia por lo que no son programas equivalentes en general; lo serán únicamente cuando los valores de entrada de las variables que se fijan coincidan.

Se debe llegar a un consenso entre generalidad y eficiencia. No es práctico tener muchos programas muy eficientes que hacen prácticamente la misma operación, pero tampoco lo es tener un programa más general que sea mucho menos eficiente que los pequeños. En estas situaciones, la evaluación parcial es donde cobra significado. Es decir, podemos primero implementar el programa de alto nivel, el cual posiblemente sea más lento que los especializados, pero que normalmente será más fácil de implementar por el programador, y posteriormente usar un evaluador parcial para obtener las versiones más eficientes del mismo, pero esta vez de forma automática.

Gracias a su simplicidad, la evaluación parcial puede ser usada en muchos marcos de trabajo. De todas formas, donde más se ha usado es en el área de generación de procesadores de lenguajes, y en especial de compiladores. En esto nos centraremos en este tema.

La evaluación parcial se puede usar en la informática gráfica para mejorar la eficiencia del cálculo de la incidencia de las fuentes de luz en una escena. También en el ámbito de las

bases de datos ya que podemos especializar la búsqueda a una determinada *query*. En el campo de las redes neuronales, el entrenamiento de las redes suele ser bastante complejo, pero si fijamos la topología de la red, podemos obtener algoritmos mucho más eficientes. También en los cálculos científicos se puede usar la evaluación parcial, por ejemplo fijando un sistema planetario dado para el cálculo de determinadas órbitas.

4.1.3. Generación automática de programas

Observemos la siguiente ecuación, asumiendo que int es un intérprete:

$$\begin{aligned} \mathsf{out} &= [\![\mathsf{source}]\!]_S\ \mathsf{input} \\ &= [\![\mathsf{int}]\!]\ [\mathsf{source},\mathsf{input}] \\ &= [\![[\![\mathsf{mix}]\!][\mathsf{int},\ \mathsf{source}]\ ]\!]\ \mathsf{input} \\ &= [\![\mathsf{target}]\!]\ \mathsf{input} \end{aligned}$$

De estas ecuaciones podemos deducir de las ecuaciones la primera proyección de Futamura, que dice que

$$\mathsf{target} = [\![\mathsf{mix}]\!][\mathsf{int},\ \mathsf{source}]$$

Es decir, que si le damos al evaluador parcial un intérprete y el código del programa como datos sobre los que especializar, obtenemos un programa target *compilado*.

Podemos también generar un compilador aplicando el evaluador parcial a si mismo, fijando la entrada para el intérprete del lenguaje para el que queremos generar el compilador:

$$\mathsf{compiler} = [\![\mathsf{mix}]\!][\mathsf{mix},\mathsf{int}]$$

A esta ecuación se le llama la segunda proyección de Futamura. El programa resultado tomará como entrada un source y obtendrá target. Veámoslo en la siguiente ecuación:

$$\begin{aligned} \mathsf{target} &= [\![\mathsf{mix}]\!]_S\ [\mathsf{int},\mathsf{source}] \\ &= [\![[\![\mathsf{mix}]\!][\mathsf{mix},\ \mathsf{int}]\ ]\!]\ \mathsf{source} \\ &= [\![\mathsf{compiler}]\!]\ \mathsf{source} \end{aligned}$$

Esta forma de generar compiladores de forma automática necesita que el evaluador parcial mix esté escrito en el mismo lenguaje que toma como entrada.

4.2. Evaluación Parcial

En esta sección vamos a profundizar en cómo es posible aplicar evaluación parcial a un lenguaje de programación sencillo. La técnica puede aplicarse a otros lenguajes más complejos como C, lenguajes funcionales, lógicos, etc. [JGS93], pero es más fácil ver cómo funciona si nos limitamos a un lenguaje simplificado.

4.2.1. El lenguaje de programación

El lenguaje de programación que vamos a considerar es un lenguaje imperativo sencillo. La característica principal es que se ejecuta de forma secuencial, siguiendo una serie de comandos, cada uno de los cuales actualiza una componente de un *estado del programa*

interno. Este estado, normalmente consistirá en los valores de ciertos registros, además de una memoria con el valor de las variables.

