UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

# Diseño de una nueva herramienta para la depuración de código Erlang

**TRABAJO FIN DE GRADO**

Grado en Ingeniería Informática

*Autor:* José Luis Jiménez García

*Tutor:* Germán Vidal Oriola

Curso 2015-2016

# Resum

Erlang és un llenguatge de programació funcional que integra facilitats per a la concurrència. A diferència de la majoria de llenguatges concurrents, Erlang seguix el model basat en pas de missatges, aconseguint així que la creació de processos siga més eficient que en altres llenguatges concurrents. Per això mateixa, Erlang s'ha convertit en un llenguatge cada vegada més popular, emprant-se en la implementació dels aspectes concurrents d'aplicacions com Twitter o Whatsapp.

En l'actualitat, el llenguatge de programació Erlang ja disposa d'un cert nombre de ferramentes de depuració. En particular, podem trobar des d'una ferramenta de depuració basada en traces i "break points", a altres més avançades basades en les tècniques de "model checking" o la generació de casos de prova mitjançant execució simbòlica. En este projecte, es pretén dissenyar e implementar una ferramenta amb interfície gràfica que combine algunes de les característiques dels diferents models de depuració, la qual cosa esperem que done lloc a una nova ferramenta que resulte d'interés per a desenvolupadors de distints nivells de codi Erlang.

**Paraules clau:** Depuració, Validació, Software, Punts de trencada, Erlang, Metaprogramación, Llenguatge funcional, Pas de missatges.

# Resumen

Erlang es un lenguaje de programación funcional que integra facilidades para la concurrencia. A diferencia de la mayoría de lenguajes concurrentes, Erlang sigue el modelo basado en paso de mensajes, consiguiendo así que la creación de procesos sea más eficiente que en otros lenguajes concurrentes. Por ello, Erlang se ha convertido en un lenguaje cada vez más popular, empleándose en la implementación de los aspectos concurrentes de aplicaciones como Twitter o Whatsapp.

En la actualidad, el lenguaje de programación Erlang ya dispone de un cierto número de herramientas de depuración. En particular, podemos encontrar desde una herramienta de depuración basada en trazas y "break points", a otras más avanzadas basadas en las técnicas de "model checking" o la generación de casos de prueba mediante ejecución simbólica. En este proyecto, se pretende diseñar e implementar una herramienta que combine algunas de las características de los diferentes modelos de depuración, lo que esperamos que dé lugar a una nueva herramienta que resulte de interés para los desarrolladores de código Erlang.

**Palabras clave:** Depuración, Validación, Software, Puntos de ruptura, Erlang, Meta programación, Lenguaje funcional, Paso de mensajes.

# Abstract

Erlang is a functional programming language that integrates facilities to concurrency. Unlike most concurrent languages, Erlang follows the message passing model, consequently delivering a more efficient process creation compared to other concurrent languages. Thus, Erlang has become increasingly popular, being employed for the implementation of the concurrent aspects of applications like Twitter or Whatsapp.

Currently, Erlang developers already have a number of debugging tools available. In particular, one can find debugging tools based on traces and break points, but also some more advanced tools based on model checking techniques or test case generators based on symbolic execution. The aim of this project is to design and implement a new GUI-based tool that combines several features from different debugging models. We hope it will result in a new useful tool for Erlang developers.

**Key words:** Debugging, Validation, Software, Breakpoints, Erlang, Metaprogramming, Functional language, Message Passing.

# Índice general

# Índice de figuras

# CAPÍTULO 1
# Introducción

El *Software* ha ido evolucionando a lo largo de la historia, pasando de estar en unos pocos dispositivos a estar presente en casi cualquier dispositivo que nos rodea, desde el control del flujo del agua que reciben nuestras viviendas hasta el control del número de kilovatios que reciben nuestras casas. Gracias a la combinación del *Software* con otras otras disciplinas, se ha permitido enriquecer éstas, generando increíbles avances que hace unos años hubiesen sido inimaginables. Algunas de las disciplinas que han sido beneficiadas son, por ejemplo, la Medicina, Aeroespacial y Arquitectura.

El campo de la Medicina, por ejemplo, uno de los últimos avances que se obtuvieron, fue la capacidad de operar a un ser humano que se encuentra en un país distinto del que realiza la cirugía. Esta operación se realiza mediante un robot, que recibe el nombre de "Da Vinci", controlado de forma remota por el cirujano. La precisión y sincronización de este robot con el operario es increíble. Gracias a este avance, se podrán operar personas en países que no disponen de los recursos necesario para poder mantener un cirujano de esa especialidad.

Dentro de la disciplina Aeroespacial, tenemos muchos avances, pero uno de los que más impacto ha tenido por parte de los medios de comunicación es el "Curiosity". Curiosity es un robot tipo rover, enviado a Marte con el objetivo de recabar información sobre el planeta rojo. Gracias a esta información, podremos realizar nuevos avances en todas las disciplinas. Además, ya se hablan de la posibilidad de aprovechar sus recursos naturales o incluso de habitar el planeta.

La realidad aumentada está teniendo una gran acogida dentro del ámbito de la Arquitectura. Desde comerciales que enseñan a sus clientes cómo va a quedar su vivienda una vez esté construida, hasta simuladores de mecánica de fluidos. Los simuladores de mecánica de fluidos, como por ejemplo IBER, permiten estudiar el comportamiento de construcciones cuando están sometidas a la presión del agua. Con este tipo de simuladores, se pueden llegar a hacer mejores construcciones, y evitan la pérdida tanto de vidas humanas como de miles de millones a las empresas.

