Generación de trazas para la ejecución concólica de programas Erlang

Trabajo fin de Máster

Máster Universitario en Ingeniería del Software, Métodos Formales y Sistemas de Información (MUISMFSI)
DSIC, UPV
Curso 2014/2015

Autor:
Onofre Coll Ruiz

Tutor:

Germán F. Vidal Oriola

Co-tutor:

Salvador Tamarit Muñoz

Este documento contiene la memoria del trabajo fin de máster del Máster en Ingeniería del Software, Métodos Formales y Sistemas de Información cursado en el DSIC de la Universidad Politécnica de Valencia.

Índice

Introducción

El software ha dejado de estar presente en unos pocos sistemas y ordenadores, y ha pasado a estar en todas partes. Desde los PCs a los televisores, pasando por los teléfonos inteligentes e incluso cualquier objeto de uso común, la presencia de software en cada aspecto de la vida ha aumentado mucho en los últimos años.

Con dicho aumento también aumenta la necesidad de garantizar la calidad y fiabilidad del software. La gravedad de un error de software puede costar desde mucho tiempo de trabajo, a grandes cantidades de dinero, o incluso vidas humanas. Por ello, el esfuerzo de investigación en métodos para validar y certificar la corrección del software aumenta cada día más. Uno de esos métodos son las pruebas del software (*testing* en inglés).

El campo de las pruebas de software es tremendamente amplio y complejo, y las técnicas que pueden asegurar la fiabilidad del software suelen ser muy complejas y costosas de llevar a cabo. Existen diversos métodos para realizar las pruebas. Una de las propuestas más importantes en el ámbito de las pruebas *automáticas* del software es la ejecución simbólica.

En los últimos años, las técnicas de ejecución simbólica han despertado un renovado interés dadas sus aplicaciones a la verificación de programas, pruebas de software, depuración de código, etc. Su principio básico consiste en ejecutar el programa tratando los valores de entrada como variables *simbólicas* (es decir, como variables cuyo valor desconocemos). Típicamente, en cada sentencia de control se anotarán las condiciones que hacen que el flujo de control vaya por un camino u otro, y se seguirán todos los caminos posibles. Con ello se trata de cubrir todos los estados posibles (mientras los estados sean finitos, en caso contrario se requerirá algún tipo de abstracción o criterio de terminación).

En el ámbito de la generación de casos de prueba, una vez concluida la ejecución simbólica del programa, las condiciones de cada *hoja* del árbol de

ejecución simbólica se usan para generar datos concretos de entrada (empleando un resolutor de restricciones). Los datos de entrada generados para cada hoja garantizarán que la ejecución concreta asociada llegará a un estado equivalente, obteniendo así una cobertura óptima en muchos casos (y mucho mejor, en general, que la cobertura obtenida mediante casos de prueba aleatorios).

Sin embargo, este método se enfrenta a varios desafíos, entre ellos la explosión de estados en programas reales, la resolución de condiciones complejas o la aparición de falsos positivos como consecuencia de las técnicas de abstracción.

Con el objeto de evitar las debilidades des técnicas basadas en ejecución simbólica, surge la técnica conocida como ejecución concólica (*concolic execution* en inglés, cuyo origen es la combinación de las palabras *CONCrete* y *symbOLIC*). Esta técnica comienza con la ejecución del programa con datos de entrada aleatorios. Paralelamente a esta ejecución, se lanza una ejecución simbólica que, a diferencia de la técnica original, estará restringida al camino que siga la ejecución concreta, pero registrando las condiciones de cada elección condicional. A continuación, se elegirá una de esas condiciones (típicamente la última), se negará o variará de forma que su resolución de lugar a datos de entrada que sigan un camino diferente, con lo se reiniciará el proceso con estos datos.

Este trabajo fin de máster se centra en el contexto de la generación automática de casos de prueba para el lenguaje Erlang, concretamente en la fase de generación de trazas a partir de una ejecución concreta, con el objeto de guiar la ejecución concólica posterior.

En el capítulo 1 presentaremos una panorámica del estado del arte en los diferentes campos implicados en el desarrollo de este trabajo: la técnica de ejecución simbólica, la ejecución concólica, el lenguaje de programación Erlang y las diferentes técnicas y herramientas conocidas para las pruebas y la generación de pruebas (tracing en inglés) de programas Erlang.

En el capítulo 2 plantearemos formalmente nuestra propuesta de sistema de pruebas, incluyendo las definiciones formales, restricciones y asunciones realizadas para plantear la ejecución concólica.

En el capítulo 3 introduciremos el diseño de nuestro sistema, una visión de cómo debe funcionar y qué partes lo compondrán. Presentaremos también la implementación realizada, el módulo de *tracing* que recogerá información para posteriormente desarrollar el resto del sistema.

Por último, comentaremos las ampliaciones posibles, algunas ideas para continuar en esta línea de investigación y las conclusiones extraídas del trabajo realizado.

Capítulo 1 – Conceptos previos

En esta sección se realizará un repaso a los estudios anteriores que motivaron este trabajo y al estado del arte en ramas relacionadas para comprender mejor el desarrollo del mismo.

1.1 - Ejecución simbólica

La idea de la ejecución simbólica fue planteada en los años setenta por King [1]. En esa época se manejaba como alternativa para comprobar el funcionamiento de los programas las pruebas y el análisis formal. Se planteó la ejecución simbólica como otra opción.

En vez de ejecutar unos casos de muestra, el programa se ejecuta simbólicamente, cambiando las entradas concretas por variables, por lo que cada ejecución simbólica representa a un conjunto de ejecuciones concretas de la misma clase. Si el flujo de control del programa no dependiera de las entradas, una sola ejecución simbólica lo cubriría por completo. En caso contrario, habría que analizar los diferentes caminos. Pese a que la cantidad de caminos también puede ser poco manejable en casos reales, King afirma que promete mejores resultados que las pruebas aleatorias para la mayoría de programas.

Para describir la ejecución simbólica se recurrió a un modelo ideal, en el que los programas solo trabajan con enteros y no se producen desbordamientos por limitaciones de hardware, los árboles de ejecución son finitos y tenemos un probador de teoremas funcional.

Se ejecuta el programa operando sobre las variables según sea necesario. El caso más interesante sucede cuando se llega a una estructura de control *IF-THEN*. Si el camino a tomar depende de las variables de entrada, que en este

caso son simbólicas, se continuará por los dos caminos, anotando en la variable *PC*[1] la condición a cumplirse para entrar al camino en el que estemos.

A partir de los caminos y de las condiciones podríamos construir el árbol de ejecución dependiente de las entradas, y usar un probador de teoremas para probar la validez del programa.

[1] *PC* indica *path condition* o condición de camino.

1.2 - Ejecución concólica

La ejecución concólica fue propuesta como concepto por Godefroid [2]. Poco después ese mismo año surge la propuesta *CUTE,* de Sen [3], en la que surge la palabra “concolic” como unión de “concrete” y “symbolic”.

Como ejemplo de funcionamiento de la ejecución concólica describiremos el funcionamiento de *CUTE*. Para explorar los caminos de ejecución, *CUTE* instrumenta el código y construye un mapa de entrada lógica, que representa de forma simbólica un grafo de la memoria. Luego se ejecuta el código instrumentado de la siguiente forma:

1. Usando el mapa lógico se genera un grafo de memoria concreto y dos estados simbólicos, uno para punteros y otro para valores primitivos.
2. Ejecuta el código con el grafo concreto, acumulando las restricciones que caracterizan los caminos de ejecución.
3. Se niega una de las restricciones recopiladas y se resuelve para hallar un nuevo mapa lógico de entrada que llevará por un nuevo camino de ejecución para repetir el proceso.

En teoría la ejecución simbólica y concreta son simultáneas, aunque en la práctica no necesariamente es así.

El mapa lógico de entrada se crea para evitar problemas con la asignación dinámica de memoria en el programa, ya que la dirección concreta puede variar entre ejecuciones. Con él se puede representar los datos de entrada del comienzo de la ejecución evitando así depender de que funciones como *malloc* asignen la misma posición de memoria física en cada ejecución.

Las unidades a probar pueden tener muchas funciones, y hacer uso de bibliotecas o de llamadas a otras funciones. Se escogerá una función como entrada y para ella serán los datos de entrada de *CUTE*. Dichas unidades no recibirán entradas ni del usuario ni de lecturas en disco, todo será proporcionado en entrada por el sistema, aunque sí podrá asignar más memoria.

La instrumentación del código añadirá la información necesaria para realizar la ejecución simbólica. Dicha ejecución simbólica será ejecutada junto a la concreta, de forma que seguirá el camino de ésta. Además reemplazará por el valor concreto de la ejecución cualquier restricción que nuestro programa no sea capaz de manejar.

El programa instrumentado tendrá dos estados simbólicos, llamados A y P, en el que A mapea las posiciones de memoria a expresiones aritméticas simbólicas, y P a expresiones de puntero simbólicas. Las expresiones aritméticas serán lineares no constantes (serían derivables del estado concreto) y las de puntero son más sencillas: o una variable o la constante NULL.

Para explorar el árbol de ejecución se utiliza una estrategia acotada *depth-first*[2]. Cada ejecución se apoya en las condicionales de la anterior y se comprueba si se sigue el camino que se había predicho. Por culpa de algunas aproximaciones (generalmente el uso de valores concretos cuando los simbólicos no son manejables) puede que a veces no se siga el camino previsto, en cuyo caso se hace saltar una excepción y se reinicia con valores nuevos.

Para resolver las restricciones se usó *lp_solve* [4] con algunas modificaciones, principalmente optimizaciones para la resolución de las restricciones de camino que salen de la ejecución.

Para probar su efectividad y rendimiento se probó sobre las estructuras del propio entorno *CUTE* y sobre la biblioteca SGLIB [5], que proporciona implementaciones genéricas de los algoritmos más típicos de las listas, vectores y otras estructuras de datos comunes para el lenguaje C.

Pese a ser una biblioteca ampliamente usada y revisada el programa fue capaz de detectar 2 errores que permanecían ocultos hasta la fecha y de los cuales fue informado el equipo de desarrollo y fueron corregidos, ambos en

[2] *Depth-first* hace referencia una estrategia de exploración de árboles que empieza por el nivel más profundo.

ejecuciones del *CUTE* menores a un segundo, lo cual demuestra la potencia de la herramienta para encontrar errores.

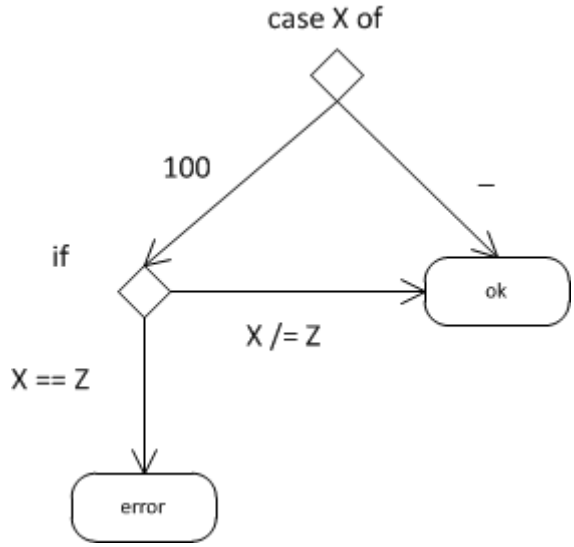
Ejemplo

Para ilustrar mejor la utilidad de la ejecución concólica veamos un pequeño ejemplo que la pondrá de manifiesto.

Consideremos el siguiente programa Erlang:

```
foo(X,Y) ->
      Z = Y*Y,
      case X of
            100 -> if
                  X == Z ->
                        something_causing_an_Error;
                  X /= Z ->
                        ok;
            _->
                  ok;
end.
```

Veamos su árbol de decisión para entenderlo mejor:



Para un programa tan sencillo como este, un sistema de testeo que generara valores de entrada aleatorios tendría problemas para encontrar la sentencia que produce el error, ya que por casualidad, entre todo el abanico de valores concretos que pueden tomar X e Y, tendría que acertar y generar una entrada con valor (100,10).

Sin embargo, con el método de la ejecución concólica el error sería encontrado rápidamente.

En primer lugar, se lanzaría una ejecución con un valor de entrada cualquiera, pongamos (0,0). Realizaríamos una ejecución concreta obteniendo la salida *ok*, y una traza que recoja el camino seguido. Forzaríamos a la ejecución simbólica a seguir el mismo camino, e ir recogiendo las restricciones que debe cumplir la variable de entrada para seguirlo. En este caso, para ir por el último camino, la ejecución simbólica tendría que recoger la restricción de éste (ninguna al haber entrado por la cláusula _), y la negación de todas las anteriores puesto que si cumpliera alguna hubiera encajado en ella por estar antes dentro de la estructura *case*, así que la única restricción sería `(X /= 100)`.

Con esa restricción, le pediríamos a nuestro sistema de solución de restricciones que nos proporcionara un valor que negara una de ellas (como

solo tenemos una elegiríamos esa) y que nos proporcionara un valor concreto para ella.

Como se ve trivialmente, obtendríamos el valor 100. Al no haber recogido restricciones sobre Y, volveríamos a ejecutar esta vez con valores (100, 0).

Esta ejecución nos devolvería de nuevo *ok*, al entrar por la segunda cláusula del *if*, y recogeríamos las restricciones `(X == 100 && Z /= X)`. Negando la última para buscar un nuevo camino tendríamos la condición `(X == 100 && Z == X)`, y a partir de ella obtendríamos los valores de entrada (100, 10) resolviendo que `Z = Y*Y`. Con estos valores de entrada conseguiríamos conducir al programa por la sentencia que produce un error, y así encontraríamos en solo 3 iteraciones un error que llevaría mucho tiempo encontrar utilizando metodologías de generación de casos de test aleatorias.

Problemas habituales

Por supuesto, no es un método perfecto. Junto a sus grandes ventajas encontramos algunos inconvenientes que limitan su efectividad.

El primero es la explosión de estados. Pese a que este problema se ve reducido en gran parte frente a la idea de la que partimos (la ejecución simbólica), programas reales pueden generar trazas de tamaños y complejidades no manejables, complicando y reduciendo la utilidad de la técnica. Pese a que se puede atacar el problema mediante la aplicación de heurísticas y simplificaciones, a día de hoy sigue siendo un problema a tener en cuenta.

El segundo gran problema es la resolución de restricciones. Para programas reales, las condiciones de camino acumuladas por la ejecución simbólica pueden alcanzar grandes complejidades, que el sistema de resolución puede no ser capaz de manejar en un tiempo razonable (o incluso ser totalmente incapaz). Por lo tanto, mejorar los sistemas de resolución de restricciones y mantener los utilizados en este tipo de sistemas de pruebas actualizados es una prioridad en el campo.

Por último, la propia naturaleza del lenguaje Erlang nos añade otra dificultad. En ocasiones, ciertos programas pueden tener comportamientos no deterministas que hagan que nuestras condiciones, que en principio deberían guiarnos por caminos aún por explorar, no lo hagan, o acabemos en un camino distinto al que esperábamos.

1.3 - Erlang – Historia y características

Presentaremos a continuación un breve resumen de la historia y las características principales de Erlang [6] según Cesarini [7].

Erlang surge en el seno de la compañía Ericsson a finales de los ochenta, como solución a la ausencia de un lenguaje que cumpliera todas las características deseables para su negocio. Recoge influencia de una gran variedad de lenguajes, entre ellos funcionales como *ML* y *Miranda*, concurrentes como *ADA*, *Modula* y *Chill* y también del mundo de los lenguajes lógicos, esencialmente *Prolog*. Se nota la herencia de dos lenguajes propietarios de Ericsson como *EriPascal* y *PLEX*, y las capacidades de actualización presentes en *Smalltalk*.

A mediados de los noventa, tras confirmar la madurez del lenguaje para grandes proyectos, surge *OTP*, un *framework* para estructurar sistemas Erlang, dotándolos de diversas bibliotecas así como de robustez y tolerancia a fallos. Como es un lenguaje creado para atacar el problema de sistemas distribuidos, tolerantes a fallos, concurrentes a gran escala y de tiempo real "suave"[3]; se encontró que esas características eran deseables no solo para el mundo de las telecomunicaciones donde se había creado, sino para servicios web, mensajerías, integración en empresas…

Ericsson liberó el lenguaje en 1998. Su popularidad fue en aumento desde entonces y en la actualidad cuenta con una creciente comunidad de desarrolladores. Veremos ahora sus principales características.

Constructos de alto nivel

Como lenguaje declarativo, se centra en describir qué debe hacerse más que el cómo. Por tanto, las funciones se leen como conjuntos de ecuaciones, y contienen habitualmente sentencias de *pattern matching* para elegir entre

[3] Usamos la palabra "suave" al hacer referencia a los *soft real-time systems*, sistemas a tiempo real que ven su utilidad perjudicada si incumplen los tiempos y por tanto se ve perjudicada la calidad del servicio sin llegar a ser considerado un error crítico del sistema.

distintos casos, extraer datos de estructuras complejas o para condiciones implícitas.

Procesos concurrentes y paso de mensajes

Erlang es un lenguaje en el que la concurrencia es fundamental. Los procesos no comparten espacio de memoria, sino que cada cual tiene su propio espacio, montículo y pila. Esto produce ciertas ventajas a la hora de evitar problemas típicos como las condiciones de carrera.

Los procesos, por tanto, se comunican mediante el paso de mensajes (que pueden ser de cualquier tipo de dato válido) de forma asíncrona. Los mensajes se recuperan desde el buzón de forma selectiva (no tienen por qué ser procesados por orden de llegada) lo que mejora la robustez de la concurrencia.

Concurrencia eficiente, segura y escalable

Los procesos se gestionan en la máquina virtual de Erlang y no tienen correspondencia directa en el sistema operativo que haya por debajo, por lo que se manejan de forma muy eficiente en entornos diversos, haciendo la concurrencia ligera y escalable.

Esto también mejora el tiempo de paso de mensajes, ya que, mientras que en otros lenguajes implica complicadas secuencias de memorias compartidas, semáforos e hilos, en Erlang solo requiere copiar dicho bloque de memoria al buzón del proceso de destino.

Al añadirle a todo esto la automatización del almacenamiento y un sistema de recolección de basura por proceso, se mantienen tiempos de respuesta aptos incluso para sistemas de tiempo real “suaves”, capaces de soportar grandes picos de carga.