A continuación mostramos la sintaxis del lenguaje:

⟨Program⟩	::=	read ⟨Var⟩,. . . , ⟨Var⟩; ⟨BasicBlock⟩$^+$
⟨BasicBlock⟩	::=	⟨Label⟩: ⟨Assignment⟩* ⟨Jump⟩
⟨Assignment⟩	::=	⟨Var⟩ := ⟨Expr⟩;
⟨Jump⟩	::=	goto ⟨Label⟩;
	|	if ⟨Expr⟩ goto ⟨Label⟩ else ⟨Label⟩;
	|	return ⟨Expr⟩;
⟨Expr⟩	::=	⟨Constant⟩
	|	⟨Var⟩
	|	⟨Op⟩ ⟨Expr⟩ . . . ⟨Expr⟩
⟨Constant⟩	::=	quote ⟨Val⟩
		for brevity ´Val
⟨Op⟩	::=	hd | tl | cons | . . .
		plus any others needed for writing
		interpreters or program specializers
⟨Label⟩	::=	any identifier or number

Ejemplo 4.2.1 Veamos un ejemplo de un programa que calcula el máximo común divisor de dos números naturales escrito en este lenguaje:

```
read x, y;
1: if x = y goto 7 else 2
2: if x ¡y goto 5 else 3
3: x := y - x
   goto 1
5: y := y - x
   goto 1
7: return x
```

■

El hecho de que usemos este lenguaje de programación y no otro más complicado para introducir la evaluación parcial es debido a que, de esta forma, podemos centrarnos en los problemas relacionados con la técnica y no en los derivados del lenguaje de programación en sí (para los que además podemos encontrar soluciones en la literatura del área). Además, este fue el primer lenguaje para el que se implementó de forma eficaz la técnica, y fue posible en parte por su sencillez. A partir de él pudo mejorarse la técnica e ir aplicándose a lenguajes más complejos. Es más, las técnicas usadas para este lenguaje no sólo se aplican a lenguajes imperativos más complejos, sino también a lenguajes con una naturaleza completamente distinta como pueden ser Scheme o Prolog.

4.2.2. El método

La especialización de programas consiste en la ejecución de tres tareas, cada una de ellas independiente del lenguaje de programación específico que se quiere transformar. Vamos a considerar a partir de ahora únicamente lenguajes de programación deterministas, y además asumiremos que todo programa tiene un conjunto de *puntos de programa* donde

se almacena el punto de control actual en cada instante de tiempo durante la ejecución. En el lenguaje descrito en este capítulo, los puntos de programa son las etiquetas; en un lenguaje funcional serán los nombres de las funciones; en un lenguaje lógico, los puntos de control están definidos por los nombres de predicados, etc. A continuación se describen las tres tareas esenciales del método:

1. Dado el valor de algunas entradas del programa, se obtiene una descripción de todos los estados computacionales alcanzables en función de los valores de entrada.
2. Se redefine el control del programa incorporando datos a los puntos de control, pudiendo dar lugar a diversas versiones especializadas de cada punto de control del programa.
3. El programa resultado normalmente contiene muchas transiciones triviales, por lo que se optimiza usando técnicas tradicionales de simplificación, obteniendo así el programa especializado (o programa *residuo*).

4.3. Evaluación parcial

Dado un programa fuente y sus datos de entrada, un intérprete lo interpreta produciendo una salida final. Dado un programa y sólo una parte de sus datos de entrada, un especializador de programas intentará ejecutar el programa dado mientras sea posible, dando como resultado un programa residuo que ejecutará el resto de la computación cuando los datos de entrada que faltan sean proporcionados.