Sin embargo, debido a la cantidad de *Software* de que disponemos, también aumenta la necesidad tanto de garantizar la calidad como la fiabilidad del *Software*, ya que un error dentro del *Software* puede causar muchas perdidas. Además, siempre existe la necesidad de revisar el proceso de desarrollo del *Software*, mediante el uso de nuevas metodologías [1, 2]. Un ejemplo de este tipo de errores es el que tuvo la empresa de inversiones Knight Capital.

Knight Capital durante la mañana de agosto de 2013, mientras sus agentes realizaban sus operaciones con normalidad, el sistema Knight Capital tuvo un error de repente. El error perduró durante aproximadamente media hora hasta que sus operarios se percataron. El error consiste en que el *software*, en un momento dado, comenzó a comprar acciones de empresas sin ningún tipo de control. Nada más percatarse del error, los operarios comenzaron a vender las acciones para intentar recuperar algo del dinero invertido. El coste directo en cuanto a las pérdidas para la empresa fue de casi 500 millones de euros, sin tener en cuenta que las acciones de la empresa bajaron un 63 % , la pérdida de credibilidad y fiabilidad.

Por ello, cada vez es mayor la inversión de las empresas en la certificación y validación del software. Uno de los métodos más utilizados en el proceso de validación es la técnica de depuración [3]. La técnica de depuración es un aspecto importante dentro de la ingeniería del software, ya que nos permite identificar y corregir algunos errores dentro de la fase de programación de un software. En algunos casos, la depuración consigue también optimizar el código fuente, aunque éste no sea su objetivo principal.

Existen varios tipos de ejecución [4], cómo formas de realizar las trazas [5] para el código fuente que trata el depurador. Además, existen otro tipo de técnicas para validar el código fuente, como por ejemplo, "model cheking" [6, 7] y "concolic testing"[8].No obstante, uno de los más utilizados dentro del campo de la depuración es la ejecución estática. La ejecución estática, a diferencia de otros tipos de ejecución, necesita de valores de entrada introducidos por el usuario. Cuando un depurador ejecuta un programa para hacer una simulación, utiliza los valores de entrada del usuario como variables estáticas. Así, es capaz de reducir el número de nodos generados y el espacio ocupado en memoria por la simulación. Otra característica de la ejecución estática es el uso de entornos y procesos donde se anotan las sentencias de control del programa. Las anotaciones le sirven al depurador para saber el flujo de control del programa. Además, dispone de entornos que le permiten recuperar el contexto de las variables, en caso de que el flujo de control lo requiera.

Actualmente el equipo de desarrollo de Erlang presenta, en su paquete oficial, el depurador *DBG*. La semántica de *DBG* se evalúa de forma estática, al igual que la presentada en este proyecto. Sin embargo, es una herramienta muy compleja, ya que requiere de un alto nivel de conocimientos sobre depuración. La herramienta que hemos desarrollado dispone de una interfaz gráfica y se centra en los aspectos más comunes de la depuración, lo cual proporciona un mayor grado de

usabilidad.

La organización de la memoria se divide en seis capítulos. Éste es el primer capítulo que sirve de motivación para la propuesta. El segundo capítulo trata de transmitir algunos conceptos claves sobre el lenguaje de programación Erlang, con el que se desarrolló el proyecto. Entre esos conceptos se encuentran nociones básicas en cuanto a la sintaxis, concurrencia y patrones de diseño. El tercer capítulo introduce unos conceptos básicos sobre metaprogramación, compilación en Erlang y sintaxis. El cuarto capítulo es la propuesta formal del proyecto desde el diseño de la aplicación, hasta la implementación de la herramienta desarrollada en este proyecto. En este capítulo se pueden encontrar cómo se realiza la evaluación estática para cada una de las instrucciones asociadas a un código fuente, introducido por el usuario. Además, en este capítulo se describe el uso de los puntos de ruptura. Cuando un usuario realiza una depuración sobre un código fuente puede apreciar una serie de resultados. El usuario puede recibir e interactuar sobre esos resultados mediante el uso de la interfaz gráfica. El quinto capítulo incluye unas breves conclusiones del proyecto realizado, así como algunas propuestas para continuar las investigación del proyecto.

# CAPÍTULO 2
# Erlang/OTP

## 2.1 El lenguaje Erlang

Erlang es un lenguaje funcional orientado a la concurrencia. Este nombre le fue dado en honor a A. K. Erlang, un científico cuyas aportaciones destacan en el área de las telecomunicaciones. Erlang nació en el seno de los laboratorios Ericsson, antes de que la empresa llegase a fusionarse con Sony. Ericsson tenía varios problemas de mantenimiento y desarrollo de aplicaciones, para las cuales usaba lenguajes concurrentes.