Robustez

Las bibliotecas que forman parte de los módulos del lenguaje se centran en mantener la robustez por encima de todo. Si programamos solo para el caso

correcto y cedemos el manejo de excepciones y tratamiento de errores a los mecanismos presentes en el sistema, estaremos construyendo programas más sencillos y, probablemente, con menos errores.

Dichos mecanismos están basados en enlaces entre procesos, lo que los mantiene informados entre ellos del estado del resto. Esto permite crear supervisores que monitorizan el estado de los procesos y actúan en consecuencia. Además, ciertos comportamientos genéricos están implementados en el *framework* OTP.

Computación distribuida

La sintaxis y semántica del lenguaje llevan la distribución incorporada, permitiendo la intercomunicación entre nodos por cualquier red mediante el protocolo TCP/IP de forma cómoda y sencilla.

Como los nodos y clústers están diseñados para ser ejecutados bajo la protección de un cortafuego, la seguridad se basa en *cookies* secretas con muy pocas restricciones en cuanto a derechos de acceso.

Integración

Erlang no siempre será el lenguaje perfecto para parte del trabajo. Por este motivo, es capaz de comunicarse con muchos otros lenguajes mediante bibliotecas de alto nivel, que permitirán usar nodos en otros lenguajes como si se trataran de nodos en Erlang, o con controladores o sockets usando protocolos típicos como HTTP, SNMP e IIOP.

Uso en el mundo real

Pese a ser un lenguaje relativamente desconocido, es ampliamente usado en diversas áreas y por diversas empresas de gran tamaño para aplicaciones que requieran de sus características.

Entre los ejemplos más destacados tendríamos:

- Yaws [8] (servidor web).
- Amazon SimpleDB [9] (base de datos).

- ejabberd [10](mensajería instántanea basada en XMPP usado por el chat de Facebook y Tuenti).
- WhatsApp [11] en sus servidores de mensajes.
- Rakuten lo usa en su sistema de archivos distribuido [12].
- Ericsson lo usa en nodos de soporte, GPRS y 3G por todo el mundo [13].
- Aparece en servidores de varios juegos, como en el motor Naos [14] del Vendetta Online [15], y en otros títulos que se apoyan en Demonware [16] como Call of Duty y otros títulos de Activision/Blizzard [17] .

1.4 - Erlang – Sintaxis básica

Sin entrar en detalle, puesto que daría para escribir libros al completo, daremos un pequeño repaso a la sintaxis más básica del lenguaje, para facilitar la comprensión del resto del trabajo a cualquier lector familiar con la programación, pero poco familiarizado con el lenguaje Erlang. Para este pequeño repaso nos basaremos en el tutorial de Trottier-Hebbert [18].

Generalmente utilizaremos el intérprete Erlang, que nos permitirá tanto escribir comandos directamente como compilar módulos y sistemas escritos en archivos fuente.

Como parte de la herencia de lenguajes lógicos (principalmente Prolog), las expresiones en Erlang deben terminar en punto para que sean ejecutadas. Podemos separar expresiones mediante comas dentro de un mismo bloque o función, pero la sintaxis siempre requerirá un punto al final.

Podremos realizar operaciones aritméticas básicas de forma intuitiva:

```
1> 2 + 15.
17
2> 49 * 100.
4900
3> 1892 - 1472.
420
```

Como en casi cualquier lenguaje funcional, las variables no pueden variar una vez asignadas. Deben comenzar en letra mayúscula o en el carácter '_'. Las variables que comiencen por dicho carácter serán consideradas "a ignorar", y se usarán para indicar valores que no importan cuales sean para la funcionalidad del programa. De hecho, para la variable '_', ni siquiera se almacenará su valor. Tendrán su utilidad en algunas situaciones en las que, por ejemplo, nos importen ciertos valores pero otros nos sean indiferentes al hacer *pattern matching*. En el siguiente código de ejemplo vemos estas dos peculiaridades. En la instrucción 6 vemos como falla el intento de guardar en

una variable un dato diferente al que ya le fue asignado. En las instrucciones 7 y 8 se aprecia que aunque asignemos un valor a '_', en la siguiente instrucción comprobamos que no tiene nada almacenado.

```
1> One.
* 1: variable 'One' no está asignada
2> One = 1.
1
3> Un = Uno = One = 1.
1
4> Two = One + One.
2
5> Two = 2.
2
6> Two = Two + 1.
** exception error: no match of right hand side value 3
7> _ = 14+3.
17
8> _.
* 1: variable '_' is unbound
```

Las variables deben empezar en mayúscula porque las palabras escritas en minúscula tienen otro significado. Son átomos. Tienen sentido en sí mismos, funcionando como una especie de constante con el mismo valor que nombre, útiles, por ejemplo, como valores de control. Por lo general no pueden contener espacios aunque cualquier cadena escrita entre comillas simples será entendida como átomo.

```
1> atom.
atom
```

Veamos ahora el álgebra booleana y las operaciones de comparación. Los booleanos *and*, *or*, *xor* y *not* funcionan de manera sencilla y previsible. Los operadores de comparación también, aunque tienen un pequeño detalle. La

comparación de igualdad o desigualdad puede tener en cuenta el tipado de los elementos o no, dependiendo del operador utilizado. En los ejemplos 8 y 9 queda ilustrada la diferencia entre =:= y ==. El primero de los operadores comprobará la igualdad de tipos mientras que el segundo no lo hará.

```
1> true and false.
false
2> false or true.
true
3> true xor false.
true
4> not false.
true
5> 5 =:= 5.
true
6> 1 =:= 0.
false
7> 1 =/= 0.
true
8> 5 =:= 5.0.
false
9> 5 == 5.0.
true
10> 5 /= 5.0.
false
```

Por supuesto, también contamos con los operadores <, >, >= y =<.

Veamos el funcionamiento de las tuplas. Sirven para organizar los datos, son la estructura de datos más básica del lenguaje. Podremos usarlas para agrupar un número previamente determinado de términos cualesquiera. Por ejemplo, para representar un punto:

```
1> Point = {10,4}.
```

```
{10,4}
```

Las veremos en muchas ocasiones combinadas con un átomo, que hará las veces de nombre de la estructura, útiles por tanto incluso para datos individuales. Así, de forma general, tendremos tuplas de este estilo:

```
3> PreciseTemperature = {celsius, 23.213}.
{celsius,23.213}
4> {point, {4,5}}.
{point,{4,5}}
```

Además, continuando en esa línea aparecen los *records*, una forma más agradable de ver las tuplas y crear estructuras de datos. Declaramos un *record* de nombre *person* con los campos *name, phone, address*. Los valores asignados a *name* y *phone* en la declaración serán valores por defecto. Veamos cómo usar a nuestra *person*. Podemos apreciar como internamente es una tupla.

```
1> rd(person, {name = "", phone = [], address}).
person
2> #person{phone=[0,8,2,3,4,3,1,2], name="Robert"}.
#person{name = "Robert",
        phone = [0,8,2,3,4,3,1,2],
        address = undefined}
```

Veamos ahora las listas. Como en la mayoría de lenguajes funcionales, las listas son la estructura más útil y usada, gracias al tipado de Erlang. Pueden contener cualquier tipo y cantidad de elementos combinándolos de cualquier manera, por ejemplo:

```
1> [1, 2, 3, {numbers,[4,5,6]}, 5.34, atom].
[1,2,3,{numbers,[4,5,6]},5.34,atom]
```

Para el manejo de listas podemos usar los operadores ++ y --, que añaden o eliminan elementos de una lista. Contamos además, con la posibilidad de

extraer cabeza y cola, tanto con las funciones *hd* y *tl* como con el constructor de listas '[H | T]'.

```
1> hd([1,2,3,4]).

1

2> tl([1,2,3,4]).

[2,3,4]

3> List = [2,3,4].

[2,3,4]

4> NewList = [1|List].

[1,2,3,4]

5> [1,2,3,4] = [X|_].

6> X.

1.
```

Además del habitual manejo de las listas como cabeza y cola, tenemos otra herramienta muy útil para manipular y construir listas, conocida como listas intensionales. Similarmente a como se plantearía matemáticamente, podemos definir expresiones de la forma:

```
1> [2*N || N <- [1,2,3,4]].

[2,4,6,8]

2> [X || X <- [1,2,3,4,5,6,7,8,9,10], X rem 2 =:= 0].

[2,4,6,8,10]

3> Weather = [{toronto, rain},{montreal, storms},{london, fog},{paris,
 sun}, {boston, fog}, {vancouver, snow}].

[{toronto,rain},

{montreal,storms},

{london,fog},

{paris,sun},

{boston,fog},

{vancouver,snow}]

4> FoggyPlaces = [X || {X, fog} <- Weather].

[london,boston]
```

A la izquierda de '||' pondremos qué obtendremos como resultado, a la derecha la variable a la que iremos metiendo cada elemento de la lista inicial, que estará tras el símbolo ' <- '. Como se ve en el ejemplo del clima justo arriba, podemos aplicar patrones y condiciones a la lista de forma que filtremos los elementos que nos interesan, en este caso la variable X encajará con cualquier ciudad, pero al poner *fog* al lado solo encajarán las tuplas en las que el segundo elemento sea ése, y nos devolverá una lista con las dos X que cumplen dicha condición.

Normalmente querremos definir un conjunto de funcionalidades asociadas en un mismo archivo, que en el caso de Erlang son conocidos como módulos. Para crear un módulo no tenemos más que empezar el archivo "*.erl" con la línea:

```
-module(name).
```

Donde en *name* pondremos el nombre del módulo, que debe ser el mismo que el del archivo sin la extensión, en forma de átomo. Aparte de *module*, los otros dos atributos de cabecera más comunes son *export* e *import*, que permitirán definir en una lista qué funciones del módulo serán usables desde otros módulos (o el intérprete), y qué funciones de otros módulos queremos usar desde este (se podrán llamar igualmente sin importarlas pero en ese caso deberemos poner el nombre del módulo delante en la forma *module:function()*). Dichos nombres de función irán acompañados siempre de la aridad de la misma.

Como ejemplo, declararemos que exportamos la función *foo*, que tiene 2 argumentos de entrada, e importaremos la función *format*, del módulo *io*, que imprimirá por consola su argumento.

```
-export([foo/2]).

-import(io, [format/1]).
```

Dentro de un módulo estructuraremos la funcionalidad en funciones. Las funciones carecen de sentencias estilo *return* como otros lenguajes, por lo que el resultado de la función es el resultado de la última expresión. En el ejemplo

vemos una función que recibe dos parámetros, *A* y *B*, imprime en la salida estándar *"Sumando…"* y hace la suma *A*+*B*. Dicha suma será el resultado de la función al ser el resultado de la última expresión.

```
foo(A,B) ->
      io:format("Sumando…"),
      A + B.
```

Vamos a ver qué cosas podemos hacer con las funciones. Entre las características más interesantes está el *pattern matching*, que nos permitirá definir funciones de diferente forma según encajen los valores de entrada, separando cada cláusula con punto y coma hasta la última, que acabará con punto. Así, podemos escribir cosas con este aspecto:

```
greet(male, Name) ->
      io:format("Hello, Mr. ~s!", [Name]),
      ok;
greet(female, Name) ->
      io:format("Hello, Mrs. ~s!", [Name]),
      ok;
greet(_, Name) ->
      io:format("Hello, ~s!", [Name]),
      ok.
```

Gracias a esta función saludaremos de forma distinta a un hombre o a una mujer, según el primer argumento de la función sea *male* o *female*. Incluso para valores cualesquiera también definimos otro comportamiento en la tercera cláusula. El segundo valor de entrada, al estar ligado a una variable hasta ese momento libre, encajará siempre y no influirá sobre la cláusula por la que entrará la función. De esta forma no se consigue nada que no se pudiera hacer con, por ejemplo, estructuras *if-then-else* sobre los valores de entrada, pero es un estilo mucho más limpio y declarativo.

Solo con *pattern matching,* pese a ser una herramienta muy potente, no podemos refinar todo lo necesario para algunos usos, y ahí entran las guardas.

Las guardas son expresiones que debemos cumplir para ejecutar una cláusula, y por tanto son expresiones evaluables a *true* o *false*. Se pueden poner varias si es necesario, escribiéndolas tras un *when* después de cerrar el paréntesis de los valores de entrada, y separando las diferentes guardas por comas si son *and*, y por punto y coma si son *or*. Veamos un pequeño ejemplo para ver si una persona puede subir a una atracción de un parque de atracciones:

```
tall_enough(merry_go_round, X) when X>=100, X=<190 -> can_ride;
tall_enough(merry_go_round,_) -> cannot_ride;
tall_enough(rollercoaster, X) when X>=140, X=<210 -> can_ride;
tall_enough(rollercoaster,_) -> cannot_ride;
```

Utilizaremos *pattern matching* para ver, en el primer valor de entrada, que atracción estamos mirando, representada por átomos. Luego se enlazará el valor a X, que será evaluada en las guardas. Si las cumple, diremos que puede subir. Si no las cumple, se intentará con la siguiente cláusula, que cumplirá siempre y le dirá que no puede subir.

También podemos encontrar sentencias *case*. Al igual que los *if* son similares a las guardas, los *case* funcionan de forma similar al *pattern matching* en la cabecera de la función. Siguiendo el ejemplo anterior de emparejar la temperatura con la unidad en la que está, vemos un ejemplo del *case*. Se ve rápidamente la equivalencia con poner los *case* como diferentes casos de *matching* de la variable *Temperature*.

```
case Temperature of
      {celsius, N} when N >= 20, N =< 45 ->
            'favorable';
      {kelvin, N} when N >= 293, N =< 318 ->
            'scientifically favorable';
      {fahrenheit, N} when N >= 68, N =< 113 ->
            'favorable in the US';
      _ ->
```

```
            'avoid beach'
end.
```

Erlang también cuenta con expresiones *if*, pero son bastante distintas a la mayoría de expresiones *if* en otros lenguajes. Dado su funcionamiento, en muchos casos tiene más utilidad utilizar guardas en los *case*, ya que funcionan de forma muy similar.

```
if X > Y ->
      a();
      X < Y -> b();
      true  -> c()
end.
```

Viendo este ejemplo queda ilustrada la peculiaridad de los *if* en Erlang. La última guarda, *true*, sería similar a un *else* tradicional, entrando por esa rama si no se cumple ninguna de las guardias anteriores.

Para acabar esta introducción a Erlang, veremos las sentencias y funciones que nos permiten crear procesos e interactuar con ellos, creando así el sistema de procesos concurrentes de Erlang.

Un proceso tan solo requiere una función para ejecutarse, dicha función será ejecutada en el proceso, y cuando acabe, el proceso desaparecerá. Para crear un proceso tenemos que utilizar la función *spawn*. Como dicha función nos devolverá un identificador del proceso creado (conocido como Pid por sus siglas inglés). Los identificadores de procesos son un tipo de datos especial y suelen representarse como un grupo de tres enteros de la forma `<0.23.12>`. Es bastante habitual usarlo de forma similar a:

```
Pid = spawn(modulename, functionname, [list_of_arguments]).
Pid = spawn(fun()->... end).
```

En el segundo caso estaríamos creando un proceso nuevo de una función que declaramos en ese mismo momento usando *fun*.

Si tenemos un proceso importante al que queremos otorgar un nombre fijo más allá de su identificador de proceso para poder ser usado desde cualquier parte del sistema, podemos usar la función *register*, que asociará el identificador de proceso a un átomo a nuestra elección.

```
register(some_name, Pid).
```

Los procesos están aislados entre sí salvo por el envío y recepción de mensajes. Enviar mensajes es muy sencillo. Solo necesitamos conocer el identificador de proceso al que queremos enviar el mensaje y usar el operador '!'.

```
Pid ! Message.
```

El *Message* puede ser cualquier cosa. Es bastante habitual enviar mensajes estructurados en tuplas o listas. Cuando enviamos un mensaje a un proceso este quedará almacenado en su buzón, hasta que dicho proceso decida leerlos.

Para leer los mensajes se utiliza la sentencia *receive*. Su comportamiento es muy parecido al de un *case*, pero depende del mensaje recibido en vez de depender del patrón que pongamos para cada *case*. Al llegar al *receive*, intentaremos encajar el primer mensaje, por orden, con cada patrón del mismo, y ejecutaremos el código de dentro si se produce el *matching*. Si el mensaje no encajara en ningún patrón, se cogería el segundo de la cola si lo hubiese. En caso de que ningún mensaje pueda ser procesado, el proceso se suspende. Los mensajes no procesados permanecen en cola, mientras que los que sí encajan son sacados del buzón.

En el siguiente ejemplo vemos un caso de proceso dedicado a escuchar mensajes, en el que si recibimos alguno de los mensajes esperados, responde al proceso que se lo envió, e incluimos un patrón comodín para capturar cualquier mensaje inesperado e informar de que se recibió. Después de responder adecuadamente a cada mensaje, se vuelve a llamar a la función para esperar al siguiente mensaje.

```
listener()->
      receive
            {From, pattern1} ->
                  From ! answer1,
                  listener();
            {From, pattern2} ->
                  From ! answer2,
                  listener();
            _ ->
                  io:format("Unknown message"),
                  listener()
      end.
```

Ejemplo

Para hacernos una idea del aspecto de un programa Erlang y de la sencillez con la que se realizan las tareas más básicas, veremos un pequeño ejemplo de programa que ilustra sus características principales: *pattern matching*, facilidad para la creación de procesos y paso de mensajes.

```
-module(foo).
-export([start/1, listen/1]).
start(N)->
      Listen = spawn(foo, listen, N),
      loop(Listen, 1).
loop(Listen, N) ->
      Listen ! {N, self()},
      receive
            ok-> ok;
            no -> loop(Listen, N+1)
      end.
listen(N) ->
      receive
```

```
            {N, From} -> From ! ok;
            {_, From} -> From ! no, listen(N)
    end.
```

En este ejemplo el programa consiste en una función *start*, que crea un proceso *listen*, del que guardará su identificador en la variable *Listen* y ejecuta la función *loop*. El proceso *listen* quedará a la espera de mensajes, y cuando los reciba contestará *ok* o *no* dependiendo si recibe la N con la que fue creado o cualquier otra cosa, parándose en el caso del *ok* o volviendo a llamar así mismo en el otro.

Loop, por su parte, enviará un mensaje a *listen*, empezando por el 1, y aguardará su respuesta. Continuará enviando mensajes incrementando el número hasta que la respuesta recibida sea *ok*. En dicho mensaje incluirá su identificador de proceso, obtenido mediante la llamada a *self/0* para que *listen* pueda contestarle.