Vamos a denotar como L al lenguaje de programación introducido en la sección anterior. La memoria del programa es una función de las variables del programa a sus valores actuales. La entrada de un programa p es una lista de valores $\mathsf{d} = (\mathsf{v}_1 \ldots \mathsf{v}_n)$ que inicialmente están asignados a $\mathsf{var}_1, \ldots, \mathsf{var}_n$. Todas las variables que no son de entrada $\mathsf{var}_{n+1}, \ldots, \mathsf{var}_{n+k}$ tienen como valor inicial la lista vacía $()$, por lo que la memoria inicial de los programas será:

$$[\mathsf{var}_1 \mapsto \mathsf{v}_1, \ldots, \mathsf{var}_n \mapsto \mathsf{v}_n, \mathsf{var}_{n+1} \mapsto (), \ldots, \mathsf{var}_{n+k} \mapsto ()]$$

Se asume que las funciones no tienen efectos colaterales en los valores de las variables. Además, las asignaciones, condicionales y goto's se ejecutan de la forma habitual, y return expression acaba la ejecución, dando como valor resultado de la ejecución del programa $[\![p]\!]_{\mathsf{L}}\,\mathsf{d}$ el valor de expression.

Para que los programas resulten más claros, usaremos las estructuras begin ... end, while... do... y repeat... until típicos de otros lenguajes y que pueden transformarse en términos de los constructores básicos, por lo que no son más que azúcar sintáctico.

4.3.1. Programas residuo y especializaciones

En esta sección vamos a definir algunas nociones básicas.

Definición 4.3.1 *Sea* p *un programa escrito en* L *que toma como entrada dos elementos, y sea* $\mathsf{d1} \in \mathsf{D}$. *Entonces, un programa* L *llamado* r *será un programa residuo para* p *con respecto a* $\mathsf{d1}$ *si, y sólo si*

$$[\![\mathsf{p}]\!]_{\mathsf{L}}[\mathsf{d1}, \mathsf{d2}] = [\![\mathsf{r}]\!]_{\mathsf{L}}\,\mathsf{d2}$$

para todo $\mathsf{d2}$.

La definición nos dice que ejecutar el programa r con la entrada d2, da siempre el mismo resultado que ejecutar p con entradas d1 y d2. Ahora nos falta tener un método para construir r de forma automática yua que r es precisamente el programa optimizado objetivo.

Definición 4.3.2 *Un programa escritp en* L *de nombre* mix *es un especializador de programas si, y sólo si, para todo programa* p*, y dato* d1∈D*, el programa* $r = [\![mix]\!]_L[p, d1]$ *es un programa residuo para* p *con respecto a* d1*. De forma simbólica:*

$$[\![p]\!]_L[d1, d2] = [\![([\![mix]\!]_L[p, d1])]\!]_L d2$$

para todo d2∈D*.*

Ejemplo 4.3.3 El siguiente programa podría ser el fragmento de código de un intérprete (ver [JGS93] para ejemplo completo). namelist sería una lista de nombres de variables, y valuelist sería la lista de los valores correspondientes de dichas variables. Los datos de entrada a este fragmento de programa son las tres variables name, namelist y valuelist.

```
while name ≠ hd (namelist) do
begin
      valuelist := tl (valuelist);
      namelist := tl (namelist);
end;
value := hd (valuelist);
```

Supongamos que tenemos un programa que es capaz de especializar al que le pasamos los valores iniciales de las variables name y namelist; por ejemplo, name = z y namelist = (x y z), pero el valor de la tercera variable (valuelist) no lo conocemos. Ya que el control del programa está determinado completamente por las dos variables de las que conocemos los valores, puede ser ejecutado de forma simbólica dando como resultado el siguiente programa residuo:

```
valuelist := tl (valuelist);
valuelist := tl (valuelist);
value     := hd (valuelist);
```

Así pues, el bucle while ha desaparecido ya que el programa residuo contiene únicamente las sentencias que no pueden ejecutarse con los datos proporcionados hasta el momento. ■

El resultado en el ejemplo anterior se podría optimizar usando diversas técnicas, pero nosotros nos vamos a centrar en la evaluación parcial, dejando las simplificaciones a un lado.

4.4. Técnicas de especialización

Un estado computacional en un lenguaje imperativo será un punto pp del programa, que representará el punto de control donde se encuentra en ese momento la ejecución, y una memoria store que contendrá los valores de las variables del programa. Representaremos

los valores de las variables como una lista como ya hemos mencionado. Cuando se ejecuta una asignación, se debe actualizar la memoria y el punto de control. Cuando se ejecuta un condicional o un goto, entonces sólo se actualiza el punto de control y no la memoria.