El principal problema que tenía era que, para realizar el mantenimiento de sus aplicaciones, debían parar la ejecución de sus aplicaciones. Este problema se debía a que los lenguajes convencionales de esa época no permitían la ejecución de código ininterrumpido. Por ese motivo, era necesario un lenguaje que permitiera realizar modificaciones de código fuente en ejecución, sin afectar al resto de los nodos de la red. Otro problema al que se enfrentaron era que, si un nodo fallaba, podía generar una propagación de errores, haciendo que el resto de los nodos dejará de funcionar. Por lo tanto, era necesario que el lenguaje siguiera una política de tolerancia a fallos especial. Es decir, a diferencia de otros lenguajes, Erlang sigue una política de (tolerancia a fallos) en la que si un nodo tiene un error, deja que falle y a continuación el nodo se reinicia. Por ello, no se suele hacer control de errores a menos que sea estrictamente necesario. Además, cabe destacar que en esa época los dispositivos tenían memorias de apenas unos *KiloBytes* de espacio. Por lo tanto, era necesario disponer de un lenguaje que proporciona algunas de las ventajas que tienen C/C++, Prolog o Smalltalk, pero que ocupara una cuarta parte de lo que ocuparía en estos lenguajes. Una vez identificaron los problemas, se pusieron en marcha y consiguieron crear este lenguaje. A finales de los 90 pasa de ser un "código cerrado" a ser "código abierto", permitiendo a la comunidad científica disponer de su código fuente.

Una de las virtudes de Erlang es que es posible integrarlo fácilmente en otros lenguajes como Python, Java, C++ o JavaScript [9]. Sólo hay que generar un pequeño modulo que haga de interfaz. La facilidad de integración es tal, que empresas como Facebook, Google e IBM, que buscan solucionar sus problemas en las comunicaciones (similares a los de Ericsson), vieron en Erlang su solución. Aun-

que, como contrapartida, la dificultad en cuanto al aprendizaje de este lenguaje es alta, especialmente para los programadores formados en lenguajes imperativos u orientados a objetos.

Dentro de Erlang podemos observar dos conceptos básicos importantes. El primero es BEAM, la máquina virtual que ejecuta el código precompilado de Erlang. Esta máquina virtual es la encargada de interpretar y ejecutar las diferentes instrucciones del código fuente, cumpliendo con la semántica del lenguaje. El segundo es OTP, las bibliotecas estándar que facilitan la implementación de las aplicaciones. Para esta aplicación solo se utilizaron las librerías *compiler*, *wx* y *set* de OTP. Existe la posibilidad de añadir nuevas funciones a la biblioteca mediante el uso del lenguaje C.

Si entramos un poco más en detalle sobre las librerías OTP[10], podemos observar una serie de características:

- *Productividad*: Este *framework* está implementado con algoritmos centrados en la velocidad de procesamiento, intentando evaluar las diferentes expresiones lo más rápido posible.
- *Estabilidad*: El sistema necesita control sobre los procesos mediante el uso de procesos de control y máquinas de estado.
- *Supervisión*: La aplicación se estructura de manera simple, facilitando la supervisión y el estado del sistema de manera automática o mediante el uso de opciones gráficas.
- *Capacidad de actualización*: El *framework* implementa patrones específicos para facilitar la actualización del código.
- *Código base estable*: Es un lenguaje declarado "sólido", ya que fue minuciosamente probado.

Como se puede observar en la figura 2.1, una aplicación que está compuesta por distintos procesos se ejecuta en una máquina virtual. Existen lenguajes que no realizan un gran esfuerzo en aprovechar al máximo el número de núcleos de la estación de trabajo. Sin embargo, Erlang intenta que, de manera automática, cada vez que lanzamos una aplicación, ésta se divida en distintas tareas, para así poder aprovechar al máximo el número de núcleos. Esto se consigue agrupando los distintos tipos de tareas en función de sus dependencias. Una vez agrupadas, Erlang intenta distribuirlas entre los diferentes núcleos disponibles en la estación de trabajo. Un ejemplo de este proceso se puede apreciar en la figura 2.2, donde se puede ver cómo intenta distribuir las diferentes tareas entre los diferentes núcleos disponibles. Al distribuir los diferentes procesos en tareas y éstos entre los

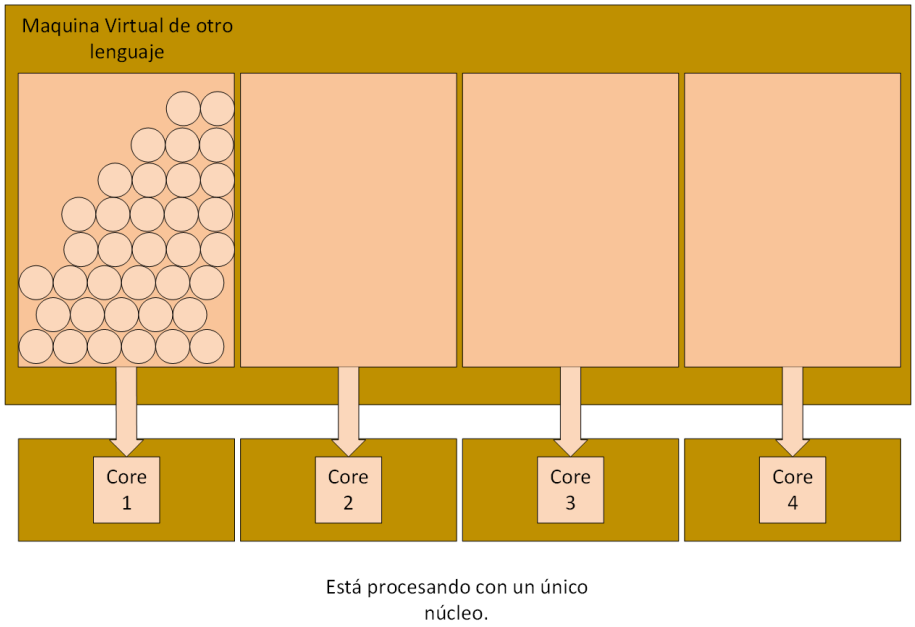


**Figura 2.1:** Ejemplo de máquina virtual de otro lenguaje de programación.

cores, se consigue aumentar la velocidad de la aplicación de una manera considerable.