1.5 - Erlang AST

Tras haber visto algo de historia y características principales de forma general, vamos a ver algunas de las construcciones básicas que serán más relevantes en este trabajo y su equivalencia en lo que llamamos Árbol de sintaxis abstracta[4].

El Árbol de sintaxis abstracta [19] no es más que otra forma de representar el código fuente. Es una estructura arborescente generada por el compilador Erlang previa a convertir el código en ejecutable.

La importancia del mismo reside en que, en ese punto de la conversión, Erlang nos permite manipular el AST, siendo una forma apta para realizar la instrumentación del código.

El código en forma de AST, como su nombre indica, puede verse como un árbol. Está formado por listas (entre [ ]) y tuplas (entre { }) anidadas de forma que representan el código de forma similar al fuente.

El programa entero estará dentro de una lista, dentro de la cual tendremos tuplas para cada instrucción, término, variable, función… Éstas últimas tendrán entre los valores de su tupla una lista más con otras tuplas dentro, esto es, el contenido de dicha función.

Veamos la representación de algunos de los elementos más habituales del lenguaje en forma de árbol de sintaxis abstracta. No se pretende cubrir toda la sintaxis y semántica del lenguaje Erlang porque sería poco práctico y útil, pretendiendo simplemente ilustrar el estilo que adquiere el código al pasarlo a AST, y centrando los ejemplos planteados en los casos más útiles para las manipulaciones que se verán durante este trabajo.

En el AST se anota la línea en la que se encuentra la parte del código que representa cada tupla, por lo que para ver las representaciones de forma no concreta utilizaremos la palabra LINE cuando deba escribirse la línea donde

[4] Arbol de sintaxis abstracta del inglés *Abstract Syntax Tree*, abreviado como AST.

se encuentra. Del mismo modo usaremos nombres genéricos de funciones, módulos y variables. Para ilustrar los operadores hemos elegido el operador de envío de mensajes '!' y *spawn* como ejemplo de llamada a función.

Erlang	**AST**
(Sin equivalencia, nombre del fichero)	`[{attribute,LINE,file,{"../src/modulename.erl",1}},`
`-module (modulename).`	`{attribute,LINE,module,modulename},`
`-export ([function1/1, function2/0]).`	`{attribute,LINE,export,[{function1,1},{function2,0}]} ,`
`function1 (Parameter) ->` contenido de la función	`{function,LINE,function1,Num_Parameters, [{clause,LINE, [{var,LINE,'Parameter'}], [], [`contenido de la función `]}`
`receive {hello,From} -> {ok, From}; Msg -> {unknown, Msg} end.`	`{'receive',LINE, [{clause,LINE, [{tuple,LINE, [{atom,LINE,hello}, {var,LINE,'From'}] }], [], [{tuple,LINE, [{atom,LINE,ok}, {var,LINE,'From'}] }] }, {clause,LINE, [{var,LINE,'Msg'}], [], [{tuple,LINE, [{atom,LINE,unknown}, {var,LINE,'Msg'}] }] }] }`

`Pid ! hello`	`{op,LINE,'!',`
	`{var,LINE,'Pid'},`
	`{atom,LINE,hello}}`

`case Variable of`
`ok->`
`Res;`
`_Err->`
`error`
`end`
	`{'case',LINE,`
	`{var,LINE,'Variable'},`
	`[{clause,LINE,`
	`[{atom,LINE,ok}],`
	`[],`
	`[{var,LINE,'Res'}]},`
	`{clause,LINE,`
	`[{var,LINE,'_Err'}],`
	`[],`
	`[{atom,LINE,error}]}`
	`]`
	`}`

`X = 1`	`{match,LINE,{var,LINE,'X'},{Integer,LINE,1}}`

`spawn (modulename, function1, [Parameter1]),`	`{call,LINE,`
	`{atom,LINE,spawn},`
	`[{atom,LINE,modulename},`
	`{atom,LINE,function1},`
	`{cons,LINE,{var,LINE,'Parameter1'},{nil,LINE}}]}`

Sin equivalencia, fin del archivo	`{eof,LINE}]`

1.6 - Criterios de cobertura para Erlang

La idea central de la cobertura es que cada trozo del programa debe ser utilizado durante las pruebas para confirmar su corrección. De esta sencilla afirmación surgen diversas formas de ver la cobertura, según nos presenta Taylor [20], dependiendo de lo que entendamos por trozo.

Si representamos el programa como un árbol en el que los nodos son los puntos en los que se toman decisiones (incluyendo ejecutar una o más veces un bucle, o no entrar en absoluto) y los caminos entre nodos son el resto de bloques, podemos hablar de *cobertura de rama* y considerarla completa si ejecuta cada camino y llega a todas las hojas de dicho árbol. Así garantizamos que todas las decisiones de cada punto de decisión han sido probadas.

También se puede tener otras medidas directas sobre el código fuente. La aproximación más directa es la *cobertura de sentencias*, que requiere que cada una de ellas sea ejecutada al menos una vez. El sentido exacto de sentencia varía entre lenguajes, y además es posible ejecutar todas las sentencias sin alcanzar el 100% de cobertura de rama.

Similar a la de rama tenemos la *cobertura de decisión*, excepto que se basa en las decisiones explícitas del código fuente. Esto podría hacernos fijarnos en decisiones que jamás se toman porque los caminos previos para llegar a ellas imposibilitarían dicho suceso. Esto no pasa con el criterio tradicional de cobertura de rama. Sin embargo, puede saltarse ciertos caminos implícitos que sí cubriría la cobertura de rama, como el efecto de no entrar por ninguno de los casos de un *case*.

La *cobertura de condición* dice que cada componente que participa en una decisión debe ser tanto verdadero como falso en algún lugar del conjunto de test para probar que funciona correctamente, puesto que para decisiones complejas puede que solo tuviéramos que probar uno de los dos estados. Esto puede lograrse sin incrementar de manera muy significativa el número de

casos. En ocasiones se combina con el de decisión para un resultado más completo.

Modificándolo un poco, tenemos el criterio de cobertura *modificado de condición/decisión*, cuya definición dice que cada condición debe afectar de forma independiente a la decisión. Con este requisito aseguramos que el efecto de cada condición se prueba con relación al resto de condiciones. Con esto reducimos un poco la cantidad de test a realizar. Este criterio es estándar y requerimiento en diversos entornos donde la seguridad es crítica, como por ejemplo en el mundo de la aviación. En otros no es mencionado explícitamente pero es una buena forma de medir la cobertura en entornos críticos y en algunos no tan críticos.

Existen otras medidas, como la presentada en el artículo *Data Flow Coverage for Testing Erlang Programs* [21], basadas en comprobar todas los pares y cadenas de pares de instrucciones que actúan sobre las mismas variables.

Cover tool

Dentro del *framework* OTP viene incluida una herramienta asociada al testing, *Common Test Environment*, que a su vez tiene un componente encargado de medir la cobertura de los test que realiza, conocida como *Cover tool* [22].

La herramienta se configura mediante un fichero de especificación, que indicará qué módulos deberán ser incluidos o excluidos del análisis. En dicha especificación también se puede incluir el nivel de cobertura, de más detallado a más general.

El fichero de configuración consta de las siguientes opciones.

```
%% List of Nodes on which cover will be active during test.

      %% Nodes = [atom()] {nodes, Nodes}.

      %% Files with previously exported cover data to include in
analysis.

      %% CoverDataFiles = [string()]{import, CoverDataFiles}.

      %% Cover data file to export from this session.
```

```
      %% CoverDataFile = string() {export, CoverDataFile}.

      %% Cover analysis level.

      %% Level = details | overview {level, Level}.

      %% Directories to include in cover.

      %% Dirs = [string()] {incl_dirs, Dirs}.

      %% Directories, including subdirectories, to
include.{incl_dirs_r, Dirs}.

      %% Specific modules to include in cover.

      %% Mods = [atom()] {incl_mods, Mods}.

      %% Directories to exclude in cover. {excl_dirs, Dirs}.

      %% Directories, including subdirectories, to
exclude.{excl_dirs_r, Dirs}.

      %% Specific modules to exclude in cover.{excl_mods, Mods}.

      %% Cross cover compilation

      %% Tag = atom(), an identifier for a test run

      %% Mod = [atom()], modules to compile for accumulated analysis

      {cross,[{Tag,Mods}]}.
```

Entre las funcionalidades más interesantes tenemos la posibilidad de exportar e importar datos sobre cobertura realizados anteriormente, de forma que evitaremos tener que realizar todos los test de una sola vez para analizar su cobertura como conjunto de test pudiendo añadir poco a poco la cobertura de cada uno de los test ejecutados.

Tras acabar los tests, la versión especialmente compilada para analizar su cobertura será parada y eliminada del sistema de forma automática, aunque se puede desactivar si nos interesa.

Toda la información relativa a la cobertura se almacena y puede ser consultada desde los resultados HTML del test o conjunto de test.

1.7 - Pruebas en Erlang

En Erlang existe una buena variedad de aplicaciones y herramientas útiles para el desarrollo y pruebas de las aplicaciones. Pese a que todo el mundo hace pruebas de una forma u otra, no existe una gran aplicación que sea gratuita, flexible, estable, y usada de forma generalizada por los desarrolladores. Por tanto, en muchos casos se recurre a herramientas propietarias o a funciones de prueba escritas manualmente. Veamos algunos ejemplos del estudio de Nagy [23].

EUnit

EUnit [24] es un servidor de pruebas ligero y de código abierto para Erlang. Intenta llevar ideas de otros entornos de pruebas unitarias a Erlang, para una rápida generación de casos de prueba, ciclos rápidos y resultados entendibles.

En las pruebas unitarias se ponen a prueba partes del programa por separado, que pueden ser funciones, módulos, procesos o incluso aplicaciones enteras. No solo se pueden probar con aserciones, sino que incluye un generador para facilitar la creación de pruebas.

Para funciones sin efectos secundarios (escritura en disco, modificación de estructuras de datos persistentes, mensajes entre procesos…) la creación de los test es relativamente inmediata, comprobando generalmente la salida respecto a la entrada y (si es necesario) el lanzamiento de excepciones. Si las funciones tienen efectos secundarios se requiere una infraestructura más compleja para su correcto testeo.

En el caso de las estructuras de datos o el manejo de entrada/salida, habitualmente se requiere una inicialización y configuración inicial antes de realizar la prueba de cierta operación, y posteriormente un limpiado del estado actual del programa. En cambio, cuando se quieren probar funciones con concurrencia, se deben crear procesos falsos que se sitúen en el lugar de otros para interactuar sobre el objeto que se está probando.

Servidor de pruebas de OTP y *Common Test Environment*

El *framework* OTP incluye un entorno de ejecución para pruebas automáticas. Soporta test en local y en remoto y guarda los resultados como páginas HTML. Dentro encontramos el *Common Test Environment* [25] [26], capaz de trabajar como caja blanca y como caja negra. Es capaz de medir la cobertura usando la herramienta de OTP, aunque es algo difícil de usar y la documentación no es suficiente.

Quickcheck

Quickcheck[27]es una herramienta comercial para probar modelos y propiedades, comparando el código con la especificación formal. Puede realizar pruebas a nivel del testeo unitario o a sistemas complejos, generando casos de test aleatorios pero controlables.

Es un método eficiente para encontrar problemas en sistemas muy complejos en etapas tempranas del desarrollo. En cuanto a sus desventajas, se echa de menos información sobre la cobertura de los test, y su aprendizaje es algo complejo.

Herramientas de refactorización

Refactorizar consiste en transformar el código aumentando su calidad (mejorando sus atributos no funcionales[5]) sin variar su comportamiento. En Erlang, las transformaciones más comunes son los cambios de nombres, conversión de estructuras de datos (de tuplas a *records*), operaciones sobre los argumentos de funciones y detección de duplicados.

Las herramientas están integradas en los editores más usados, basadas en el análisis estático del código, aunque su capacidad es limitada. Entre las herramientas más conocidas están *Wrangler* [28], para *Emacs* y *Eclipse*; y *RefactorErl* [29], que funciona de forma independiente bajo Windows o sistemas Unix.

[5] Los atributos no funcionales son los que no afectan al comportamiento del programa, como la legibilidad, mantenibilidad, expresividad o complejidad.

Herramientas de análisis y comprobación

Dialyzer [30]es una herramienta de análisis estático para localizar discrepancias en el código. Encuentra errores de escritura, redundancia de test, *bytecodes* poco seguros para la máquina virtual, y trozos de código no alcanzable.

Como comprobador de modelos tenemos la herramienta *McErlang* [31], con buen soporte para las características más peculiares del lenguaje y capaz de verificar sistemas distribuidos.

Manejo de paquetes y ensamblado automático

Para la ayuda a la instalación y publicación de aplicaciones OTP tenemos la herramienta *Faxien* [32]. Además *Sinan* [33] es capaz de ensamblar y compilar las aplicaciones, generar documentación, ejecutar análisis con *Dialyzer*, pruebas unitarias, informes de salida y manejo de dependencias.

También podemos utilizar Rebar [34], herramienta de ensamblado que minimiza el trabajo de configuración, se encarga de manejar las dependencias y permite reusar bibliotecas de diversas fuentes con facilidad.

Pruebas de carga

Para poner a prueba el escalado y rendimiento de aplicaciones basadas en TCP/IP podemos utilizar la herramienta *Tsung* [35], que puede simular miles de usuarios virtuales desde diferentes máquinas clientes para poner a prueba nuestra aplicación servidor.

Concuerror

Concuerror [36] es una herramienta para detectar errores relacionados con la concurrencia que podrían ocurrir durante la ejecución de un programa Erlang. Está orientada a ser usada en metodologías de desarrollo guiadas por los test.

Mediante la exploración del espacio de estados del programa intenta detectar secuencias de entrelazado de procesos que produzcan efectos no deseados, como pueden ser excepciones, violación de asertos y bloqueos.

Su funcionamiento se resume de la siguiente forma. Desde una interfaz gráfica, el usuario prepara un test para ser ejecutado. Lo primero que hace *Concuerror* es instrumentar el programa, analizando sintácticamente el mismo y transformándolo como corresponde. Tras esta transformación se compila y se carga, y en ese momento es cuando el planificador de la herramienta ejecuta todas las posibles secuencias de entrelazado para dicho test, informando de los resultados en la interfaz gráfica. Haciendo uso de la información, el usuario puede corregir y testear iterativamente la aplicación para ver cómo varía el funcionamiento del programa.

Smother

Smother forma parte del proyecto PROWESS [20]. Es una herramienta con funcionalidad similar al *cover* pero más exhaustiva, ya que realiza una compilación especial, ejecuta y guarda resultados y presenta su análisis como una estructura de datos Erlang para facilitar su integración en herramientas de mayor tamaño.

Smother implementa el criterio de cobertura MC/DC para condiciones booleanas, pero como la mayoría de decisiones en Erlang son por *pattern matching*, *Smother* también implementa dicho criterio para ellas asegurando que cada patrón se da y no se da durante los casos de test, y además que no se da de todas las formas posibles.

Dicha implementación en el *pattern matching* es importante para las sentencias *case* y para el patrón *receive* que maneja la recepción de mensajes de otros procesos también por el mismo sistema.

Para poder realizar todo esto, *Smother* hace uso de la instrumentación, insertando llamadas para realizar anotaciones sobre la ejecución mediante la API de refactorización *Wrangler*. Una vez instrumentado y guardada la información, se buscan las decisiones o patrones y se envían al servidor de procesos de *Smother*, creando una estructura analizable. La información contendrá datos sobre la localización, el tipo y el contenido de dicha decisión, de forma que se pueda determinar con ella la cobertura MC/DC.

En el caso del *pattern matching* en la estructura se anota si el patrón se cumple y cuantas veces, y cuando no se cumple también se comprueba si alguno de sus componentes sí lo hace. Pese a que hay ligeras diferencias en la implementación, el tratamiento de este mecanismo es similar en estructuras *case, receive* y en las cabeceras de función.

1.8 - Tracing en Erlang

Cualquier lenguaje de programación debería ofrecer herramientas o mecanismos para realizar trazas a bajo nivel o capaces de ser la base de herramientas usadas para resolver problemas de ese tipo. En Erlang las funciones de traza estuvieron presentes desde el principio dada la experiencia de Ericsson con los conmutadores de telefonía. Evolucionaron hasta convertirse en un conjunto de herramientas que ofrecen una visibilidad bastante completa del sistema y por tanto reducen el tiempo de respuesta ante los errores.

La base del mecanismo de trazas es la BIF (*built-in function*, función implementada directamente en el intérprete) `erlang:trace/3`. Nos da capacidad para monitorizar memoria, ejecución y concurrencia. Puede ser usada sin compilar el código especialmente para ello, con lo cual se puede usar en sistemas ya activos, lo que la convierte en una herramienta potente para solucionar problemas en grandes sistemas ya entregados y en ejecución.

Todos los eventos de traza se pasan como mensaje al proceso que lanzó la llamada, que pasará a ser el proceso *tracer*. Dicho proceso recibirá mensajes de la forma `{trace, Pid, Tag, Data1 [,Data2]}`, donde el segundo campo de datos es opcional. Los eventos de traza generados son llamadas a función (locales y globales), recolección de basuras y uso de memoria, y actividades relacionadas con los procesos y los mensajes.

La llamada tiene la forma `erlang:trace(PidSpec, Bool, TraceFlags)`. En el primer argumento pasaremos un identificador del proceso a seguir, o alguno de los átomos:

- *existing,* activará la traza para todos los procesos que existan en el momento de la llamada y ninguno más.
- *new,* activará la traza para todos los que se creen a partir de ahora.
- *all*, activará la traza para todos.