Supongamos ahora que disponemos de sólo una parte de los datos de entrada. En ese caso, tanto el estado inicial como los siguientes estarán incompletos. Una forma de representar la memoria incompleta del programa es dividir las variables en aquéllas estáticas (sabemos su valor en timpo de especialización) y aquéllas dinámicas. De esta forma, un estado computacional parcial será un par (pp,vs) de forma que vs es una lista de los valores de las variables estáticas.

A esta clasificación entre variables estáticas y dinámicas se le llama *división*. Existen distintas formas de realizar la división o clasificación. Nosotros asumismo que la división es uniforme en cualquier punto del programa, es decir, que siempre es válida esta división (no cambia a lo largo del código). Por último, asumiremos también que la división es congruente, en el sentido de que toda expresión estática, debemos poder obtenerla sólo a partir de los datos estáticos del punto precedente. En otras palabras, toda variable que dependa de algún valor dinámico será catalogada como dinámica.

Ejemplo 4.4.1 (División) En el siguiente código podemos clasificar las variables *name* y *namelist* como estáticas y *value* y *valuelist* como dinámicas. Esta división cumple las propiedades mencionadas en el párrafo anterior.

```
while name ≠ hd (namelist) do
begin
      valuelist := tl (valuelist);
      namelist := tl (namelist);
end;
value := hd (valuelist);
```

■

La elección de la división de las variables del sistema puede ser una tarea más complicada de lo que parece a simple vista. Vamos a ver por qué usando un ejemplo. Observemos el siguiente código:

```
iterate: if Y ≠ 0 then begin
           X := X + 1;
           Y := Y - 1;
           goto iterate;
         end;
```

Si conociéramos el valor inicial de X, podríamos decidir que dicha variable pertenecerá al fragmento de variables estáticas, mientras que Y será dinámica, ya que el valor de la variable X en cada iteración depende (aparentemente al menos) sólo de su valor precedente. Sin embargo, esta división no funciona ya que el valor de X depende también del número de iteraciones que se hagan, y este número depende del valor de Y. En este ejemplo por tanto, X debería clasificarse como variable dinámica

4.4.1. Especialización por punto de control

La idea de la especialización por punto de control es la de incorporar los valores de las variables estáticas en los puntos de control de forma explícita. Vamos a ilustrar la técnica usando un ejemplo.

Ejemplo 4.4.2 Lo primero que vamos a hacer es convertir el ejemplo anterior de forma que, de forma que aparezcan únicamente etiquetas y *goto*'s:

```
search: if name=hd(namelist) goto found else cont;
cont:   valuelist := tl (valuelist);
        namelist := tl (namelist);
        goto search;
found:  value := hd (valuelist);
```

Este programa es equivalente al mostrado anteriormente. ■

Siempre que queramos especializar un programa, tendremos que hacerlo en función de unos datos determinados. Por ello, el paso siguiente a la conversión sería la elección de la división de variables estáticas y dinámicas. En nuestro ejemplo usaremos la división discutida anteriormente.

Ejemplo 4.4.3 La división será:

- variables estáticas: name y namelist
- variables dinámicas: value y valuelist

Los valores que usaremos para especializar el programa, serán:

- name = z
- namelist = (x y z)

Por lo tanto, el valor inicial de *vs* (que recordemos que es la lista de variables estáticas del sistema) será

$$\mathsf{vs} = (\mathsf{z}, (\mathsf{x}\ \mathsf{y}\ \mathsf{z}))$$

■

Una vez definidos los valores iniciales de las variables estáticas se determina cuáles serán los posibles valores de dichas variables en cada punto de control. Más adelante mostraremos cómo es posible automatizar dicho proceso. De momento nos conformamos con dar la idea intuitiva de qué son los puntos de control especializados, es decir, los puntos de control con valores de las variables presentes:

Ejemplo 4.4.4 Con los datos definidos en el ejemplo, a lo largo de la ejecución vs puede tener varios valores que serían coherentes con el programa.