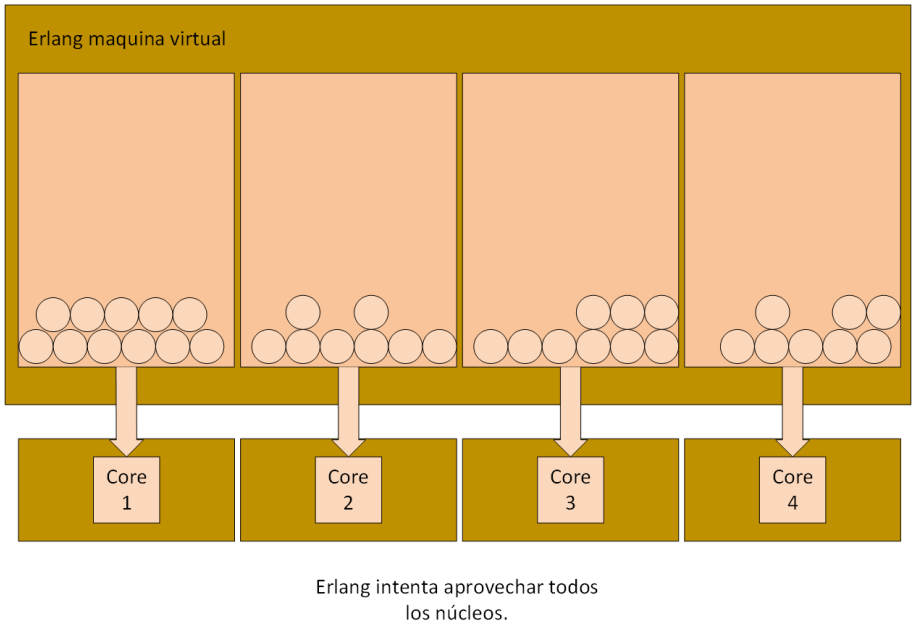


**Figura 2.2:** Ejemplo de máquina virtual del lenguaje Erlang.

## 2.2 Conceptos básicos Erlang

Existen diferentes tipos de ficheros en Erlang[11] y que se diferencian por la extensión del archivo. La extensión más frecuente, en la que vamos a centrarnos,

es ".erl" (la extensión para la elaboración de módulos). Existen otros formatos como ".hrl", que son archivos de inclusión, en donde se suelen incorporar macros y se suelen incluir en varios módulos. Otro uso de estos archivos de inclusión es para definir estructuras de datos. Estas estructuras se podrán exportar simplemente importando el archivo de inclusión.

Como muestra en el programa *ejemploCalculadora.erl* se puede observar una estructura mínima de un módulo en Erlang. Este ejemplo nos ayudará a comprender la sintaxis del lenguaje:

- El símbolo " %", se utiliza para generar los comentarios. De esta forma el desarrollador da información complementaria sobre el código fuente. Este elemento es esencial para futuros desarrolladores, ya que permite clarificar el funcionamiento de un bloque de código.

- La instrucción "-module()ütiliza como parámetro el nombre del fichero sin extensión. Dicha instrucción, tiene como funcionalidad dar más información al compilador de código Erlang.

- La instrucción "-export()recibe como parámetro una lista '[]', con una serie de elementos, que reciben el nombre de átomos y están separados por una coma. Todos los átomos que se encuentren en esta lista son los mismos átomos literales de funciones creadas dentro del módulo. Si no se pone el átomo que haga referencia a un método dentro del export, el método no será visible fuera del módulo. Los átomos de la lista del export incluyen una barra y un número. Ese número representa la cantidad de parámetros que recibe dicha función (es decir, su aridad).

```
1  %%%-------------------------------------------------------------------
2  %%%Esto es un comentario
3  %%%-------------------------------------------------------------------
4  -module(calculadora).
5  -export([
6          suma/2,
7          resta/2,
8          multiplicacion/2,
9          division/2]).
10
11 suma(A, B) ->
12     A + B.
13
14 resta(A, B) ->
15     A - B.
16
17 multiplicacion(A, B) ->
18     A * B.
19
20 division(A, B) ->
21     if
22       B > 0 ->
```

```
23          A / B;
24        true ->
25          0
26    end.
```

**Listing 2.1:** Archivo *ejemploCalculadora.erl*.

La estructura de una función en el lenguaje Erlang[12] es las siguiente:

- *Nombre*: Es un átomo literal que sirve como identificador de la función, intentando siempre ser lo más identificativo posible. Por ejemplo *nombreFuncion*.

- *Parámetros*: Son las variables que van a recibir su valor desde fuera de la función. Por ejemplo *nombreFuncion(Param1, Param2, ParamN)*.

- La flecha especifica el inicio del bloque de instrucciones que va ejecutar la función.

- En algunas ocasiones es necesario utilizar sentencias de todo tipo, en este caso un 'if'. La sentencia tiene un identificador que cuando se utiliza hace referencia al inicio del bloque de código. A continuación, comienza el bloque de código que se anida a esa sentencia. Por último, se utiliza la sentencia 'end' que especifica la terminación del bloque de instrucciones que ha sido anidado.

- El '.' especifica el final del bloque de instrucciones constituye el cuerpo de la función.