El propio proceso *tracer* no puede ser objeto de seguimiento porque produciría bucles infinitos. El segundo argumento será *true* o *false* para activar algunos aspectos del *tracing*, aspectos que se pasarán en el tercer argumento como átomos, una lista de *trace flags* especificando qué eventos se quiera generar o suprimir. Entre ellos encontramos los átomos:

- *send,* para activar la traza sobre eventos de envío de mensaje. Generará eventos, dependiendo de si el proceso al que se envía existe o no, de la forma
  - `{trace, Pid, send, Message, To}`
  - `{trace, Pid, send_to_non_existing_process, Message, To}`
- *'receive',* para activar la traza sobre eventos de recepción. Genera eventos de la forma
  - `{trace, Pid, 'receive', Message}`
- *running,* para rastrear transferencias de estado del módulo, función y aridad así como el identificador de proceso con el que empezó a ejecutarse y con el que terminó. Sus mensajes de traza son un par de la forma
  - `{trace, Pid, in, {M, F, Arity}}`
  - `{trace, Pid, out, {M, F, Arity}}`
- *procs,* para activar la traza sobre eventos relacionados con los procesos, como la creación de nuevos procesos o su terminación, el enlace y el registro. Sus mensajes de traza tienen la forma
  - `{trace, Pid, spawn, Pid2, {M, F, Args}}`
  - `{trace, Pid, exit, Reason}`
  - `{trace, Pid, link | unlink, Pid2}`
  - `{trace, Pid, getting_linked | getting_unlinked, Pid2}`
  - `{trace, Pid, register | unregister, Pid2}`
- *set_on_spawn* y *set_on_first_spawn* controlan la herencia de dichos *flags* de forma que el primero hace que todos los procesos hereden los *flags* del creador, incluido el propio *set_on_spawn*, mientras que en el segundo caso heredarán todos menos ese de forma que los "nietos" no heredarían dichos *flags*. Por tanto, el primero es transitivo, y el segundo no.

- Similarmente existen los átomos *set_on_link* y *set_on_first_link* que funcionan de forma parecida al los anteriores pero actúan sobre la herencia de los enlaces y no sobre la creación de procesos.
- *garbage_collection* permite recopilar información sobre los eventos relacionados con el recogedor de basura, su inicio y terminación. Recoge información sobre la pila, el montículo y el búfer. Produce eventos de la forma:
  - `{trace, Pid, gc_start, Info}`
  - `{trace, Pid, gc_end, Info}`
- *cpu_timestamp* hace que todos los tiempos de las trazas sean relativos al tiempo de CPU y no a la hora.

Además, podemos usar la función `erlang:trace_pattern/3` para crear un conjunto de funciones sobre las que realizar la traza. Combinado con la función anterior, realizaríamos unas trazas de forma que obtendríamos la traza de aquellas funciones que queremos ejecutadas solo por los procesos de nuestro interés.

Usando estas dos funciones tenemos *dbg tracer*, que hace uso de esas funciones de forma más sencilla para el usuario. Gracias a su pequeño impacto en el rendimiento del sistema, lo hace idóneo para realizar trazas de grandes sistemas. A forma de resumen, podemos ver su funcionalidad con su función de ayuda `dbg:h/1`.

```
1> dbg:h().
The following help items are available:
p, c
- Set trace flags for processes
tp, tpl, ctp, ctpl, ctpg, ltp, dtp, wtp, rtp
- Manipulate trace patterns for functions
n, cn, ln
- Add/remove traced nodes.
tracer, trace_port, trace_client, get_tracer, stop, stop_clear
- Manipulate tracer process/port
```

```
i

- Info

call dbg:h(Item) for brief help a brief description

of one of the items above.

ok
```

Capítulo 2 - Propuesta

2.1 - Sintaxis del lenguaje

Presentamos la sintaxis del lenguaje considerada a la hora de definir formalmente la semántica tanto de la ejecución Erlang normal como de la ejecución concólica.

Está ligeramente simplificada, excluyendo algunas características como módulos, excepciones, registros, tipos binarios, monitores, puertos o procesos enlazados, que no son especialmente difíciles pero complicarían la notación y definiciones.

$$
\begin{aligned}
&Program \ni pgm ::= f(\overline{X_n}) \to e. \\
&\qquad\qquad\quad | \quad f(\overline{X_n}) \to case\ e\ of\ cl\ end. \\
&\qquad\qquad\quad | \quad pgm\ pgm \\
&Exp \ni e ::= bv \mid [e_1|e_2] \mid \{\overline{e_n}\} \\
&\qquad\quad | \ X \mid e(\overline{e_n}) \qquad\qquad (n > 0) \\
&\qquad\quad | \ e_1 \,¡\, e_2 \mid receive\ cl\ end \\
&BasicValue \ni bv ::= a \mid n \mid \mathrm{p} \mid [\,] \mid \{\,\} \\
&Value \ni v ::= bv \mid [v_1 \mid v_2] \mid \{\overline{v_n}\} \qquad (n > 0) \\
&Pattern \ni p ::= bv \mid X \mid [p_1 \mid p_2] \mid \{\overline{p_n}\} \qquad (n > 0) \\
&Clauses \ni cl ::= \ p_1\ when\ g_1 \to \ e_1; \\
&\qquad\qquad\quad \ldots; \\
&\qquad\qquad\quad p_n\ when\ g_n \to \ e_n \qquad (n > 0) \\
&where\ a \in Atom, n \in Number, \mathrm{p} \in Pid, X \in Var, g \in Guard
\end{aligned}
$$

Consideraremos una notación más simple de forma que cada función contiene una sola regla con variables como parámetros y el *pattern matching* de la función será representado mediante una expresión *case*. Además, cada función tendrá como mucho una única expresión como cuerpo. Podemos ver que cualquier programa puede ser adaptado a estas restricciones.

Los patrones en el lado izquierdo se pueden transformar de la siguiente forma:

$$f(\overline{p_n})\ when\ g_1 \rightarrow e_1;$$
$$\ldots$$
$$f(\overline{p'_n})\ when\ g_m \rightarrow e_m.$$
$$\downarrow$$
$$f(\overline{X_n}) \rightarrow\ case\ \{\overline{X_n}\}\ of$$
$$\{\overline{p_n}\}\ when\ g_1 \rightarrow\ e_1$$
$$\ldots$$
$$\{\overline{p'_n}\}\ when\ g_m \rightarrow\ e_m.$$

Podemos transformar el *pattern matching:*

`p = e, e2, …,en` en `case e of p` → `e2, …, en`.

Si en las secuencias `e1,…,en` e_1 no es de la forma `p = e`, podemos transformarlo en `case e of _` → `e2,…,en`.

Por último, cuando hay expresiones *case* que no están en la parte más alta de la función, introduciremos funciones auxiliares para que lo sean, de la siguiente forma:

$$f(\overline{X_n}) \rightarrow\ C\ [case\ \{\overline{X_n}\}\ of$$
$$\{\overline{p_n}\}\ when\ g_1 \rightarrow\ e_1$$
$$\ldots$$
$$\{\overline{p'_n}\}\ when\ g_m \rightarrow\ e_m].$$
$$\downarrow$$
$$f(\overline{X_n}) \rightarrow\ C\ [\ g\ (\overline{X_n})\ ]$$
$$g(\overline{Y_n}) \rightarrow\ case\ \{\overline{Y_n}\}\ of$$
$$\{\overline{p_n}\}\ when\ g_1 \rightarrow\ e_1$$
$$\ldots$$
$$\{\overline{p'_n}\}\ when\ g_m \rightarrow\ e_m].$$

Donde *C[ ]* es un contexto arbitrario y *g* es una símbolo de función *fresh*[6].

Por tanto, los programas son secuencias de funciones, cada una definida por una sola regla y cuyas expresiones del cuerpo solo pueden contener valores básicos, listas, tuplas, variables, aplicaciones de funciones, y paso y recogida de mensajes.

Por simplicidad, excluiremos de la sintaxis las funciones anónimas declaradas tal que:

```
fun(X1,…, Xn) -> e end
```

y por tanto asumiremos que todas están declaradas en el programa. Además de las definidas, podremos usar los habituales operadores *built-in*: lógicos(*and*, *not*, *or*, *xor*), relacionales (>, <, =<, >=, ==, /=), aritméticos (+,-,*,div), de manejo de listas(hd, tl, element, ++, length) y funciones relacionadas con la concurrencia (*self* y *spawn*). Tan solo las acciones concurrentes producen efectos secundarios.

Denotaremos las variables en mayúsculas, átomos en minúsculas, *p*, *p'*… para identificar procesos (pids), *f/n, g/m…* para funciones y *op* para las operaciones *built-in* distintas a *self* y *spawn*. El dominio de los pids y el de los átomos deben ser disjuntos.

[6] Llamaremos *fresh* a un nombre nuevo no utilizado previamente en el programa.

2.2 - Semántica de Erlang

No existe una definición semántica estándar y aceptada para Erlang, aunque si hay varias aproximaciones en diferentes trabajos en la materia. Presentaremos ahora una aproximación basada en el trabajo de H.Svensson [37].

Erlang sigue una semántica operacional primero por la izquierda y por el interior. Cada expresión *C[e]* consta de un contexto *C* y una expresión *e* donde tendrá lugar la siguiente reducción.

$$\begin{aligned} C ::= [\,] \mid & \; case\ C\ of\ cl\ end \mid C\,!\,e \mid v\,!\,C \mid C\ (e_1, \dots, e_n) \\ & \mid f(v_1, \dots, v_i, C, e_{i+2}, \dots e_n) \\ & \mid op(v_1, \dots, v_i, C, e_{i+2}, \dots e_n) \\ & \mid [\,v_1, \dots, v_i, C|e] \mid \{v_1, \dots, v_i, C, e_{i+2}, \dots e_n\} \end{aligned}$$

Denotaremos un proceso Erlang mediante la tupla <p; *e*; *q*> en la que p es el identificador de proceso, *e* es la expresión que será evaluada y *q* es el buzón de mensajes del proceso. Un sistema Erlang vendrá definido por el par (Π,Q), donde Π es un grupo de procesos y Q el sistema de mensajes. Asumiremos, dado que no es relevante para nuestro propósito, que no hay orden en Q. El sistema de mensajes Q será el conjunto <p, p', *q*> donde *q* es una lista de mensajes que el proceso p envía al p'.

Erlang solo garantiza que el orden será preservado entre los mensajes que se envíen entre un par de procesos. Sin embargo, si otro mensaje es enviado a través de un tercer proceso pueden llegar en cualquier orden.

Veamos ahora la definición formal de la semántica explicando las menos claras.

$$(self)\ \frac{}{(<\mathrm{p};\mathrm{C}\ [\ \mathrm{self}(\,)];q> \,\&\, \Pi, Q) \rightarrow_\tau (<\mathrm{p};\mathrm{C}[\mathrm{p}];q> \,\&\, \Pi, Q)}$$

$$(builtin)\ \frac{eval\left(op(\overline{v_n})\right) = v}{(<p;\mathrm{C}\ [op(\overline{v_n})];q> \,\&\, \Pi, Q) \rightarrow_\tau (<p;\mathrm{C}[v];q> \,\&\, \Pi, Q)}$$

Por simplicidad, asumiremos que *eval* es capaz de evaluar todas las funciones *builtin* que no tienen efectos secundarios.

$$(fun)\frac{f(\overline{X_n}) \to e. \in pgm}{(<\mathrm{p};\mathrm{C}\,[f(\overline{v_n})];q>\,\&\,\Pi,Q) \to_\tau (<\mathrm{p};\mathrm{C}[\hat{\mathrm{e}}\,\{\overline{X_n \mapsto v_n}\}];q>\,\&\,\Pi,Q)}$$

Para *fun*, asumimos que el programa *pgm* es una variable global del sistema de transición. Denotamos con *ê* una copia de *e,* cuyas variables tienen nombres *fresh*. $\overline{X_n \mapsto v_n}$ denota una sustitución que enlaza las variables X_1… X_n a sus valores correspondientes en *v*. Aplicar la sustitución σ a una expresión *e* se denota con *e* σ.

$$(case)\frac{match(v,cl) = (e,\sigma)}{(<\mathrm{p};\mathrm{C}\,[case\ v\ of\ cl\ end];q>\,\&\,\Pi,Q) \to_\tau (<\mathrm{p};\mathrm{C}[e\sigma];q>\,\&\,\Pi,Q)}$$

En el *case* asumimos que la función *match* es la encargada del *pattern matching*. Concretamente, toma el valor *v* y las cláusulas p_1 *when* g_1 → e_1;…; p_n *when* g_n → e_n y devuelve un par (e_i, σ) si *i* es el número más pequeño tal que $p_i\sigma = v$ y $eval\ (g_i\sigma) = true$.

$$(receive)\frac{q = [q_1,\dots,q_l]\ \wedge matchrec\ (q,\overline{p_m when\ g_n \to\ e_n}) = (e_i,\sigma_i,q_j) \wedge q' = [q_1,\dots,q_{j-1},q_{j+1},\dots,q_l]}{(<\mathrm{p};\mathrm{C}\,[receive\ cl\ end];q>\,\&\,\Pi,Q) \to_\tau (<\mathrm{p};(\mathrm{C}[e_i])\sigma_i;q'>\,\&\,\Pi,Q)}$$

En este caso usaremos una función similar a *match*, *matchrec*, que tomará la cola de mensajes *q* y las cláusulas *cl* y determinará el primer mensaje que *match(v,cl) = (e,σ)* devolviendo *(e,σ,q')* donde *q'* es la nueva cola tras eliminar el mensaje *v*.

$$(spawn)\frac{\mathrm{p}'is\ a\ fresh\ pid}{(<\mathrm{p};\mathrm{C}\,[spawn(f,\overline{v_n})];q>\,\&\,\Pi,Q) \to_\tau (<\mathrm{p};\mathrm{C}[p'];q\ >\,\&<\mathrm{p}',\overline{v_n},[\,]>\,\&\ \Pi,Q)}$$

$$(send)\frac{v_1 = \mathrm{p}' \in Pid\ \wedge add\_msg(\mathrm{p},\mathrm{p}',v,Q) = Q'}{(<\mathrm{p};\mathrm{C}\,[v_1!\,v_2];q>\,\&\,\Pi,Q) \to_\tau (<\mathrm{p};\mathrm{C}[v_2];q\ >\,\&\ \Pi,Q')}$$

En la regla *send* vemos como el mensaje queda guardado en el sistema de mensajería mediante la función *add_msg* cuyo funcionamiento es obvio.

Además guardará el Pid de origen y de destino del mismo. Nótese que el mensaje no llega al destino hasta que no entre en juego la regla *sched*.

$$(sched)\ \frac{(\mathrm{p},\mathrm{p}') \in sched\ (\Pi, Q\ ) \wedge delivery(\mathrm{p},\mathrm{p}', \Pi, Q) = (\ \Pi', Q')}{(\Pi, Q) \rightarrow_{\alpha} (\Pi', Q')}$$

Por último, la regla *sched* usa la función auxiliar homónima para modelar una política de planificación. Básicamente elegirá dos identificadores de proceso entre los cuales la cola no esté vacía y moverá el primer mensaje *q* al buzón local del proceso receptor originando un nuevo par *(Π',Q')*.

Todas las reglas están etiquetadas con τ excepto la última. Usamos esta notación para marcar que son computaciones normalizadas. Dado un estado *s*, denotaremos como $s\downarrow^{\tau}$ el estado que resulta desde *s* aplicando solo las transiciones etiquetadas con τ, hasta que no sea posible seguir aplicándolas. Dichas computaciones serán deterministas puesto que no habrá intervención del planificador entre ellas, y así conseguimos reducir el espacio de búsqueda. Para que esto sea justo, asumimos que ningún proceso Erlang estará en ejecución indefinidamente sin llegar a un *receive*.

2.3 - Semántica de la ejecución generadora de traza

Ahora vamos a ver la semántica de ejecución para una ejecución simbólica guiada por la traza generada por una semántica concreta. Formalmente:

$$(self)\ \frac{}{(<\mathrm{p};\mathrm{C}[\ \mathrm{self}(\ )];q>\ \&\ \Pi,Q)\xrightarrow{\lozenge}_{\tau}(<\mathrm{p};\mathrm{C}[p];q>\ \&\ \Pi,Q)}$$

$$(builtin)\ \frac{eval\big(op(\overline{v_n})\big)=v}{(<p;\mathrm{C}\ [op(\overline{v_n})];q>\ \&\ \Pi,Q)\xrightarrow{\lozenge}_{\tau}(<p;\mathrm{C}[v];q>\ \&\ \Pi,Q)}$$

$$(fun)\ \frac{f\big(\overline{X_n}\big)\rightarrow e.\in pgm}{(<\mathrm{p};\mathrm{C}\ [f(\overline{v_n})];q>\ \&\ \Pi,Q)\xrightarrow{f}_{\tau}(<\mathrm{p};\mathrm{C}\big[\hat{\mathrm{e}}\ \{\overline{X_n\mapsto v_n}\}\big];q>\ \&\ \Pi,Q)}$$

$$(case)\ \frac{match(v,cl)=(e,\sigma)}{(<\mathrm{p};\mathrm{C}\ [case\ v\ of\ cl\ end];q>\ \&\ \Pi,Q)\xrightarrow{(f,i)}_{\tau}(<\mathrm{p};\mathrm{C}[e\sigma];q>\ \&\ \Pi,Q)}$$

$$(receive)\ \frac{q=[q_1,\ldots,q_l]\ \wedge matchrec\ \big(q,\overline{p_m when\ g_n\rightarrow\ e_n}\big)=\big(e_i,\sigma_i,q_j\big)\wedge q'=\big[q_1,\ldots,q_{j-1},q_{j+1},\ldots,q_l\big]}{(<\mathrm{p};\mathrm{C}\ [receive\ cl\ end];q>\ \&\ \Pi,Q)\xrightarrow{(f,i,j)}_{\tau}(<\mathrm{p};(\mathrm{C}[e_i])\sigma_i;q'>\ \&\ \Pi,Q)}$$

$$(spawn)\ \frac{\mathrm{p}'is\ a\ fresh\ pid}{(<\mathrm{p};\mathrm{C}\ [spawn(f,\overline{v_n})];q>\ \&\ \Pi,Q)\xrightarrow{(\mathrm{p}',f)}_{\tau}(<\mathrm{p};\mathrm{C}[\mathrm{p}'];q\ >\ \&<\mathrm{p}',\overline{v_n},[\ ]>\ \&\ \Pi,Q)}$$

$$(send)\ \frac{v_1=\mathrm{p}'\in Pid\ \wedge add\_msg(\mathrm{p},\mathrm{p}',v,Q)=Q'}{(<\mathrm{p};\mathrm{C}\ [v_1!\,v_2];q>\ \&\ \Pi,Q)\xrightarrow{\mathrm{p}'}_{\tau}(<\mathrm{p};\mathrm{C}[v_2];q\ >\ \&\ \Pi,Q')}$$

$$(sched)\ \frac{(\mathrm{p},\mathrm{p}')\in sched\ (\ \Pi,Q\ )\wedge delivery(\mathrm{p},\mathrm{p}',\pi,Q)=(\ \Pi',Q')}{(\Pi,Q)\xrightarrow{(\mathrm{p},\mathrm{p}')}_{\alpha}(\Pi',Q')}$$

Consideramos la siguiente información de traza:

- Con el símbolo ◊ expresamos que no hay nada relevante en la transición.
- Para el desplegado de funciones etiquetaremos la transición con el símbolo de la llamada.
- Cuando se evalúa una expresión *case*, etiquetamos la transición con un par *(f,i)* donde *f* es la función donde aparece dicho *case* e *i* es el número de rama escogida. Se procederá demanera similar con los *receive*.