- En primer lugar, en el punto de control cuya etiqueta es search, vs puede tener los valores: (z, (x y z)), (z, (y z)) y (z, (z));
- En el punto de control cuya etiqueta es cont, vs sólo puede tomar dos valores: (z, (x y z)) y (z, (y z));

- Por último, en el punto de control con etiqueta found puede tener el valor (z, (z)).

■

Definición 4.4.5 (Punto de programa especializado) *Un punto de programa especializado es una tuple* (pp,vs) *donde* pp *es un punto de programa del código original, y* vs *es un conjunto de valores de las variable sestáticas del sistema*

A partir de la definición de punto de control especializado, podemos definir el conjunto de puntos de control especializados de un programa determinado. De hecho este conjunto será el que nos permita obtener de forma automática el programa optimizado.

Definición 4.4.6 *Llamamos* poly *al conjunto de todos los puntos de programa especializados* (pp,vs) *que son alcanzables durante la ejecución del programa.*

Es decir, que en poly tendremos todos los puntos de control que puedan aparecer en el programa especializado.

Ejemplo 4.4.7 En nuestro ejemplo, y tal y como habíamos adelantado,

```
poly = { (search,(z,(x y z))), (search,(z,(y z))),
         (search,(z,(z))),
         (cont,(z,(x y z))), (cont,(z,(y z))),
         (found,(z,(z))) }
```

■

Debemos, por un lado, saber cómo obtener los puntos de control, y por otro, cómo generar el código optimizado a partir de dichos puntos de control. Vamos a asumir que hemos obtenido los puntos de control automáticamente y que queremos generar el código. Para ello veremos que existen una serie de tablas que podemos usar para, a partir del código original, ir construyendo el optimizado.

Supongamos que tenemos un punto especializado (pp,vs). En el código original, la etiqueta pp estará asociada a un bloque de código básico, que consistirá en una secuencia de comandos. El código generado será la concatenación del código generado para cada uno de dichos comandos.

Necesitamos tener implementadas una serie de funciones que nos permitan trabajar con el código. Por ejemplo, supongamos que exp es una expresión y s es la lista de valores de las variables estáticas. Entonces necesitaremos por un lado una función eval(exp, vs) que devolverá el valor de una expresión estática exp. Además, necesitaremos una función reduce(exp,vs) que, en caso de que exp no sea estática, realizara desdoblamientos y sustituciones de las partes estáticas de exp, obteniendo así una expresión especializada. Por ejemplo, si vs nos dice que una variable b tiene el valor 2, entonces reduce transformaría la expresión b * b + a a la expresión 4 + a.

A continuación mostramos las tablas que definen el código generado para cada posible comando del lenguaje:

Comando	Acción de especialización	Código generado
`X := exp` `X` dinámica	`reduced_exp:=reduce(exp,vs)`	`X := reduced_exp`
`X := exp` `X` estática	`val:=eval(exp,vs)` `vs:=vs[X↦val]`	
`return exp`	`reduced_exp:=reduce(exp,vs)`	`return reduced`
`goto pp'`	`goto (pp',vs)`	

La tabla de arriba muestra la generación para todos los comandos excepto para el condicional. A continuación mostramos de forma separada la generación de código cuando nos encontramos con un comando condicional de tipo if exp goto pp' else pp"

Casos	Acción de especialización	Código generado

`exp` dinámico	`reduced_exp:=reduce(exp,vs)`	`if reduced_exp`
		`goto (pp',vs)`
		`else (pp'',vs)`

`exp` estático `val` cierto	`val:=eval(exp,vs)`	`goto (pp',vs)`
`exp` estático `val` falso	`val:=eval(exp,vs)`	`goto (pp'',vs)`

Ahora vamos a considerar cómo calcular el conjunto poly de forma automática ya que hasta este momento hemos supuesto que ya teníamos calculado ese conjunto.

Vamos a asumir que pp0 es la primera etiqueta del programa, y que vs0 es el valor inicial de las variable sestáticas. Está claro que (pp0,vs0) debería estar en el conjunto poly. Además, todos los puntos especializados alcanzables desde un punto que pertenezca a poly, deberían estar también en poly por definición. Por ello podemos decir que poly no es mas que el cierre de vs0 bajo la relación *alcanzable desde*.