Como puede intuirse, existen varios tipos de sentencia. Podemos ver otro ejemplo en el archivo *ejemploSemaforo.erl*, en donde se utilizan otros dos conceptos muy importantes de Erlang. El primero de ellos es el *pattern matching*[13]. Cuando ejecutamos código fuente se generará un proceso. Este proceso dispone a su vez de una cola enlazada asociada. En la cola enlazada, cuando hay una llamada de *pattern matching*, se desencola cada uno de los elementos de la llamada, empezando por el nombre de la función y siguiendo por los parámetros de la misma, buscando el mismo patrón dentro del código. Por ejemplo, si Colores vale 'rojo', N devolverá 'parar'. Otra forma de hacer *pattern matching*es mediante el uso de del simbolo ';'. Si nos fijamos, podemos apreciar que los métodos se llaman igual y reciben el mismo número de parámetros. En el caso en que no hará *matching* con el primer método o el segundo, hará *matching* con el tercero. En la segunda opción del 'case', la variable que le precede intentara hacer *matching* con todas las opciones que preceden a 'of'. La flecha indicará, como siempre, el inicio de ejecución de bloque y el ';' que le precede, un caso nuevo. La etiqueta 'end' al igual que en la sentencia 'id' indica el final del 'case', dando a entender a la máquina virtual que no hay más casos que evaluar.

```
-module(semaforo).
-export([estados/1]).

estados({error, Otro}) ->
  'Error inesperado: ' ++ Otro;

estados({remoto, Otro}) ->
  'Error remoto: ' ++ Otro;

estados(Colores) ->
  case Colores of
  {rojo, N} when N >= 20, N =< 40 ->
                        'parar';
  {amarillo, N} when N >= 10, N < 20 ->
                        'prudencia';
  {verde, N} when N >= 0, N < 10 ->
                        'adelante';
  _ ->
                        'Error en el estado'
  end.
```

**Listing 2.2:** Archivo *ejemploSemaforo.erl*.

## 2.3 Concurrencia

Un concepto importante dentro de Erlang es la concurrencia[14]. La concurrencia se consigue en Erlang a través del uso de procesos. La velocidad de creación de estos procesos es del orden de microsegundos, e independiente del número de procesos. Pero estos procesos no funcionan de una manera tradicional, sino que se ejecutan con procesos independientes del sistema operativo. En la concurrencia basada en el modelo de actores, cada proceso dispone de su propio espacio en memoria y su propia pila en lugar de un espacio compartido en memoria. Pero estos procesos no pueden interferir unos con otros sin tener los permisos adecuados. Si esto no fuera así, podrían producirse una infinidad de errores, como por ejemplo *deadlock*.

Para poder realizar la comunicación entre los diferentes procesos, se utiliza el concepto de paso de mensajes[15]. Esos mensajes son asíncronos, por lo que los procesos pueden seguir realizando las tareas sin tener que preocuparse de esos mensajes. Además, los mensajes no necesitan un tipo de dato específico, haciendo más fácil el proceso de transmisión de mensajes entre los diferentes procesos. Los procesos disponen de una cola de entrada en donde se irán almacenando los diferentes mensajes. Esta cola no está ordenada, pero la recogida de la cola la hace por proceso selectivo, por lo que se minimiza el problema. Realmente los problemas residen más en el entorno de ejecución a nivel de red que en el sistema en sí mismo.

En Erlang podemos diferenciar tres aspectos básicos dentro de la concurrencia:

1. *Creación de procesos*: Consiste en utilizar la instrucción *spawn(Mod, Func, Args)*, tal y como se muestra en la figura 2.3. Esta instrucción, cuando es ejecutada, devuelve un Pid (Process Identifier), que es el nombre asociado a un proceso (un ejemplo sería <0,63,2 >). En cuanto a los parámetros de la función, disponemos de un primer parámetro que representa donde se encuentra la función y qué va a ejecutar el proceso. El segundo parámetro hace referencia a la función que va ejecutar el proceso y se encuentra dentro del módulo especificado. Dicha función puede requerir una serie de parámetros, que se especificarán dentro de una lista, representada en *Args*. En el caso de que no se requieran parámetros, será necesario introducir el valor lista vacía '[]'. La instrucción base también dispone de una variante para especificar el nodo en donde se encuentra el módulo. Si se diera el caso de que el módulo no existiera, entonces el proceso se crearía. Sin embargo, este proceso reportaría un error.