- Los *spawn* serán etiquetados con el par *(p,f)* correspondiente al identificador de proceso del nuevo y a la función desde la cual fue creado.
- Los *send* serán etiquetados con la información del proceso al que van dirigidos.
- Por último, los pasos del planificador irán asociados a un par *(p,p')* que representarán el proceso origen y el destino.

Por tanto, definiremos una traza del programa tal que: Sea *pgm* es el programa y el sistema inicial s_0 = *(<p_0,e_0>,[ ])*. Sea $s_0 \xrightarrow{w_1} s_1 \xrightarrow{w_2} s_2 \ldots \xrightarrow{w_n} s_n$ una ejecución que genera traza. La traza asociada será la lista *[w_1, w_2, …, w_n].*

En la práctica puede que la información recogida en la traza varíe ligeramente para evitar ciertos problemas relacionados con la implementación.

2.4 - Semántica de la ejecución concólica

Veremos la semántica de la ejecución simbólica guiada por una traza, esto es, la ejecución concólica, para explorar tan solo computación simbólica en vez de todo el espacio de búsqueda que suele ser inmenso si no infinito.

$$(self)\ \frac{}{(\lozenge\colon \pi, < \mathrm{p}; \mathrm{C}\ [\ \mathrm{self}(\ )]; q > \&\ \Pi, Q) \overset{\lozenge}{\rightsquigarrow}_{\tau} (\pi, < \mathrm{p}; \mathrm{C}[\mathrm{p}]; q > \&\ \Pi, Q)}$$

$$(builtin1)\ \frac{eval\big(op(\overline{v_n})\big) = v}{(\lozenge\colon \pi, < p; \mathrm{C}\ [op(\overline{v_n})]; q > \&\ \Pi, Q) \overset{\lozenge}{\rightarrow}_{\tau} (< p; \mathrm{C}[v]; q > \&\ \Pi, Q)}$$

$$(builtin2)\ \frac{\exists i.\, p_i\ is\ not\ a\ value, X\ is\ a\ fresh\ variable, and\ g\ \equiv\ \big(X =\ op(\overline{p_n})\big)}{(\lozenge\colon \pi, < p; \mathrm{C}\ [op(\overline{p_n})]; q > \&\ \Pi, Q) \xrightarrow{(id,g)}_{\tau} (\pi, < p; \mathrm{C}[X]; q > \&\ \Pi, Q)}$$

$$(fun)\ \frac{f\big(\overline{X_n}\big) \rightarrow e. \in pgm}{(f\ \colon \pi, < \mathrm{p}; \mathrm{C}\ [f(\overline{p_n})]; q > \&\ \Pi, Q) \overset{f}{\rightarrow}_{\tau} (\pi, < \mathrm{p}; \mathrm{C}\big[\hat{\mathrm{e}}\ \{\overline{X_n \mapsto p_n}\}\big]; q > \&\ \Pi, Q)}$$

$$(case)\ \frac{< e, \sigma, g > \in unify\ (p, i, cl)}{\big((f, i)\ \colon \pi, < \mathrm{p}; \mathrm{C}\ [case^{f}\ p\ of\ cl\ end]; q > \&\ \Pi, Q\big) \xrightarrow{(\sigma,g)}_{\tau} (\pi, < \mathrm{p}; (\mathrm{C}[e])\sigma; q > \&\ \Pi, Q)}$$

$$(receive)\ \frac{(e, \sigma, g, q')\ \in unifyrec(q, i, cl)}{\big((f, i, j)\ \colon, < \mathrm{p}; \mathrm{C}\ [receive\ cl\ end]; q > \&\ \Pi, Q\big) \xrightarrow{(\sigma,g)}_{\tau} (\pi, < \mathrm{p}; (\mathrm{C}[e])\sigma; q' > \&\ \Pi, Q)}$$

$$(spawn)\ \frac{}{\big((\mathrm{p}', f)\ \colon \pi, < \mathrm{p}; \mathrm{C}\ [spawn(f, \overline{p_n})]; q > \&\ \Pi, Q\big) \overset{\lozenge}{\rightarrow}_{\tau} (\pi, < \mathrm{p}; \mathrm{C}[\mathrm{p}']; q > \& < \mathrm{p}', f(\overline{p_n}), [\ ] > \&\ \Pi, Q)}$$

$$(send)\ \frac{v = \mathrm{p}' \in Pid\ \wedge add\_msg(\mathrm{p}, \mathrm{p}', p, v, Q) = Q'}{(\mathrm{p}'\colon \pi, < \mathrm{p}; \mathrm{C}\ [v!\, p]; q > \&\ \Pi, Q) \overset{\lozenge}{\rightarrow}_{\tau} (< \mathrm{p}; \mathrm{C}[p]; q\ >\ \&\ \Pi, Q')}$$

$$(sched)\ \frac{delivery(\mathrm{p}, \mathrm{p}', \pi, Q) = (\ \Pi', Q')}{\big((\mathrm{p}, \mathrm{p}')\ \colon \pi, \Pi, Q\big) \overset{\lozenge}{\rightarrow}_{\alpha} (\pi, \Pi', Q')}$$

Formalizando, podremos decir que un sistema Erlang simbólico es una tupla de < *π, Π, Q, PC, Θ* > donde *Π* y *Q* representan el grupo de procesos y el sistema de mensajes, *π* a la traza de ejecución concreta, *PC* es una condición de camino y *Θ* una secuencia de sustituciones.

2.5 - Algoritmo de ejecución concólica

Procederemos de la siguiente forma:

1. Consideraremos que la función inicial siempre es *main/n*.
2. Se comienza la ejecución con $(< p_0, main\ (\overline{v_n}) >, [\,])$ con datos de entrada arbitrarios usando la semántica de traza. Llamemos π a la traza asociada *[w_1, w_2, …, w_n]*.
3. Ahora elegiremos un evento de la forma w_j de tal forma que su definición contenga ramas de un *case* o un *receive* sin explorar. Llamaremos *(f, l)* a dicha rama.
4. Ejecutamos concólicamente con la llamada $main(\overline{X_n})$ y la traza *π' = [w_1, w_2, …, w_{j-1}, (f,l)]*. La salida será una serie de sustituciones $\sigma_1 \ldots \sigma_c$ y guardas $g_1 \ldots g_c$ tal que $main(\overline{X_n})\ \sigma_1 \ldots \sigma_c \theta$, donde *θ* representa una sustitución *ground* que hace que se cumpla $(g_1 \wedge \ldots \wedge g_c)\ \theta$, y será la nueva entrada del programa.
5. Por último, volveremos al paso dos utilizando la nueva llamada $main(\overline{X_n})\ \sigma_1 \ldots \sigma_c \theta$ y actualizando las ramas de casos visitados.

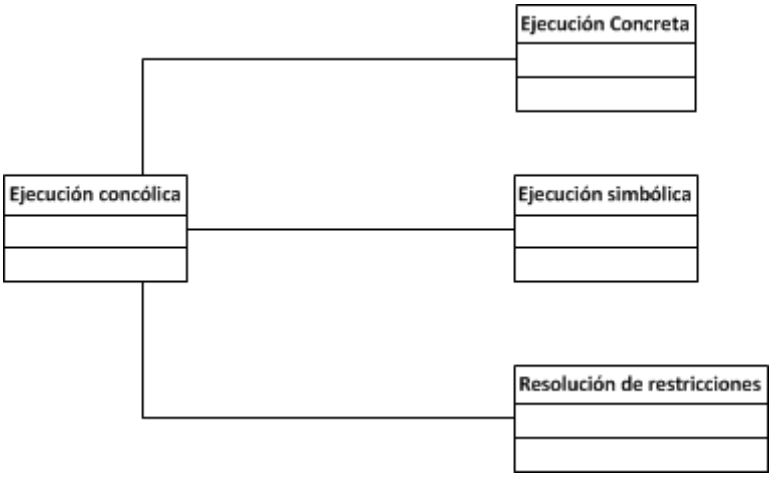
Capítulo 3 – Diseño e implementación

Nuestra propuesta de ejecución "concólica" está basada en tres grandes pasos:

1) Una ejecución concreta de la que guardamos una traza.
2) Una ejecución simbólica con la que recogemos las restricciones sobre las variables que lleva la ejecución concreta previa.
3) Negación de alguna de las restricciones y generación de nuevos valores de entrada para repetir el proceso desde el principio.

Todo está integrado en una sola aplicación Erlang a la que proporcionaremos una llamada inicial con valores de entrada cualesquiera e irá produciendo casos de entrada y ejecuciones en busca de posibles errores y terminaciones no esperadas.

De forma muy genérica, nuestra herramienta tendría la siguiente estructura.



Desde el módulo principal se recibe la llamada inicial al programa a probar, que será pasada a los módulos de ejecución concreta, que generará una traza del programa; y a la ejecución simbólica, que guiada por dicha traza recorrerá el programa recogiendo las restricciones y modificaciones que se aplican a las

variables de entrada para seguir ese camino concreto. Se modificará alguna de ellas para llevar a la ejecución por caminos no explorados. Con esta lista de restricciones se llamará al módulo encargado de resolverlas, de forma que obtendrá un conjunto concreto de variables que cumplen con las restricciones que se le pasan, y con esos nuevos datos de entrada se repetirá el proceso.

Cabe destacar que las decisiones sobre qué restricción negar para elegir el nuevo camino de ejecución se deben tomar basándose en criterios de cobertura. Cada decisión debe estar orientada a aumentar la cobertura del código lo máximo posible, optando por los caminos que ofrezcan mayor posibilidad de explorar nuevos fragmentos del programa. Para ello se pueden implementar diversos criterios de cobertura (desde cobertura de sentencias a criterios más complejos como MC/DC) y experimentar con ellos con el sistema completo buscando el que ofrezca resultados más satisfactorios.

Pese a que nuestra traza pretende ser lo menos intrusiva posible, es probable que dada la instrumentación del código podamos, en algunos casos, influir en el comportamiento del planificador y producir situaciones que el programa original no experimentaría. Para paliar estos efectos podemos combinar nuestro sistema de trazas con el *Concuerror*, que nos permitirá analizar todas las planificaciones posibles y por tanto comprobar el correcto funcionamiento del sistema pese a la instrumentación.

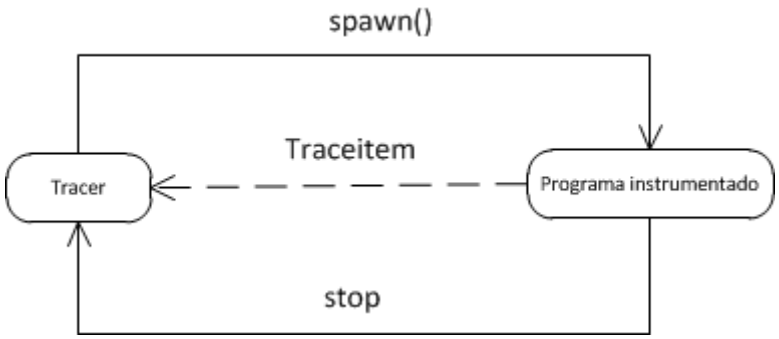
Además, la combinación con el *Concuerror* nos ayuda a aumentar la cobertura en general, comprobando, para cada caso de test generado por nuestro sistema, todas las posibles variaciones del planificador para dicha entrada. Obviamente para sistemas grandes en los que solo la ejecución concólica ya produzca muchísimos casos de test y un largo proceso de prueba, la combinación con el *Concuerror* puede hacer dicho proceso mucho más largo. Sin embargo, los beneficios son obvios y sería interesante comprobar de esta forma las secciones más críticas del objeto de prueba.

3.1 - Ejecución concreta

El bloque de ejecución concreta será el encargado de generar un programa instrumentado de tal forma que su ejecución nos proporcione una traza con la información relevante para los otros pasos del sistema.

Consta de dos módulos: el primer módulo será el *tracer*, que guiará el proceso de la ejecución concreta y guardará la traza. Para ello recibirá como único argumento la llamada inicial del programa a testear, y devolverá la traza del programa, ya sea en forma de variable para ser utilizada posteriormente, o bien imprimiendo de una forma legible el resultado por la salida estándar.

A nivel de procesos nuestra estructura es muy sencilla. Podemos verla representada en la siguiente figura.

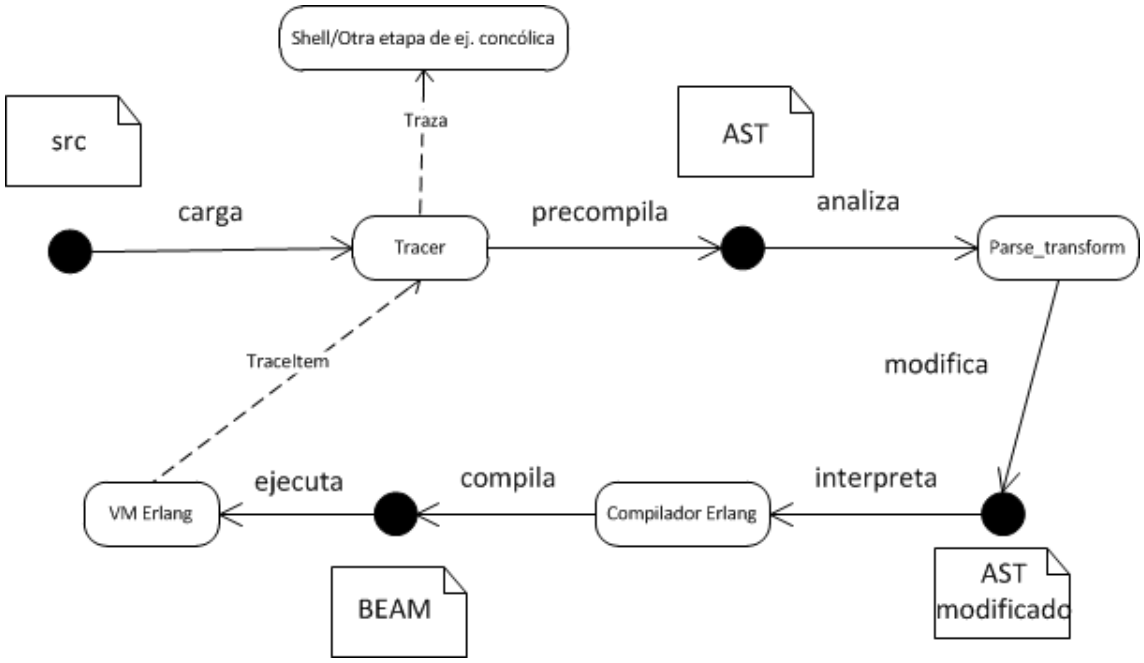


Nuestro *tracer* preparará el código del programa a testear, instrumentándolo y lo lanzará en otro proceso mediante la función *spawn*. Quedará a la escucha de los mensajes que produce la instrumentación (*Traceitem*) hasta que reciba un *stop* indicándole que la ejecución ha terminado.

Para instrumentar el programa utilizaremos una característica de Erlang que nos permite compilar un archivo fuente en tiempo de ejecución y además, como parámetro, podemos incluirle un módulo de transformación de código.

Ése será nuestro segundo módulo. En él podremos definir para cada uno de los patrones que nos interesen en nuestra traza, el tipo de mensaje que queremos generar.

En el siguiente esquema podemos ver las etapas que atravesará el código del programa que esté siendo sometido al testeo.



Leeremos la fuente partiendo de la llamada inicial, que será precompilada en formato de árbol sintáctico abstracto (AST) y, en este formato, se analizarán sus sentencias y aplicarán las transformaciones definidas en *parse_transform*. A partir de ahí se generará el AST modificado, que será compilado y ejecutado de manera normal. Sus modificaciones producirán un mensaje con información para cada evento relevante, que será enviado al *tracer*. Cuando el *tracer* reciba el mensaje de que la ejecución ha terminado, imprimirá la traza o la pasará al siguiente paso de la ejecución concólica.

Para las sentencias *case*, se quiere guardar información sobre la localización de la sentencia, las diferentes alternativas disponibles, de qué depende cada una de ellas y por qué camino ha entrado la ejecución concreta.

Para las sentencias *receive*, de forma similar, queremos saber localización de la sentencia, diferentes alternativas y por qué camino ha entrado la ejecución. Sin embargo, no se depende de una variable concreta sino de la recepción de mensajes y de proceder a su comprobación de patrones con las cláusulas del *receive*, y por tanto no podemos guardar esa información. En cambio, sí que

puede resultar útil conocer la cola de mensajes del proceso así que se hace una tercera modificación al código añadiendo a la traza información sobre la cola de mensajes de cada proceso cada vez que se produce una instrucción de envío de mensajes. Dicha información debe ser suficiente para identificar unívocamente el mensaje que fue seleccionado en cada instrucción *receive*. Se instrumentará el envío de mensajes, asignando a cada mensaje enviado un nuevo valor numérico y adaptando los patrones de las construcciones *receive* a esa modificación, de forma que cualquier mensaje enviado pasará de ser:

```
To ! Message            >> To ! {Id, Message}
```

y los patrones:

```
Pattern           >> {Id, Pattern}
```

Esta modificación hará que, cuando ejecutemos el programa modificado desde nuestro primer módulo *tracer*, el programa modificado nos envíe en forma de mensajes los eventos que queremos registrar en nuestra traza. Deberemos procesarlos y almacenarlos como sea conveniente y permanecer en escucha hasta el fin de la ejecución concreta, cuando devolveremos o imprimiremos la traza.

Veamos ahora, organizado por módulos, el comportamiento de cada una de las funciones, junto con sus parámetros de entrada y salida y las estructuras de los mismos.