Los puntos alcanzables dependerán del tipo de comando que estemos considerando. Habrá comandos que puedan tener varios sucesores, igual que habrá comandos que afectarán al valor de las variables estáticas. En particular, el comando return no tendrá sucesores, mientras que el condicional podrá tener hasta dos sucesores. El resto de comandos tendrán siempre un sucesor. En la siguiente tabla damos las reglas para saber los sucesores de un punto:

punto de control	sucesores de pp, vs	
`return`	{}	
`goto pp'`	{(pp',vs)}	
`if exp goto pp'` `else pp'`	{(pp',vs)} {(pp'',vs)} {(pp',vs),(pp'',vs)}	si `exp` cierto si `exp` falso si `exp` dinámica

Una vez sabemos cómo computar los sucesores y cómo generar el código para cada comando, en la Figura 4.1 damos el algoritmo que, dado un programa y los valores de las variables estáticas, obtiene el programa especializado.

Ejercicio 4.4.8 *Obtened el programa residuo a partir del programa fuente siguiente las variables estáticas dadas.*

```
poly := { (pp0, vs0) };
while poly tenga puntos no marcados (pp,vs) do
begin
     mark (pp,vs);
     generar el código para los bloques básicos empezando en pp usando vs;
     poly := poly ∪ successors(pp,vs)
end
```

Figura 4.1: Pseudocódigo del algoritmo de evaluación parcial.

- *Programa:*

```
search: if name=hd(namelist) goto found else cont;
cont:   valuelist := tl (valuelist);
        namelist := tl (namelist);
        goto search;
found:  value := hd (valuelist);
```

- *Valor de las variables estáticas* name *y* valuelist*:*

  vs = (z, (x y z))

El resultado final del ejercicio siguiendo el algoritmo de la Figura 4.1, debería ser el siguiente:

```
(search, (z, (x y z))): goto (cont, (z, (x y z)));
(cont, (z, (x y z))):   valuelist := tl (valuelist);
                        goto (search, (z, (y z)));
(search, (z, (y z))):   goto (cont, (z, (y z)));
(cont, (z, (y z))):     valuelist := tl (valuelist);
                        goto (search, (z, (z)));
(search, (z, (z))):     goto (found, (z, (z)));
(found, (z, (z))):      value := hd (valuelist);
```

El resultado como puede verse es un poco complejo y poco intuitivo, por ello se recurre a la técnica de simplificación llamada *compresión de transiciones* que permite eliminar saltos de código redundantes.

Definición 4.4.9 *Sea* pp *una etiqueta del programa* p*, y consideremos un salto* goto pp*. La compresión de transiciones nos dice que podemos sustituir dicho* goto *copiando el bloque de código básico etiquetado por* pp*.*

De esta forma, al aplicar la técnica al resultado del ejercicio anterior obtendríamos el siguiente programa:

```
(search, (z, (x y z))): valuelist := tl (valuelist);
                        valuelist := tl (valuelist);
                        value := hd (valuelist);
```

Con esta técnica hemos obtenido un código más limpio y eficiente. Sin embargo, no debemos usar la técnica de forma arbitraria, ya que podemos tener un problema de excesivo código duplicado.

La técnica de compresión de saltos puede hacerse tanto una vez obtenido el código optimizado (como en el ejemplo anterior), como durante la generación del código. En este segundo caso tenemos la ventaja de que no tendremos que obtener el código (quizás demasiado extenso), sino que poco a poco vamos obteniendo un programa más compacto que el que resultaría al aplicar el algoritmo descrito anteriormente [JGS93].

4.5. Evaluación parcial *online* vs *offline*

En los ejemplos hemos visto que la evaluación parcial consistía en dos pasos fundamentales: primero se calcula la división de las variables para el programa, y a continuación se especializaba el programa, usando los valores concretos para las variables estáticas. A este método se le llama evaluación parcial *offline*.

En todo momento, un evaluador parcial deberá tomar ciertas decisiones, como qué valores de variables deben usarse para especializar, o qué transiciones deben ser objeto de la compresión de transiciones. Estas elecciones se realizan en función de una *estrategia*.