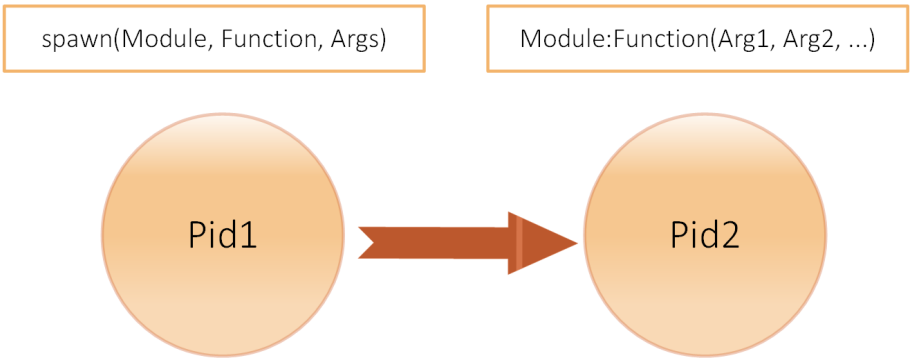


**Figura 2.3:** Creación de un nuevo proceso.

2. *Paso de mensajes*: Una vez disponemos de diferentes procesos, es necesario poder transmitir información entre ellos. Cuando se crea un proceso, automáticamente se crea un buzón para sus mensajes. Ese buzón de entrada consiste en una cola enlazada, donde se van encolando los mensajes según el proceso los va recibiendo. El método de envío de un mensaje entre distintos procesos consiste en el operador '!' . Si nos fijamos en la figura 2.4, podemos observar un ejemplo en el que el proceso Pid1 envía un mensaje al proceso Pid2. El operador '!' necesita dos parámetros. En primer lugar, a su lado izquierdo necesita disponer del Pid del proceso al cual queremos enviar el mensaje. En segundo lugar, a su derecha, el mensaje. El mensaje ha de ser encapsulado usando una estructura de datos (por ejemplo, una tupla).

3. *Recepción de mensajes*: Además de enviar, también hay una forma de recibir los mensajes en Erlang en un proceso determinado. Si observamos la figura 2.5, Pid1 envió un mensaje a Pid2. Ahora es el turno de que Pid2 reciba el mensaje, mediante el uso de la instrucción receive. El funcionamiento de esta instrucción consiste en hacer *pattern matching*con el mensaje. En este ejemplo hará *pattern matching* con la primera posición porque cumple la condición. Los dos mensajes utilizan el mismo átomo llamado 'envio'.

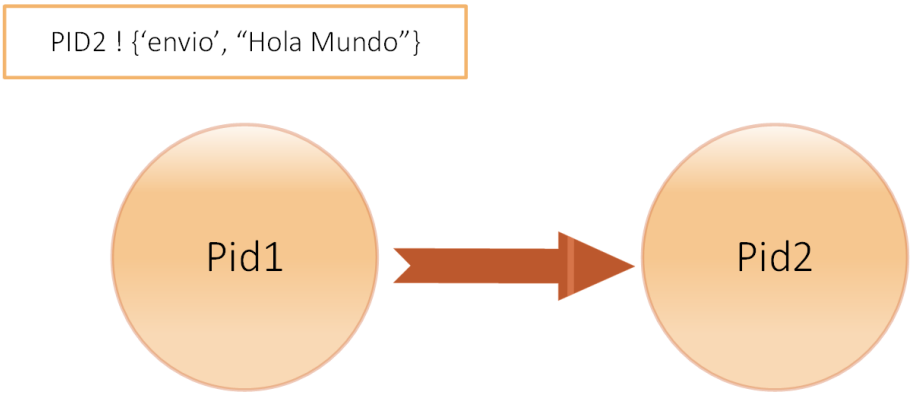


**Figura 2.4:** Envió de un mensaje entre procesos.

Muchos desarrolladores añaden un patrón, ilustrado en este ejemplo, que recibe el nombre de sumidero. Consiste en usar 'Other', que es una variable que puede ser cualquier tipo, por lo que siempre hará *matching* con cualquier mensaje. El motivo de su implementación es, sobre todo, para que no se pierdan mensajes, aunque también puede usarse para la depuración y control de errores.

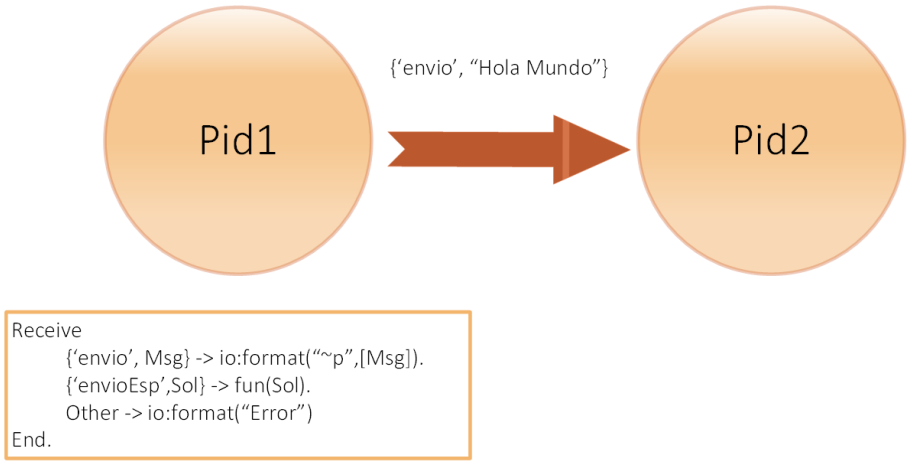


**Figura 2.5:** Recepción de un mensaje de un proceso.

## 2.4 Patrones de diseño

Dentro de OTP podemos encontrar un concepto básico: El árbol de supervisión, formado por trabajadores y supervisores [16]. Los trabajadores son procesos encargados de la lógica de la aplicación y los supervisores del correcto funcionamiento de los trabajadores. Un supervisor tiene la potestad de reiniciar uno o varios trabajadores en el caso de detectar un mal funcionamiento.