→Tipos de datos: *trace*

La variable Trace será la encargada de recoger todos los eventos sobre los que queramos almacenar información. Por tanto será una lista cuyo tamaño dependerá del programa a analizar, y que contendrá eventos de traza de diversos tipos. Veamos qué tipo de información almacena en cada caso.

```
>>Eventos tipo receive // antes de procesar un mensaje

>>{NEvent, {{TEvent = messages, Queue},Pid}}
```

- NEvent = Número de evento, autonumérico creciente para cada evento registrado en la traza.
- TEvent = Tipo de evento. Para guardar los mensajes que habían en cola previamente a la ejecución del *receive* usamos *messages*.
- Pid = Proceso que está a punto de procesar un mensaje o suspenderse en espera de que éstos lleguen.

```
>>Eventos tipo receive // tras procesar un mensaje

>>{NEvent, {TEvent,{Line, NClauseselected,Nmess, Clauseselected,
Allclauses}}}
```

- NEvent = Número de evento, autonumérico creciente para cada evento registrado en la traza.
- TEvent = Tipo de evento. Para los *receive* es el átomo *receive_trace*.
- Line = Línea en la que se encuentra la estructura *receive* para identificarla respecto a otras.
- Nclauseselected = Número de la cláusula por la que ha entrado el *receive*.
- Nmess = Identificador del mensaje que ha sido recibido.
- Clauseselected = Cláusula por la que ha entrado el *receive*.
- Allclauses = Resto de cláusulas.

```
>>Eventos tipo case

>>{NEvent,{TEvent,{Line,NClauseselected,Expression,Clauseselected,All
clauses}}}
```

- NEvent = Número de evento, autonumérico creciente para cada evento registrado en la traza.
- TEvent = Tipo de evento. Para los *case* es el átomo *case_trace*.
- Line = Línea en la que se encuentra la estructura *case* para identificarla respecto a otras.
- Nclauseselected = Número de la cláusula por la que ha entrado el *case*.
- Expression = Expresión de la que dependía la decisión del *case*.
- Clauseselected = Cláusula por la que ha entrado el *case*.
- Allclauses = Resto de cláusulas.

```
>>Eventos tipo send
>>{NEvent, {TEvent,{Pid,Call,{messages, Queue}}}}
```

- NEvent = Número de evento, autonumérico creciente para cada evento registrado en la traza.
- TEvent = Tipo de evento. Para los *case* es el átomo *send_trace*.
- Pid = Pid del proceso que envía el mensaje.
- Call = AST de la llamada al operador '!'.
- Queue = Cola de mensajes del receptor del mensaje antes del *send* en cuestión.

→Módulo case_tracer

Función	*trace/1*
Parámetros de entrada	Call (string)
Salida	Trace (trace)
Llamadas	*parse_call/1*, *compile/2*, *receive_loop/2*, *pretty_printer/1*
Descripción	Función principal. Recibe la llamada, lanza la compilación y ejecución modificadas, el bucle de trazado y devuelve la traza o la imprime

Función	*receive_loop/2*
Parámetros de entrada	Current (int), Trace (trace)
Salida	Trace (trace)
Llamadas	*receive_loop/2*
Descripción	Función que espera la llegada de mensajes con información de la traza y los añade acumulándolos junto a un contador de elementos.

Función	*parse_call/1*
Parámetros de entrada	ExprString (string)
Salida	Term (AST)
Llamadas	
Descripción	Función auxiliar. Analiza la llamada pasada como string y la devuelve en forma de AST.

Función	*parse_expr/1*
Parámetros de entrada	Func (string)
Salida	Res (AST)
Llamadas	
Descripción	Función auxiliar. Prepara la llamada inicial para ser analizada.

Función	*pretty_printer/1*
Parámetros de entrada	Trace(trace)
Salida	
Llamadas	*pretty_printer/1*
Descripción	Función auxiliar. Imprime recursivamente todos los eventos registrados en la traza de forma entendible para el usuario humano.

Función	*freshid/0*
Parámetros de entrada	
Salida	Id(int)
Llamadas	
Descripción	Función auxiliar. Solicita al proceso encargado de generar valores autonuméricos para identificar mensajes el siguiente valor disponible.

Función	*newid/1*
Parámetros de entrada	N(int)
Salida	
Llamadas	*newid/1*
Descripción	Función auxiliar. Será lanzada como proceso aparte con valor inicial 1. Responderá a los mensajes con su valor actual y se llamará de nuevo incrementando en 1 más su valor.

→Modulo *case_clause_sender_pt*

Función	*parse_transform/2*
Parámetros de entrada	Forms (AST), _
Salida	NForms (AST)
Llamadas	*case_clause_sender_fun/1*
Descripción	Función de entrada a la transformación. Con los Forms del compilador (código en AST) mapea una llamada a *case_clause_sender_fun/1* para cada Form, devolviendo los nuevos tras la modificación.

Función	*case_clause_sender_fun/1*
Parámetros de entrada	T (AST)
Salida	form
Llamadas	*case_clause_sender_expr/1*
Descripción	Función auxiliar. Garantiza que los AST tras la transformación sigan bien formados.

Función	*case_clause_sender_expr/1*
Parámetros de entrada	Form (AST) →{'case',LINE,E,Clauses} →{'receive',LINE,Clauses} →{op, LINE,'!',To,Message}
Salida	Form (AST)
Llamadas	*change_clauses/5, change_clauses2/4*
Descripción	Busca entre los Form los que cumplan alguno de los patrones de evento a registrar para modificarlos o llamar a la función que los modifique. La modificación incluye el instrumentado con Id de los mensajes y la correspondiente adaptación del patrón del *receive*.

Función	*change_clauses_case/5*
Parámetros de entrada	Form (AST) →[{clause,LINE,Pattern,Guards,Body}| Clauses], Num,E,PatternsClauses,CaseLine
Salida	AST
Llamadas	*change_clauses_case/5*
Descripción	Para cada alternativa de una estructura *case* añade el paso de un mensaje al *tracer* de forma que según el camino elegido se envíe información al respecto.

Función	*change_clauses_receive/4*
Parámetros de entrada	Form (AST) →[{clause,LINE,Pattern,Guards,Body}| Clauses],Num,PatternsClauses,CaseLine
Salida	AST
Llamadas	*change_clauses_receive/4*

Descripción	Para cada alternativa de una estructura *receive* añade el paso de un mensaje al *tracer* de forma que según el camino elegido se envíe información al respecto. Al no depender de ningún término sino de un mensaje recibido tiene un parámetro de entrada menos que *change_clauses*/5. Además, modificará los patrones para aceptar los mensajes instrumentados con número de mensaje.

3.2 - Ejecución simbólica

Para poder realizar la ejecución simbólica es necesario implementar un intérprete para el lenguaje y añadirle la capacidad de manejar variables simbólicas. Como es obvio, dicha implementación es un trabajo titánico para un lenguaje completo y con tantas características peculiares como es Erlang. Por lo tanto, el diseño y la implementación de un sistema de ejecución simbólica se salen del alcance de este trabajo.

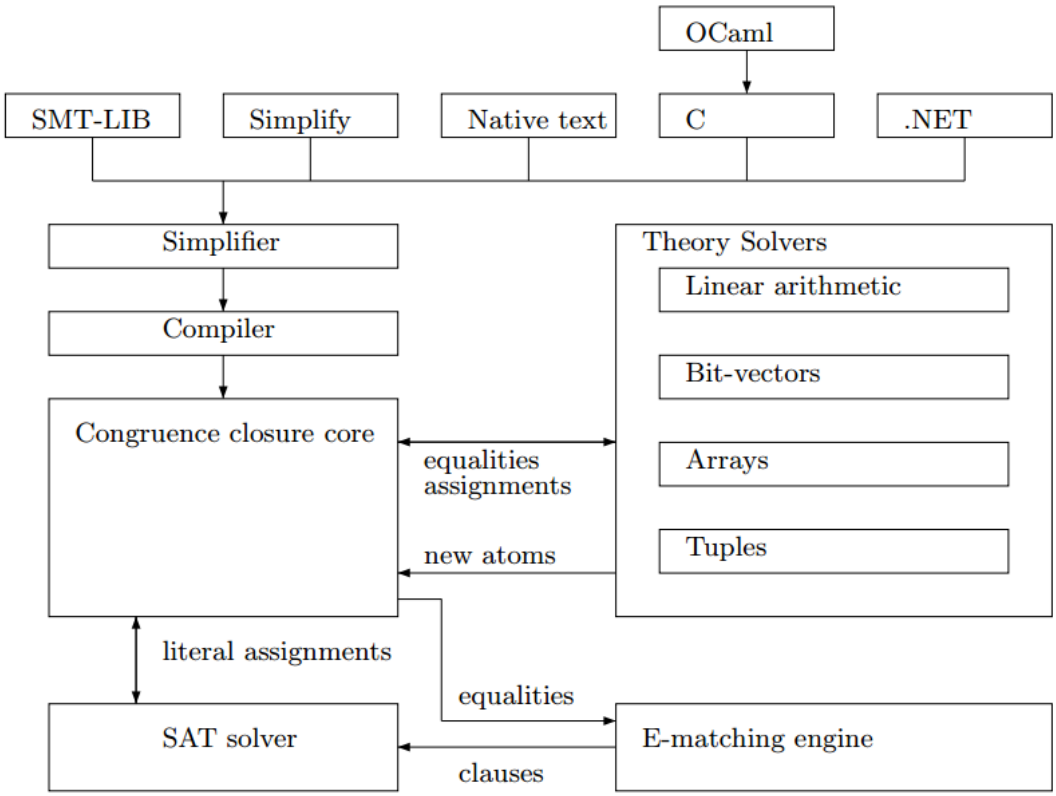
3.3 - Resolución de restricciones

Como se comentaba anteriormente, uno de los problemas de estas técnicas aparece a la hora de resolver las restricciones acumuladas durante la ejecución simbólica para obtener nuevos datos.

Para el caso de nuestro sistema, se utilizará el probador de teoremas y solucionador de restricciones Z3 [38], desarrollado por el departamento de investigación de Microsoft.

Para las restricciones con las que se manejaría el sistema inicialmente es una herramienta extremadamente poderosa y solo se usaría una pequeña parte de su potencia, pero consideramos que es innecesario desarrollar nuestro solucionador de restricciones existiendo algunos con eficacia y potencia más que demostrada y contrastada.

Z3 trabaja con SMT (satisfiability modulo theories) generalizando la satisfacibilidad (SAT) añadiendo algunas teorías de primer orden. Está implementado en C++ aunque ofrece interfaces de conexión en diversos lenguajes para facilitar la interoperabilidad dentro de sistemas más grandes. Entre estos lenguajes tenemos C, C++, .NET, Ocaml y Python. También acepta el formato SMT-LIB.



En la imagen podemos ver su esquema de funcionamiento. Primeramente se realizan ciertas simplificaciones y tras compilarlo se inicia la búsqueda de la satisfacibilidad entre el resto de módulos, obteniendo como respuesta SAT o UNSAT.

Una de sus funciones más interesantes en nuestro contexto es que, si se determina la satisfacibilidad de nuestro conjunto de restricciones, es capaz de producir valores concretos que cumplen dicha teoría, siendo dichos valores los ideales como nuevos candidatos a valores de entrada en nuestra ejecución concólica.

Para entender mejor las capacidades del Z3, veamos de forma rápida y con ejemplos [39] cómo usar sus características principales de forma sencilla usando como entrada SMT-LIB 2.0. Podremos escribir scripts como secuencias de comandos.

Para declarar constantes de un tipo determinado, usaremos el comando *declare-const* y para declarar funciones *declare-fun*.

Por ejemplo declaramos una constante *a*, y una función *f* que recibe un booleano y un entero y devuelve un entero.

```
(declare-const a Int)
(declare-fun f (Int Bool) Int)
```

Para poder hacer algo, es importante introducir nuestras fórmulas, que harán las veces de restricciones, con el comando *assert*. Una vez introducidas podremos comprobar si nuestro conjunto de declaraciones y asertos son satisfacibles. Añadiremos, por ejemplo, la restricción de que nuestra constante *a* sea menor que 10, y que *f* aplicado a *a* y *true* debe ser menor que 100.

```
(assert (> a 10))
(assert (< (f a true) 100))
(check-sat)
```

Como estas fórmulas pueden ser satisfechas, el Z3 nos devuelve la respuesta *sat*. En caso contrario, nos devolvería *unsat*, o incluso puede devolvernos *unknown* si es incapaz de determinar la satisfacibilidad de nuestro conjunto de declaraciones y aserciones.

Como nos ha devuelto *sat*, podemos usar el comando *get-model* para pedirle una interpretación que haga que todas las fórmulas sean *true*. Así, añadimos

```
(get-model)
```

y obtendríamos, al ejecutar, la respuesta:

```
sat
(model
  (define-fun a () Int
    11)
  (define-fun f ((x!1 Int) (x!2 Bool)) Int
    (ite (and (= x!1 11) (= x!2 true)) 0
```

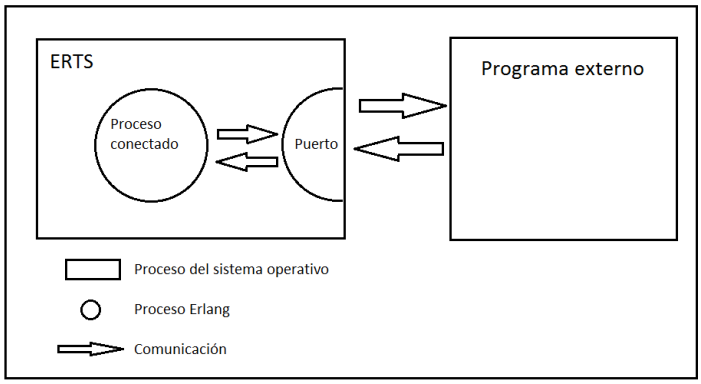
```
        0))

)
```

El modelo devuelto está basado en definiciones. La primera nos define que el valor de a para nuestro modelo será 11. La segunda está basada en una estructura *if-then-else*, abreviada como *ite*. Hemos definido los parámetros de entrada como x!1 y x!2. Para x!1 = 11 y x!2 = true, devolveremos 0. Para los otros casos, también. Para un modelo tan sencillo la respuesta es algo simple, pero cuando crece en complejidad se ve la auténtica potencia de la herramienta.

Para realizar la comunicación entre el resto del sistema Erlang y el módulo de resolución de restricciones será necesaria la implementación de un puerto que realice la comunicación entre ambos lenguajes. Erlang nos proporciona el mecanismo de los puertos para intercomunicar con otros lenguajes. Desde el lado Erlang, deberemos crear el puerto usando la función `open_port({spawn, ExtPrg},[Options]}.`

Con este método podremos enviar mensajes al puerto creado y así pasar información al otro programa.

Por supuesto, en el otro lado, sea el lenguaje que sea, deberemos preparar una interfaz capaz de interactuar con este tipo de intercambio de mensajes. El programa deberá leer la información que le llegue por la entrada estándar (el descriptor de fichero 0) y la salida estándar (el descriptor de fichero 1).



La principal limitación de los puertos es que no son capaces de enviar paquetes muy grandes de información, por lo que en muchas ocasiones se acaba reduciendo el intercambio de mensajes a mensajes de control y si debe haber paso de datos se produce mediante ficheros temporales o similares.

3.4 - Implementación: Ejecución concreta

Para ser capaces de generar una traza del programa, debemos instrumentar nuestro código previamente a su ejecución.

Nuestro *tracer* recibe como entrada una llamada inicial (la misma que le proporcionamos a todo el sistema de ejecución concólica). A partir de ahí, se consigue el nombre del módulo al que estamos llamando y se carga el archivo para modificarlo. Se lanzará el código modificado como un proceso aparte y el proceso de traza quedará a la espera de los mensajes de éste para construir la traza.

Para modificar el código utilizaremos la utilidad de Erlang `compile:file(File, Options)` que nos permite compilar un código fuente Erlang y modificarlo sobre la marcha de acuerdo a la función que le pasemos por *Options* si es de la forma `{parse_transform, Module}`. Nuestra función *parse_transform* recibirá el código en formato abstracto y deberá devolverlo en este mismo formato.

Para ver el aspecto y manipulación del código en formato abstracto podemos estudiar el uso de la función `epp:parse_file(FileName, Options)`. Está incluida en la biblioteca *epp* (Erlang code PreProcessor) [40].

Para nuestro uso, le pasaremos el nombre del archivo y nos devolverá, si todo ha ido bien, un `{ok, [Form]}`. En la variable *Form* tendremos cada expresión del código fuente que estemos analizando en formato abstracto.

Para ilustrar su funcionamiento veremos cómo responde ante un sencillo programa de ejemplo.

```
-module(hello).
-export([send_hello/1, receive_hello/0]).


send_hello(To) ->
 To ! {hello, self()}.
```

```
receive_hello() ->
 receive
  {hello, From} ->
   {ok, From};
  Msg ->
   {unknown_msg, Msg}
 end.
```

El árbol abstracto que obtendríamos de este programa es el siguiente.

```
[{attribute,1,file,{"../src/hello.erl",1}},
 {attribute,1,module,hello},
 {attribute,2,export,[{send_hello,1},{receive_hello,0}]},
 {function,4,send_hello,1,
     [{clause,4,
          [{var,4,'To'}],
          [],
          [{op,5,'!',
               {var,5,'To'},

{tuple,5,[{atom,5,hello},{call,5,{atom,5,self},[]}]}}]}]},
 {function,7,receive_hello,0,
     [{clause,7,[],[],
          [{'receive',8,
               [{clause,9,
                    [{tuple,9,[{atom,9,hello},{var,9,'From'}]}],
                    [],
                    [{tuple,10,[{atom,10,ok},{var,10,'From'}]}]},
                {clause,11,
                    [{var,11,'Msg'}],
                    [],
```

```
[{tuple,12,[{atom,12,unknown_msg},{var,12,'Msg'}]}]}]}]}]}]},
 {eof,13}]
```

Como se puede observar, pese a ser una representación menos intuitiva que el código fuente sigue siendo relativamente simple de entender para alguien con conocimiento sobre el lenguaje. Cada término, función, operador… del código es codificado en una tupla indicando su tipo, posición, valor y argumentos si los tuviera. Este lenguaje intermedio entre la fuente y el compilador es el que será modificado para añadirle el paso de mensajes con la información deseada al proceso *tracer*.

Recorreremos la información buscando todas las estructuras correspondientes a las cláusulas *case* o *receive*. Cuando se encuentren dichas sentencias se modificará cada una de sus opciones (o *clause*) para que, si la ejecución entra por ellas, envíen un mensaje a un proceso *tracer* que recogerá todos ellos y construirá la traza seguida por dicha ejecución teniendo en cuenta qué variable influye y cuáles eran los valores que hacían tomar uno u otro camino de ejecución.