Definición 4.5.1 *Una estrategia es* online *si los valores específicos calculados durante la especialización de un programa pueden afectar la decisión sobre la acción que debe ejecutarse. Si no pueden afectar a la elección, entonces se dice que es una estrategia* offline.

Casi todas las estrategias usadas por evaluadores parciales son estrategias *offline*, basadas en un preproceso de los programas fuente. Pueden combinarse también técnicas *online* y *offline*. Debemos decir sin embargo, que la principal ventaja de la evaluación parcial *online* es que si es capaz de explotar la información extra en tiempo de especialización, puede obtener programas residuo mejores (más optimizados). La evaluación parcial *offline* por su parte, hace posible la generación de compiladores aplicándoselo a sí mismos.

4.6. Consideraciones finales

En esta sección vamos a ver una visión global de la técnica de evaluación parcial descrita en este tema. Sabemos que todo evaluador parcial necesita una serie de datos de entrada:

1. el programa fuente que queremos optimizar/especializar;
2. una división de las variables del sistema entre variables estáticas o dinámicas y,
3. los valores de las variables estáticas en función de los cuales se hará la especialización.

La salida de un evaluador parcial dadas las entradas anteriores será un programa residuo que, normalmente, será más eficiente que el programa original. El algoritmo usado para obtener dicho programa optimizado puede ser definido de distintas formas, tanto dentro de la opción *offline* como de las alternativas *online*. El descrito en este capítulo en primer lugar calcula el conjunto poly de puntos de programa especializados. A partir de dichos puntos, general el código a partir de cada bloque de código asociado a las etiquetas.

Una vez obtenido el programa resultante, aplica la técnica de compresión de transiciones, lo que dará como resultado un programa mucho más compacto y eficiente, aunque ya sabemos que se debe tener cuidado y no aplicar la compresión de forma arbitraria.

Por último, como los puntos especializados de programa son muy largos debido a cómo se definen (es necesario hacerlo así para poder automatizar el proceso), se cambian las etiquetas por números, lo que compacta aún más el código.

Vamos a ver ahora un posible escenario de aplicación de la evaluación parcial (se pueden encontrar los detalles en [JGS93]. Supongamos que queremos escribir un compilador para programas de la máquina de Turing. A nuestra disposición tenemos un intérprete para dicho lenguaje, y además tenemos un evaluador parcial para el lenguaje en el que está escrito el intérprete. En este ejemplo, el intérprete del lenguaje Turing está escrito en el lenguaje imperativo introducido en este capítulo.

Para obtener de forma completamente automática un compilador para el lenguaje de Turing, lo que podemos hacer es ejecutar nuestro evaluador parcial dándole como datos de entrada el mismo evaluador parcial (nótese que en este caso debe ser un evaluador parcial auto-aplicable, es decir, que el lenguaje de entrada coincida con el lenguaje en el que está escrito el evaluador parcial), y el intérprete del lenguaje de la máquina de Turing que teníamos escrito en el lenguaje imperativo sencillo. Tanto el intérprete como el evaluador parcial deben estar escritos en el lenguaje de entrada del evaluador parcial.

Según las proyecciones de Futamura, como resultado de la ejecución del evaluador parcial, obtendremos de forma completamente automática un compilador para el lenguaje de la máquina de Turing, ya que habremos especializado el evaluador en función del intérprete.

Bibliografía

[BCM+92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[Bei90] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold (NY), 2nd edition, 1990.

[BJNT00] A. Bouajjani, B. Jonsson, M.Ñilsson, and T. Touili. Regular Model Checking. In E. A. Emerson and A. P. Sistla, editors, *Proceedings of the 12th International Conference on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 403–418. Springer-Verlag, 2000.

[Bru91] M. Bruynooghe. A practical framework for the abstract interpretation of logic programs. *Journal on Logic Programming*, 10(2):91–124, 1991.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, New York, 1977. ACM Press.

[CC92] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13:103–179, 1992.

[CD93] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *ACM Symposium on Principles of Programming Languages*, pages 493–501, 1993.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8:244–263, 1986.