A la hora de implementar software utilizando el modelo trabajador-supervisor, podemos observar una serie de patrones diseño. Estos patrones de diseño en este modelo reciben el nombre "Behaviour"[17]. "Behaviour" es un módulo que in-

cluye una serie de funciones que intenta abstraer el comportamiento de diferentes roles dentro del nivel de comunicación mediante el uso de servicios. La utilización de patrones también hace que sea más fácil leer y entender el código escrito por otros programadores. Sólo es necesario incluir su directiva en el programa y añadir un átomo que represente el rol del servicio. Además, hay que exportar las diferentes funciones del servicio que van a ser utilizadas. Casi todas las funciones creadas en dichos módulos son *callbacks*. Los diferentes roles soportados son los siguientes:

1. *Gen_server*: Para implementar el modelo cliente-servidor. El modelo de cliente-servidor se caracteriza por un servidor central y un número arbitrario de clientes. El modelo cliente-servidor se utiliza para las operaciones de gestión de recursos, donde varios clientes diferentes quieren compartir un recurso común. El servidor se encarga de gestionar este recurso.

2. *Gen_fsm*: Para la ejecución de máquinas de estados finitos. Una máquina de estados finitos (FSM) se puede describir como un conjunto de relaciones de la forma:

$$Estado(S)xEvento(E) -> Acciones(A), Estado(S') \tag{2.1}$$

   Estas relaciones se interpretan en el sentido de izquierda a derecha. Siguiendo este ejemplo, "si estamos en el estado S y el evento E ocurre, vamos a realizar acciones A y transitar al estado S'. Para un FSM implementado utilizando el comportamiento *gen_fsm*, las reglas de transición de estado se escriben como una serie de funciones de Erlang.

3. *Gen_ event*: Para disponer la funcionalidad de gestión de eventos. En OTP, un gestor de eventos es un objeto con nombre al que los eventos predefinidos se pueden suscribir y enviar notificaciones al gestor. Un evento puede ser, por ejemplo, un error, una alarma, o algún evento prediseñado que va a ser conectado. El gestor de eventos puede disponer o no de controladores de eventos. Cuando el gestor de eventos se notifica acerca de un evento, el evento es procesado por todos los controladores de eventos instalados. Por ejemplo, un administrador de eventos de errores de manipulación puede por defecto tener un controlador instalado que escribe mensajes de error en el terminal. Los mensajes de error se guardan en un archivo temporal durante un periodo de tiempo. Esto le da tiempo al usuario a instalar otro controlador si ve que es necesario. Cuando el registro en el archivo ya no es necesario, se elimina este controlador de eventos. Un gestor de eventos se implementa como un proceso y cada controlador de eventos se implementa como un módulo de devolución de llamada. El gestor de eventos mantiene esencialmente una lista de módulo, estados pares, donde cada módulo es un controlador de eventos, y el estado es el estado interno de ese controlador de eventos.

4. *Supervisor*: Un supervisor es responsable de iniciar, detener, y realizar el seguimiento de sus procesos hijos. El objetivo principal de un supervisor es la de mantener a sus procesos hijo vivos reiniciando cuando sea necesario. El supervisor genera una lista con los hijos y una serie de características esenciales de los mismos. Los procesos hijo se inician en el orden indicado por la lista, y se terminan en el orden inverso.

```
-module(serverMemoria).
-behaviour(gen_server).

-export([init/1, handle_call/3, handle_cast/2, handle_info/2, terminate
    /2]).

-export([start/0, insertVal/2, insertVal/1, returnVal/1, deleteVal/1,
    exeMemoria/0 , updateVal/2]).

start() ->
  gen_server:start_link({local, ?MODULE}, ?MODULE, [], []).

updateVal(Name, Value) ->
  gen_server:call(?MODULE, {updateVal, Name, Value}).
insertVal(Name, Value) ->
  gen_server:call(?MODULE, {insertVal, Name, Value}).
insertVal(Name) ->
  gen_server:call(?MODULE, {insertVal, Name, notUse}).
returnVal(Variable) ->
  gen_server:call(?MODULE, {returnVal, Variable}).
deleteVal(Variable) ->
  gen_server:call(?MODULE, {deleteVal, Variable}).
exeMemoria() ->
  gen_server:call(?MODULE, {memori}).


init([]) ->
  Memoria = dict:new(),
  {ok, Memoria}.


noUsaVal([T|R], _A) ->
  case T of
    {Name, [notUse]} ->
      _Asub2 = lists:merge([_A,[{Name, notUse}]]),
      noUsaVal(R, _Asub2);
    {Name, notUse} ->
      _Asub2 = lists:merge([_A,[{Name, notUse}]]),
      noUsaVal(R, _Asub2);
    true ->
      noUsaVal(R, _A)
  end;
noUsaVal([],_A)->
_A.

handle_call({memori}, _From, Memoria) ->
  ListVals = dict:to_list(Memoria),
  Response = noUsaVal(ListVals,[]),
```

```
47    {reply, Response, Memoria};
48
49  handle_call({insertVal, Name, Value}, _From, Memoria) ->
50    Response = case dict:is_key(Name, Memoria) of
51      true ->
52        NewStateMemoria = Memoria,
53        {already_checked_out, Value};
54      false ->
55        NewStateMemoria = dict:append(Name, Value, Memoria),
56        ok
57    end,
58    {reply, Response, NewStateMemoria};
59
60  handle_call({updateVal, Name, Value}, _From, Memoria) ->
61    Response = case dict:is_key(Name, Memoria) of
62      true ->
63        NewStateInnerMemoria = dict:erase(Name, Memoria),
64        NewStateMemoria = dict:append(Name, Value, NewStateInnerMemoria),
65        ok;
66      false ->
67        NewStateMemoria = dict:append(Name, Value, Memoria),
68        ok
69    end,
70    {reply, Response, NewStateMemoria};
71
72
73  handle_call({returnVal, Name}, _From, Memoria) ->
74    Response = case dict:is_key(Name, Memoria) of
75      true ->
76        {Name, lists:nth(1, dict:fetch(Name, Memoria))};
77      false ->
78        {not_checked_out, Name}
79    end,
80    {reply, Response, Memoria};
81
82  handle_call({deleteVal, Book}, _From, Memoria) ->
83    NewStateMemoria = dict:erase(Book, Memoria),
84    {reply, ok, NewStateMemoria};
85
86  handle_call(_Message, _From, Memoria) ->
87    {reply, error, Memoria}.
88
89
90  handle_cast(_Message, Memoria) ->
91    {noreply, Memoria}.
92  handle_info(_Message, Memoria) ->
93    {noreply, Memoria}.
94  terminate(_Reason, _Memoria) ->
95    ok.
```