Esta información de la traza será enviada de nuevo al módulo principal del programa en caso de estar utilizando el sistema al completo, si estamos utilizando el módulo de traza de forma independiente podremos imprimir los resultados por la salida estándar de una forma comprensible para el usuario.

Vamos a analizar en detalle el comportamiento del módulo *case_tracer*.

Ejecutaremos nuestro *tracer* con la llamada

```
1>case_tracer:trace("modulename:functionname(args)").
```

Usando la función *parse_call* que a su vez hace uso de las funciones *erl_scan:string*, *erl_parse:parse_term* y *erl_parse:parse_exprs* convertiremos una llamada en Erlang a una llamada en formato AST, y de ahí obtendremos el nombre del módulo.

Con el nombre del módulo podemos utilizar la función *compile:file* para compilar el módulo con el nombre que acabamos de obtener.

```
compile:file(atom_to_list(ModName)++".erl",
[{parse_transform,case_clause_sender_pt}])
```

Al meterle como argumento en las opciones una función de transformación, haremos que el compilador ejecute la modificación indicada en el *parse_transform*, del cual detallaremos su funcionamiento más adelante.

Es conveniente observar que el resultado del *compile:file* será un *ok* o un *error*, y en caso de *ok* se generará el ejecutable.

Registraremos el proceso *tracer* para poder llamándolo utilizando un átomo,

```
register(case_tracer,self())
```

y lanzaremos la llamada inicial sobre el módulo modificado y, ahora, compilado, añadiendo al final un mensaje de *stop* para que el *tracer* sepa cuándo termina la ejecución. Acto seguido crearemos la variable Trace, que será el resultado del bucle en el que esperaremos todos los mensajes del módulo modificado que acabamos de lanzar.

```
spawn(fun() -> erl_eval:expr(AExpr,[]),case_tracer!stop end),
Trace = lists:reverse(receive_loop(0,[])),
```

Para acabar, devolveremos la variable *Trace* o imprimiremos de forma comprensible la traza obtenida, según si queremos enlazarlo con un sistema más grande (como la ejecución concólica al completo) o tan solo visualizar la traza.

El funcionamiento del *receive_loop* es sencillo. Funcionará de forma recursiva, recibiendo en cada ejecución la traza anterior y cuántos elementos van registrados.

Esperará el mensaje de *stop* para detener la ejecución, o de eventos de traza, con los que generará la nueva traza y actualizará el recuento lanzando recursivamente de nuevo la función, por ejemplo:

```
receive

TraceItem = {send_trace, _} ->

                  receive_loop(Current + 1,
[{Current,TraceItem}|Trace]);
```

Y ahora veamos detalladamente el funcionamiento del módulo de transformación, empezando por las dos primeras funciones.

```
parse_transform(Forms,_) ->

      [erl_syntax_lib:map(

            fun case_clause_sender_fun/1,

            Form) || Form <- Forms].


case_clause_sender_fun(T) ->

      case_clause_sender_expr(erl_syntax:revert(T)).
```

Entraremos al modulo de transformación por la función `parse_transform/2`, que recibirá como argumento el árbol abstracto del programa, que guardaremos en la variable *Forms*. Para cada *Form* del conjunto de *Forms*, ejecutaremos la función `case_clause_sender_fun(T)`, que será la encargada de reformar el árbol de nuevo con el resultado de la trasformación que realizará *case_clause_sender_expr*.

Veamos primero el modificador de sentencias *case*:

```
case_clause_sender_expr({'case',LINE,E,Clauses}) ->

      PatternsClauses =

            [Pattern || {clause,_,Pattern,_,_} <- Clauses],

      {'case',LINE,E,change_clauses_case(Clauses,1,E,PatternsClauses,L
INE)};
```

Obtendremos todos los patrones y se los pasaremos al modificador de cláusulas junto con el resto de información de la cláusula.

Para el *receive* tendremos un comportamiento más complejo:

```
case_clause_sender_expr({'receive',LINE,Clauses}) ->

      PatternsClauses =

            [Pattern || {clause,_,Pattern,_,_} <- Clauses],

Nbody = [   {match, LINE, {var, LINE,'MessageQueue'},

              {call,LINE,{remote,LINE,

                 {atom,LINE,erlang},{atom,LINE,process_info}},

              [{call,LINE,

                 {atom,LINE,self},[]},{atom,LINE,messages}]}},

            {op,LINE,'!',

                {atom,LINE,'case_tracer'} ,

                 {tuple,LINE,

                       [{var,LINE,'MessageQueue'},

                       {call,LINE,{atom,LINE,self},[]}]

              }},

      {'receive',LINE,change_clauses_receive(Clauses,1,PatternsClauses
,LINE)}],

{block, LINE, Nbody};
```

Para cada bloque *receive*, añadiremos justo antes del comienzo el paso de un mensaje al proceso *tracer* con información sobre el Pid del proceso actual y la cola de mensajes del mismo. Pasaremos las diferentes cláusulas a *change_clauses* para instrumentarlas.

Cuando encontramos una operación de envío de mensajes entraremos por el siguiente código:

```
case_clause_sender_expr({op, LINE,'!',To,Message})->

      Nbody = [{op,LINE,'!',{atom,LINE,'case_tracer'},

            {tuple,LINE,[

                 {atom,LINE,'send_trace'},

               {tuple,LINE,[

                  {call,LINE,{atom,LINE,self},[]},
                  {string, LINE, lists:flatten(io_lib:format("~p !
~p", [To, Message]))},

            {call,LINE,{remote,LINE,
```

```
                    {atom,LINE,erlang},
                    {atom,LINE,process_info}},
                    [To,{atom,LINE,messages}]}
             ]}
  ]} },
              {op,LINE,'!',To,{tuple,LINE,
                   [{call,LINE,{remote,LINE,
                       {atom,LINE,case_tracer},
                       {atom,LINE,freshid}},[]},
                        Message]}}
       ],
{block,LINE,Nbody};
```

En este caso realizaremos dos modificaciones. Crearemos una instrucción de envío de mensaje a el *tracer* con datos sobre el envío realizado y la cola del receptor del mensaje, y además modificaremos el envío para añadir un identificador único a cada mensaje con la función *freshid()*.

Por último tendremos una versión para cualquier otra expresión que no entre en las anteriores, que dejaremos tal cual estaba.

```
case_clause_sender_expr(Other) ->
      Other.
```

Para modificar las cláusulas de dentro del *receive* o del *case* usaremos otra función, *change_clauses* o *change_clauses2*. Su funcionamiento es sencillo, recorren recursivamente las cláusulas de cada estructura y añaden el paso de un mensaje al proceso *tracer* para cada camino para así poder enviar información sobre el camino escogido. Para el caso del *case*:

```
change_clauses_case([],_,_,_,_) ->
      [];
change_clauses_case([{clause,LINE,Pattern,Guards,Body}|Clauses],Num,E
,PatternsClauses,CaseLine) ->
      NClauses = change_clauses(Clauses,Num +
1,E,PatternsClauses,CaseLine),
```

```
      NBody =
            [{op,LINE,'!',
                  {atom,LINE,'case_tracer'},
                   {tuple,LINE,[
                        {atom,LINE,'case_trace'},
                        {tuple,LINE,[
                              {integer,LINE,CaseLine},
                              {integer,LINE,Num},

      {string,LINE,lists:flatten(io_lib:format("~p",[E]))},

      {string,LINE,lists:flatten(io_lib:format("~p",[Pattern]))},

      {string,LINE,lists:flatten(io_lib:format("~p",[PatternsClauses])
)}
                        ]}
                  ]} }
             | Body],
      [{clause,LINE,Pattern,Guards,NBody}|NClauses].
```

Mientras queden cláusulas, añadiremos toda la información a un mensaje previo al resto del cuerpo original.

Para el caso del *receive*, además, deberemos modificar el patrón de cada cláusula para que acepte los mensajes instrumentados, tal que:

```
change_clauses_receive([],_,_,_)->
      [];
change_clauses_receive([{clause,LINE,Pattern,Guards,Body}|Clauses],Nu
m,PatternsClauses,CaseLine) ->
      NClauses = change_clauses2(Clauses,Num +
1,PatternsClauses,CaseLine),
      NBody =
            [{op,LINE,'!',
                  {atom,LINE,'case_tracer'},
```

```
                    {tuple,LINE,[
                         {atom,LINE,'receive_trace'},
                         {tuple,LINE,[
                               {integer,LINE,CaseLine},
                               {integer,LINE,Num},
                               {var,LINE,'MId'},

      {string,LINE,lists:flatten(io_lib:format("~p",[Pattern]))},

      {string,LINE,lists:flatten(io_lib:format("~p",[PatternsClauses])
)}
                         ]}
                    ]} }
              | Body],
              NPattern = [{tuple,LINE,[{var,LINE,'MId'},hd(Pattern)]}],
      [{clause,LINE,NPattern,Guards,NBody}|NClauses].
```

Como se puede ver, la principal diferencia es la creación del *NPattern* a partir del patrón original y de añadirle una variable *MId* delante, el identificador de mensaje que llegará como parte de la instrumentación de los mismos.

Con esto acaba el proceso de transformación, que instrumentará nuestro programa para producir una traza.

En la siguiente sección veremos algunos ejemplos de trazas con programas sencillos.

3.5 - Ejemplos

Para ver cómo funciona nuestra herramienta de trazado, veamos ejemplos de trazas sobre programas sencillos (sus códigos fuente se pueden ver entre los anexos) con valores de entrada elegidos al azar. Para facilitar su comprensión se mostrará la salida formateada por la función *pretty_printer/1* y no la variable *Trace* en su totalidad.

test.erl

El primer ejemplo lo veremos sobre este sencillo programa secuencial. Recibe tres variables y contiene un par de estructuras *case*, que modificarán los valores de entrada de diversas formas dependiendo del valor de los mismos. La traza generada es la siguiente:

```
1> case_tracer:trace("test:f(2,3,5)").

Trace registered from "test:f(2,3,5)"

*******************************

Event 0

Type case_trace line 8 related to "{var,8,'X'}"

Selected clause 2 == "[{integer,11,2}]"



Event 1

Type case_trace line 19 related to
"{op,19,'>',{var,19,'Y'},{integer,19,0}}"

Selected clause 1 == "[{atom,21,true}]"



Trace end.

ok
```

Como podemos ver, la traza registra los dos eventos *case*, su posición en el código, de qué expresión depende la decisión y el camino elegido.

hello.erl

En este pequeño programa probaremos la traza sobre dos procesos que tienen una conversación de mensajes. El proceso inicial lanzará al otro y le dirá *hello*, a lo que el otro responderá con *hi*. La traza generada es la siguiente:

```
1> case_tracer:trace("hello:hello()").

Trace registered from "hello:hello()"

******************************

Event 0

Type send_trace sent by <0.132.0> call "{var,6,'Pid'} !
{tuple,6,[{call,6,{atom,6,self},[]},{atom,6,hello}]}"

Target process queue []


Event 1

 Process <0.133.0> queue before receive []


Event 2

 Process <0.132.0> queue before receive []


Event 3

Type receive_trace line 13, message number 1

Selected clause 1 == "[{tuple,14,[{var,14,'Pid'},{atom,14,hello}]}]"


Event 4

Type send_trace sent by <0.133.0> call "{var,15,'Pid'} !
{atom,15,hi}"

Target process queue []


Event 5

 Process <0.133.0> queue before receive []
```

```
Event 6

Type receive_trace line 7, message number 2

Selected clause 1 == "[{atom,8,hi}]"


Trace end.

ok
```

Podemos ver como se mantiene la información sobre los mensajes pasados, y para cada cláusula *receive* obtenemos información del mensaje recibido y de qué patrón cumplía dicho mensaje.

También se observa en *Event 2* cómo pese a ya haber enviado el mensaje, al llegar a la primera sentencia *receive* su cola está vacía puesto que aun no ha llegado a su buzón.

testreceive.erl

Este último ejemplo, pese a seguir siendo sencillo y no hacer nada útil en realidad, pretende combinar las características de los anteriores, teniendo un *case* que modificará el valor de entrada, la creación de un proceso que pasará mensajes y envío de mensajes en cantidad para poder observar el comportamiento de la cola.

Su traza es la siguiente:

```
1> case_tracer:trace("testreceive:init(5)").

Trace registered from "testreceive:init(5)"

******************************

Event 0

Type case_trace line 6 related to "{var,6,'X'}"

Selected clause 3 == "[{var,11,'N'}]"


Event 1

Type send_trace sent by <0.55.0> call "{call,16,{atom,16,self},[]} !
{var,16,'NX'}"

Target process queue []
```

```
Event 2

Type send_trace sent by <0.56.0> call "{var,32,'Y'} ! {var,32,'X'}"

Target process queue []


Event 3

Type send_trace sent by <0.55.0> call "{call,17,{atom,17,self},[]} !
{op,17,'+',{var,17,'NX'},{integer,17,1}}"

Target process queue [{1,4}]


Event 4

Type send_trace sent by <0.55.0> call "{call,18,{atom,18,self},[]} !
{op,18,'+',{var,18,'NX'},{integer,18,2}}"

Target process queue [{1,4},{2,4},{3,5}]


Event 5

Type send_trace sent by <0.55.0> call "{call,19,{atom,19,self},[]} !
{op,19,'+',{var,19,'NX'},{integer,19,3}}"

Target process queue [{1,4},{2,4},{3,5},{4,6}]


Event 6

Type send_trace sent by <0.55.0> call "{call,20,{atom,20,self},[]} !
{op,20,'+',{var,20,'NX'},{integer,20,4}}"

Target process queue [{1,4},{2,4},{3,5},{4,6},{5,7}]


Event 7

Type send_trace sent by <0.55.0> call "{call,21,{atom,21,self},[]} !
{op,21,'+',{var,21,'NX'},{integer,21,5}}"

Target process queue [{1,4},{2,4},{3,5},{4,6},{5,7},{6,8}]


Event 8

Type send_trace sent by <0.55.0> call "{call,22,{atom,22,self},[]} !
{integer,22,2}"

Target process queue [{1,4},{2,4},{3,5},{4,6},{5,7},{6,8},{7,9}]
```

```
Event 9
 Process <0.55.0> queue before receive [{1,4},
                                        {2,4},
                                        {3,5},
                                        {4,6},
                                        {5,7},
                                        {6,8},
                                        {7,9},
                                        {8,2}]

Event 10
Type receive_trace line 24, message number 8
Selected clause 2 == "[{integer,26,2}]"


Trace end.
ok
```

Vemos como se captura el *case*, seguido del envío de múltiples mensajes entre los eventos 1 y 8, el estado previo de la cola antes del *receive* y cómo elegimos el mensaje marcado con el identificador 8 de la cola, que encaja en el patrón de la cláusula 2.

Capítulo 4 – Conclusiones

4.1 - Ampliaciones y trabajo futuro

Este trabajo fin de máster surge ante el interés por explorar las capacidades del lenguaje de programación Erlang, y la idea de que una técnica tan potente y popular en otros entornos como es la ejecución concólica pudiera ser aplicable también a Erlang para generar casos de prueba automáticamente, ya que ello podría redundar en una mejora de las aplicaciones desarrolladas en Erlang y en la producción de software de mayor calidad.

Aunque hemos planteado el esquema completo de un sistema de generación de pruebas basado en ejecución concólica en este trabajo, la implementación solo ha cubierto la parte del generador de trazas. El desarrollo e implementación del sistema completo es un trabajo que llevaría mucho más tiempo del disponible para un trabajo de este nivel y que se deja como trabajo futuro (no en vano es el objetivo de un proyecto del Plan Nacional con una duración de tres años). Pese a todo, la implementación realizada pretende servir de punto de partida para el desarrollo del sistema completo.

Durante la realización de este trabajo se preparó el artículo [41], que sienta las bases formales del proceso y que esperamos publicar en el futuro cercano.

Para poder realizar una ejecución simbólica, se requiere implementar un intérprete propio que nos permita trabajar con variables en vez de valores concretos y que sea capaz de ser guiado por la traza previamente generada durante la ejecución concreta. Esta implementación podría estar realizada sobre Erlang, pero no es estrictamente necesario que sea así (como alternativa, podría generarse un *modelo* que luego se podría tomar como entrada a un sistema de ejecución simbólica implementado en otro lenguaje de programación más apropiado, como por ejemplo Prolog).

En cualquier caso, se trata de un trabajo muy laborioso puesto que habría que recrear toda la semántica de Erlang en nuestro propio intérprete y además dotarlo de las capacidades necesarias para manejar variables simbólicas.

Para completar el sistema, habría que implementar una interfaz con algún resolutor de restricciones. Como ya se vio anteriormente, la opción mejor posicionada sería el uso del resolutor Z3 desarrollado por Microsoft, empleando los puertos de Erlang para la interconexión con un programa python que incluya las librerías Z3. Además, sería necesario codificar y decodificar las restricciones de Z3 en Erlang.

Por otro lado, cabe destacar que el sistema expuesto en este trabajo solo cubre los aspectos secuenciales de Erlang (aunque se acepten programas concurrentes como entrada). Así, las activaciones de diferentes procesos pueden variar de orden en ejecuciones distintas y, por ello, producir situaciones inesperadas en programas que, en principio, habríamos considerado libres de errores mediante la técnica de la ejecución concólica. Durante el desarrollo de este trabajo se tuvo en cuenta esta posibilidad, y aunque de forma totalmente conceptual, se planteó una posible solución basada en la combinación con la herramienta Concuerror. Dicha herramienta genera, para unos datos de entrada concretos, todos las posibles planificaciones (*interleavings* en inglés) que puede considerar la máquina virtual Erlang.

La idea es sencilla: como el fin último de la ejecución concólica es ejecutar una serie de casos de test elegidos adecuadamente para cubrir todo el programa, se podría generar una interfaz entre ambos sistemas que, para cada caso de test relevante generado por la ejecución concólica, se lanzara el mismo caso en Concuerror para analizar todos las posibles variantes y confirmar la estabilidad de su comportamiento o descubrir las potenciales variaciones.

Obviamente, dicho sistema podría ser muy costoso de ejecutar temporalmente para sistemas de cierto tamaño, si bien podría ser muy interesante para

sistemas o partes de sistemas críticos en los que el más mínimo fallo sea totalmente inadmisible ya que se alcanzaría un grado de cobertura tal que haría casi imposible que alguna ejecución futura no hubiera sido contemplada por la unión de ambos sistemas.