[CGL94] E. M. Clarke, O. Grumberg, and D. E. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16:1512–1542, 1994.

[CGP99] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, Cambridge, MA, 1999.

[CMCHG96] E. M. Clarke, K. L. Mcmillan, S. Campos, and V. Hartonas-GarmHausen. Symbolic Model Checking. In *Proceedings of the 8th International Conference on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 419–422. Springer-Verlag, July/August 1996.

[Cou] P. Cousot. Introduction to abstract interpretation. http://web.mit.edu/afs/athena.mit.edu/course/16/16.399/www.

[Cou99] J.-M. Couvreur. On-the-fly verification of linear temporal logic. In *Proceedings of the World Congress on Formal Methods in the Development of Computing Systems*, volume I of *Lecture Notes in Computer Science*, pages 253–271. Springer-Verlag, September 1999.

[Cou01] P. Cousot. Abstract Interpretation Based Formal Methods and Future Challenges. In R. Wilhelm, editor, *Informatics, 10 Years Back - 10 Years Ahead*, Lecture Notes in Computer Science, pages 138–156. 2001.

[Dam96] D. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands, 1996. PhD thesis.

[DGG97] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19(2):253–291, 1997.

[DO91] R.A. DeMillo and A.J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9), 1991.

[DT02] E. Díaz and J. Tuya. Comparación de técnicas metaheurísticas para la generación automática de casos de prueba que obtengan una cobertura de software. In *Actas de ADIS*, 2002.

[GL97] F. Glover and M. Laguna. *Tabu Search*. Kluwer, Norwell, MA, 1997.

[JGS93] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, jun 1993.

[JN95] N. D. Jones and F. Nielson, editors. *Abstract Interpretation: a Semantics Based Tool for Program Analysis*, volume 4 of *Handbook of Logic in Computer Science*, 1995.

[JSE96] B.F. Jones, H.-H. Sthamer, and D.E. Eyres. Automatic structural testing using genetic algorithms. *Software Engineering Journal*, 11(5):299–306, September 1996.

[KGV83] S. Kirkpatrick, C.D. Gellat, and M.P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, 1983.

[Kor90] B. Korel. Automated Software Test Data Generation. *IEEE Transactions on Software Engineering*, 16(8), August 1990.

[Mar93] K. Marriot. Frameworks for Abstract Interpretation. *Acta Informatica*, 30:103–129, 1993.

[Mic99] Z. Michalewicz. *Genetic Algorithms + Data Strictires = Evolution Programs*. Springer-Verlag, 1999.

[Mit96] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, 1996.

[MMS01] C.C. Michael, G. McGraw, and M.A. Schatz. Generating Software Test Data by Evolution. *IEEE Transactions on Software Entineering*, 27(12):1085–1110, December 2001.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems. Safety.* Springer-Verlag, Berlin, 1995.

[MS89] K. Marriot and H. Sondergaard. A Tutorial on Abstract Interpretation of Logic Programs. In *6th IEEE International Symposium on Logic Programming*, 1989.

[Mye83] Glenford J. Myers. *El Arte de Probar el Software.* El Ateneo (John Wiley & Sons, Inc. 1979), 1983.

[Myn90] B.T. Mynatt. *Software Engineering with Student Project Guidance.* Prentice-Hall International, 1990.

[Nta98] S. Ntafos. On Random and Partition Testing. In *Proceedings of the ISSTA 98*, volume 23 of *ACM SIGSOFT Software Engineering Notes*, pages 42–48. March 1998.

[PHP99] R.P. Pargas, M.J. Harrold, and R.R. Peck. Test-Data Generation using Genetic Algorithms. *The Journal of Software Testing, Verification and Reliability*, 9:263–282, 1999.

[Pre97] *Ingeniería del Software.* Pressman R., 1997.

[Sch01] L.M. Schmitt. Theory of Genetic Algorithms. *Theoretical Computer Science*, 259:1–61, 2001.

[Sch04] L.M. Schmitt. Theory of Genetic Algorithms II. *Theoretical Computer Science*, 310:181–231, 2004.

Glosario