4.2 - Conclusiones

El lenguaje Erlang combina la concurrencia con las características de un lenguaje funcional, lo que lo hace muy expresivo. Pese a no ser un lenguaje idóneo como lenguaje de programación universal, se trata de una herramienta potente para tareas que requieran una alta capacidad de creación de procesos y escalabilidad.

La ejecución concólica se plantea como una alternativa con buenas perspectivas en el ámbito de las pruebas del software y la validación de programas. Así, se pueden obtener muy buenos resultados de cobertura reduciendo de forma significativa los problemas de la ejecución simbólica (principalmente la explosión de estados y la incapacidad de resolver algunas restricciones).

En esta memoria, se ha diseñado un sistema de ejecución concólica para un lenguaje funcional y concurrente como Erlang, empezando por la base formal, trabajando el diseño del sistema y se ha implementado el módulo encargado de realizar la ejecución concreta instrumentada para producir una traza de ejecución.

Es un sistema grande y complejo, y se ha tenido que estudiar en profundidad literatura diversa en varios ámbitos para asentar la base teórica necesaria para empezar a resolver el problema de partida. Las principales contribuciones de esta memoria se han resumido en [41].

Referencias

[1] J. C. King, “Symbolic Execution and Program Testing,” *Commun. ACM,* pp. 385-394, 1976.

[2] P. Godefroid, N. Klarlund and K. Sen, “DART: directed automated random testing,” *PLDI,* 2005.

[3] K. Sen, D. Marinov and G. Agha, “CUTE: a Concolic Unit Testing Engine for C,” *ESEC/SIGSOFT FSE,* pp. 263-272, 2005.

[4] M. Berkelaar, “lp_solve,” [Online]. Available: http://lpsolve.sourceforge.net/5.5/.

[5] “SGLIB,” [Online]. Available: http://sglib.sourceforge.net/.

[6] “Erlang home page,” [Online]. Available: http://www.erlang.org.

[7] F. Cesarini and S. J. Thompson, Erlang Programming, O'Reilly, 2009.

[8] C. Wikström, “Yaws,” [Online]. Available: http://yaws.hyber.org/.

[9] “Amazon SimpleDB,” [Online]. Available: http://aws.amazon.com/es/simpledb/.

[10] Process one, “Ejabberd HP,” [Online]. Available: https://www.process-one.net/en/ejabberd/.

[11] R. Reed, “WhatsApp scaling,” [Online]. Available: http://www.erlang-factory.com/upload/presentations/558/efsf2012-whatsapp-scaling.pdf.

[12] Y. Hara, “Erlang Central - Introducing Leofs,” [Online]. Available: http://www.erlang-factory.com/upload/presentations/744/leofs-erlang-factory-sfbay-2013.pdf.

[13] L. Ekeroth and P.-M. Hedström, “Ericsson.com,” [Online]. Available:

http://www.ericsson.com/ericsson/corpinfo/publications/review/2000_03/files/2000034.pdf.

[14] “Naos engine,” [Online]. Available: http://www.naos-engine.com/.

[15] “Vendetta Online,” [Online]. Available: http://www.vendetta-online.com/.

[16] “Demonware,” [Online]. Available: http://www.demonware.net/.

[17] M. Dowse, “Erlang Factory - Erlang and First Person Shooters,” 2011. [Online]. Available: http://www.erlang-factory.com/upload/presentations/395/ErlangandFirst-PersonShooters.pdf.

[18] F. Trottier-Hebert, “Learn you some Erlang,” [Online]. Available: www.learnyousomeerlang.com.

[19] “Erlang.org - Erl_syntax,” [Online]. Available: http://erlang.org/doc/man/erl_syntax.html.

[20] R. Taylor and J. Derrick, “Prowess: Property Based Testing of Web Services,” 2013.

[21] M. Widera, “Data Flow Coverage for Testing Erlang Programs,” *Erlang Workshop,* 2004.

[22] “Erlang cover tool,” [Online]. Available: http://www.erlang.org/doc/man/cover.html.

[23] T. Nagy and A. Nagyné, “Erlang testing and tools survey,” in *Erlang workshop*, 2008.

[24] R. Carlsson and M. Rémond, “Eunit: a lightweight unit testing framework for erlang,” *Erlang Workshop,* pp. 1-1, 2006.

[25] J. Blom and B. Jonsson, “Automated test generation for industrial erlang

applications," *Erlang Workshop,* pp. 8-14, 2003.

[26] U. Wiger, G. Ask and K. Boortz, "World-class product certificacion using erlang," *Erlang Workshop,* pp. 24-33, 2002.

[27] T. Arts, J. Hughes, J. Johansson and U. Wiger, "Testing telecoms software with quviq quickcheck," *Erlang Workshop,* pp. 2-10, 2006.

[28] H. Li and S. Thompson, "Tool support for refactoring functional programs," *PEPM: Symposium on Partial evaluation and semantics-based program manipulation,* pp. 199-203, 2008.

[29] L. Lövei, Z. Horváth, T. Kozsik and R. Király, "Introducing records by refactoring," *Erlang Workshop,* pp. 18-28, 2007.

[30] T. Lindahl and K. Sagonas, "Detecting software defects in telecom applications through lightweight static analysis: A war story.," *APLAS: Programming Languages and Systems: Proceedings of the Second Asian Symposium,* vol. 3302, pp. 91-106, 2004.

[31] L.-A. Fredlund and H. Svensson, "Mcerlang: a model checker for a distributed functional programming language," *ICFP: International conference on functional programming,* pp. 125-136, 2007.

[32] "Faxien," [Online]. Available: http://code.google.com/p/faxien.

[33] "Sinan," [Online]. Available: http://code.google.com/p/sinan.

[34] "Rebar HP," [Online]. Available: https://github.com/rebar/rebar.

[35] "Tsung," [Online]. Available: http://tsung.erlang-projects.org.

[36] S. Aronis, "Concuerror," 2014. [Online]. Available: www.concuerror.com.

[37] H. Svensson, L.-Å. Fredlund and C. B. Earle, "A unified semantics for future Erlang," in *Erlang Workshop*, 2010.

[38] L. M. de Moura and N. Bjørner, “Z3: An Efficient SMT Solver,” in *TACAS*, 2008.

[39] Microsoft Research, “Rise4fun,” 2014. [Online]. Available: www.rise4fun.com/Z3/tutorial/guide.

[40] “Erlang.org - Epp,” [Online]. Available: http://www.erlang.org/doc/man/epp.html.

[41] O. Coll, A. Palacios, S. Tamarit and G. Vidal, “Concolic Testing in Erlang,” 2014.

[42] C. S. Pasareanu, “A survey of new trends in symbolic execution for software testing and analysis,” *STTT,* vol. 11, no. 4, pp. 339-353, 2009.

[43] P. Godefroid, M. Y. Levin and D. A. Molnar, “Automated Whitebox Fuzz Testing,” *NDSS,* 2008.

[44] J. Calleja, “Ricston blog,” 2013. [Online]. Available: www.ricston.com/blog.

[45] G. Vidal, “Concolic Execution and Test Case Generation in Prolog,” in *LOPSTR: 24th International Symposium on Logic-Based Program Synthesis and Transformation*, 2014.

[46] G. Vidal, “Symbolic Execution in Erlang,” in *PSI: Ershow Informatics Conference*, 2014.

Anexos

Código fuente

case_tracer.erl

```
-module(case_tracer).


-export([trace/1, pretty_printer/1, newid/1, freshid/0,
parse_call/1]).


trace(Call) ->

      register(ser,spawn(case_tracer,newid,[1])),


      InitialCall = parse_call(Call),

      AExpr =

            case is_list(InitialCall) of

                  true ->

                        hd(parse_expr(InitialCall++"."));

                  false ->

                        InitialCall

            end,

      {call,_,{remote,_,{atom,_,ModName},_},_} = AExpr,

      %io:format("~p\n~p\n",[AExpr,ModName]),

      compile:file(atom_to_list(ModName) ++
".erl" ,[{parse_transform,case_clause_sender_pt}]),

      %Return = compile:file(atom_to_list(ModName) ++
".erl" ,[{parse_transform,case_clause_sender_pt}]),

      %io:format("Return: ~p\n",[Return]),

      register(case_tracer,self()),

      try

            erlang:purge_module(ModName)
```

```
      catch
            _:_ -> ok
      end,
      code:load_abs(atom_to_list(ModName)),
      spawn(fun() -> erl_eval:expr(AExpr,[]),case_tracer!stop end),
      Trace = lists:reverse(receive_loop(0,[])),
      unregister(case_tracer),
      unregister(ser),
      io:format("Trace registered from
~p~n**********************************~n",[Call]),
      pretty_printer(Trace).
      %Trace.


receive_loop(Current,Trace) ->
      receive
            stop ->
                  Trace;
            TraceItem = {case_trace, _} ->
                  receive_loop(Current + 1,
[{Current,TraceItem}|Trace]);


            TraceItem = {send_trace, _} ->
                  receive_loop(Current + 1,
[{Current,TraceItem}|Trace]);


            TraceItem = {{messages, _},_} ->
                  receive_loop(Current + 1,
[{Current,TraceItem}|Trace]);


            TraceItem = {receive_trace, _} ->
                  receive_loop(Current + 1,
[{Current,TraceItem}|Trace])
```

```
      end.



parse_expr(Func) ->
    case erl_scan:string(Func) of
            {ok, Toks, _} ->
                case erl_parse:parse_exprs(Toks) of
                        {ok, _Term} = Res ->
                            Res;
                        _Err ->
                            {error, parse_error}
                end;
            _Err ->
                {error, parse_error}
    end.


parse_call(ExprString) ->
    case erl_scan:string(ExprString ++ ".") of
            {ok, Toks, _} ->
                case erl_parse:parse_term(Toks) of
                        {ok, Term} ->
                            Term;
                        _Err ->
                              case erl_parse:parse_exprs(Toks) of
                                    {ok, [Term]} ->
                                          Term;
                                    _Err ->
                                    {error, parse_error}
                            end
                end;
            _Err ->
```

```
                {error, parse_error}
    end.


newid(N) ->
    receive
    {getid, Pid} ->
    Pid ! {freshid,N},
                        newid(N+1);
      _ -> newid(N)
    end.


freshid() ->
    ser ! {getid, self()},
    receive
      {freshid,Id} ->
        Id
    end.


pretty_printer([]) ->
      io:format("Trace end.~n");
pretty_printer([{NEvent,{{messages,Queue},Pid}}|OtherEvent]) ->
      io:format("Event ~p~n ", [NEvent] ),
      io:format("Process ~p queue before receive ~p~n~n",
[Pid,Queue]),
      pretty_printer(OtherEvent);
pretty_printer([{NEvent, {TEvent = receive_trace,{Line,
NClauseselected,NMess, Clauseselected,_Allclauses}}}|OtherEvent]) ->
      io:format("Event ~p~n", [NEvent] ),
      io:format("Type ~p line ~p, message number ~p~n", [TEvent, Line,
NMess]),
      io:format("Selected clause ~p == ~p~n~n",
[NClauseselected,Clauseselected]),
      pretty_printer(OtherEvent);
```

```
pretty_printer([{NEvent, {TEvent = case_trace,{Line, NClauseselected,
Expression, Clauseselected,_Allclauses}}}|OtherEvent]) ->

      io:format("Event ~p~n", [NEvent] ),

      io:format("Type ~p line ~p related to ~p~n", [TEvent, Line,
Expression]),

      io:format("Selected clause ~p == ~p~n~n",
[NClauseselected,Clauseselected]),

      pretty_printer(OtherEvent);

pretty_printer([{NEvent, {TEvent,{Pid,Call,{messages,
Queue}}}}|OtherEvent]) ->

      io:format("Event ~p~n", [NEvent] ),

      io:format("Type ~p sent by ~p call ~p~n", [TEvent, Pid, Call]),

      io:format("Target process queue ~p ~n~n", [Queue]),

      pretty_printer(OtherEvent);

pretty_printer([{NEvent, {TEvent,{Line, _}}}|OtherEvent]) ->

      io:format("Event ~p~n", [NEvent] ),

      io:format("Type ~p line ~p ~n", [TEvent, Line]),

      pretty_printer(OtherEvent).
```

case_clause_sender_pt.erl

```
-module(case_clause_sender_pt).


-export([parse_transform/2,ref_append/1]).

-export([case_clause_sender_expr/1]).


ref_append(File) ->

      {ok, Forms} = epp:parse_file(File, [], []),

    Comments = erl_comment_scan:file(File),

    NForms = parse_transform(Forms,[]),

    lists:map(fun(Form) -> io:format("~p\n",[Form]) end, NForms),

    _FinalForms = erl_recomment:recomment_forms(NForms,Comments).

    %io:format("~s\n",[erl_prettypr:format(FinalForms)]).
```

```
parse_transform(Forms,_) ->
      NForms = [erl_syntax_lib:map(
            fun case_clause_sender_fun/1,
            Form) || Form <- Forms],
      %lists:map(fun(Form) -> io:format("~p\n",[Form]) end, NForms),
      NForms.


case_clause_sender_fun(T) ->
      case_clause_sender_expr(erl_syntax:revert(T)).



case_clause_sender_expr({'case',LINE,E,Clauses}) ->
      PatternsClauses =
            [Pattern || {clause,_,Pattern,_,_} <- Clauses],
      {'case',LINE,E,change_clauses_case(Clauses,1,E,PatternsClauses,L
INE)};


case_clause_sender_expr({'receive',LINE,Clauses}) ->
      PatternsClauses =
            [Pattern || {clause,_,Pattern,_,_} <- Clauses],
      Nbody = [
      {match, LINE, {var,
LINE,'MessageQueue'},{call,LINE,{remote,LINE,{atom,LINE,erlang},{atom
,LINE,process_info}},[{call,LINE,{atom,LINE,self},[]},{atom,LINE,mess
ages}]}},
      {op, LINE,
'!',{atom,LINE,'case_tracer'},{tuple,LINE,[{var,LINE,'MessageQueue'},
{call,LINE,{atom,LINE,self},[]}]}},
      {'receive',LINE,change_clauses_receive(Clauses,1,PatternsClauses
,LINE)}],
            {block, LINE, Nbody};


case_clause_sender_expr({op, LINE,'!',To,Message})->
```

```
      Nbody =

            [

            {op,LINE,'!',

                  {atom,LINE,'case_tracer'},

                   {tuple,LINE,[

                        {atom,LINE,'send_trace'},

                        {tuple,LINE,[

                              {call,LINE,{atom,LINE,self},[]},

                              {string, LINE,
lists:flatten(io_lib:format("~p ! ~p", [To, Message]))},


      {call,LINE,{remote,LINE,{atom,LINE,erlang},{atom,LINE,process_in
fo}},[To,{atom,LINE,messages}]}

                        ]}

                   ]} },


{op,LINE,'!',To,{tuple,LINE,[{call,LINE,{remote,LINE,{atom,LINE,case_
tracer},{atom,LINE,freshid}},[]},Message]}}



       ],

       %io:format("~nNBODY ~p",[Nbody]),

       {block,LINE,Nbody};


case_clause_sender_expr(Other) ->

      Other.




change_clauses_case([],_,_,_,_) ->

      [];

change_clauses_case([{clause,LINE,Pattern,Guards,Body}|Clauses],Num,E
,PatternsClauses,CaseLine) ->

      NClauses = change_clauses_case(Clauses,Num +
1,E,PatternsClauses,CaseLine),

      NBody =
```

```
            [{op,LINE,'!',
                  {atom,LINE,'case_tracer'},
                   {tuple,LINE,[
                        {atom,LINE,'case_trace'},
                        {tuple,LINE,[
                              {integer,LINE,CaseLine},
                              {integer,LINE,Num},

      {string,LINE,lists:flatten(io_lib:format("~p",[E]))},

      {string,LINE,lists:flatten(io_lib:format("~p",[Pattern]))},

      {string,LINE,lists:flatten(io_lib:format("~p",[PatternsClauses])
)}
                        ]}
                   ]} }
             | Body],
      [{clause,LINE,Pattern,Guards,NBody}|NClauses].


change_clauses_receive([],_,_,_)->
      [];
change_clauses_receive([{clause,LINE,Pattern,Guards,Body}|Clauses],Nu
m,PatternsClauses,CaseLine) ->
      NClauses = change_clauses_receive(Clauses,Num +
1,PatternsClauses,CaseLine),
      NBody =
            [{op,LINE,'!',
                  {atom,LINE,'case_tracer'},
                   {tuple,LINE,[
                        {atom,LINE,'receive_trace'},
                        {tuple,LINE,[
                              {integer,LINE,CaseLine},
                              {integer,LINE,Num},
                              {var,LINE,'MId'},
```

```
      {string,LINE,lists:flatten(io_lib:format("~p",[Pattern]))},

      {string,LINE,lists:flatten(io_lib:format("~p",[PatternsClauses])
)}
                        ]}
                  ]} }
            | Body],
            NPattern = [{tuple,LINE,[{var,LINE,'MId'},hd(Pattern)]}],
      [{clause,LINE,NPattern,Guards,NBody}|NClauses].
```

testreceive.erl

```
-module(testreceive).
-export([init/1,sender/2]).


init(X)->
      NX =
            case X of
                  1 ->
                        0;
                  2 ->
                        1;
                  N ->
                        N - 1
            end,
      spawn(testreceive,sender,[NX,self()]),
      self()!NX,
      self()!NX+1,
      self()!NX+2,
      self()!NX+3,
```

```
      self()!NX+4,
      self()!NX+5,
      self()!2,
      receive
            1 -> one;
            2 -> two;
            3 -> three
            %; _ -> another
      end.

sender(X,Y) ->
      Y ! X.
```

test.erl

```
-module(test).

-export([f/3 ]).

f(X,Y,Z) ->
      % N = 3,
      NX =
            case X of
                  1 ->
                        0;
                  2 ->
                        1;
                  N ->
                        N - 1
                  % ;
                  % _ ->
```

```
                  %     N - 2
            end,
      case Y > 0 of
            true -> Y - NX;
            false ->
                  case Z of
                        1 -> NX - Y;
                        _ -> NX + Y
                  end
      end.
```

hello.erl

```
-module(hello).
-export([hello/0, hi/0]).


hello()->
      Pid = spawn(hello, hi, []),
      Pid ! {self(),hello},
      receive
            hi ->
                  bye
            end.


hi()->
      receive
            {Pid, hello} ->
                  Pid ! hi,
                  hi()
            end.
```