**Listing 2.3:** Archivo *serverMemoria.erl*.

Hemos visto diferentes patrones de comportamiento sobre una estructura en árbol basada en supervisión. En este proyecto, el patrón utilizado es el gen_server que utiliza el modelo cliente servidor. Con el objetivo de aclarar cuál es el comportamiento de un proceso estándar de servidor en Erlang, se ha desarrollado un servidor que implementa dicho patrón. Para ello, lo primero que hemos de hacer es introducir la directiva y un átomo para que cargue las diferentes funciones.

El átomo de este ejemplo es gen server. Estas etiquetas disponen ya de funcionalidades creadas para simplificar el proceso de trabajo en servidores. Tampoco es necesario añadir ningún tipo de argumento auxiliar a la hora de compilar el servidor. Se van a omitir las líneas estándar de una aplicación hecha en Erlang. En la línea dos se ve cómo usamos la directiva que hace referencia a los patrones para el árbol, en la que se le pasa un átomo indicando el tipo de comportamiento seleccionado deseado.

Se han utilizado dos directivas export, para diferenciar las que son funciones del behaviour de las que son del módulo. A continuación, vamos a proceder a explicar cómo funciona cada una de estas callbacks:

- *Init*: Esta función se ejecuta de manera asíncrona cuando se ejecuta la función start link() .
- *Start_link*: El principal uso de esta función es generar la estructura con la que va trabajar el proceso.
- *Handle_call*: Cuando un proceso comunique con él y éste espere una respuesta. Esta *callback* responde al proceso que hace la llamada, por lo que actualizara su estructura y notificará al proceso.
- *Handle_cast*: Funciona exactamente igual que el call pero no responde al proceso, sólo actualizará su estructura.
- *Handle_info*: Transmitir información al servidor.
- *Terminate*: Finaliza el proceso.

El primer proceso *serverMemoria.erl* se encarga de representar un entorno en memoria, utilizando los conocimientos mostrados en el punto anterior. Este entorno almacena el estado de las variables que se ejecutan dentro de una función. En el fichero *worker.erl* se encuentra un cliente que hace uso de este entorno, para un proceso en concreto. Vamos a centrarnos en qué hace el fichero *worker.erl*. Este programa inicializa un proceso de gestión de memoria, mediante la función *init* y le dice qué variables van a ser utilizadas en ese entorno. A continuación, existe un método *update*, que se encargará de simular las diferentes iteraciones dentro de un problema. Cada iteración, tendrá una repercusión sobre las variables actualizando su valor. El método *getNotUse* solicita al servidor que le indique qué variables no tienen valor aún en su entorno. Estas variables no van a ser utilizadas por la función y pueden ser eliminadas. Como se puede observar en el fichero, en el último método *fin*, se puede solicitar al proceso que vacíe su memoria, puesto que ya no son necesarias sus variables.

```
-module(worker).

-export([init/0,info/1, getDep/0, update/0, getNotUse/0,
    getMemoriaCompleta/0, fin/2]).

info(QUE)->
  serverMemoria:info(QUE).
init() ->
  serverMemoria:start(),
  serverMemoria:insertVal("A1"),
  serverMemoria:insertVal("A2"),
  serverMemoria:insertVal("A3"),
  serverMemoria:insertVal("A4"),
  serverMemoria:insertVal("A5").

update() ->
  %%Iteracion 1
  serverMemoria:updateVal("A1", "a+1"),
  serverMemoria:updateVal("A3", "a-6"),
  %%Iteracion 2
  serverMemoria:updateVal("A1", "a+6"),
  serverMemoria:updateVal("A3", "a+1"),
  serverMemoria:updateVal("A5", "A1"),
  %%Iteracion 3
  serverMemoria:updateVal("A1", "A2"),
  serverMemoria:updateVal("A2", "A3-3"),
  serverMemoria:updateVal("A3", "Zero"),
  serverMemoria:updateVal("A5", "a+2").

getNotUse() ->
  Record = serverMemoria:exeMemoria(),
  Record.
getDep() ->
  Record = serverMemoria:exeMemoriaDepdent(),
  Record.
getMemoriaCompleta() ->
  Record = serverMemoria:exeMemoriaCompleta(),
  Record.

fin(_Reason, _Memoria) ->
  serverMemoria:terminate(_Reason, _Memoria).
```

**Listing 2.4:** Archivo *worker.erl*.

# CAPÍTULO 3
# Metaprogramación

## 3.1 Conceptos básicos de metaprogramación

Actualmente existen una enorme cantidad de lenguajes de programación. Cada uno de estos lenguajes de programación tiene una serie de virtudes dependiendo del entorno en donde trabajen. Estas virtudes son las que animan a los desarrolladores a elegir un lenguaje de programación u otro cuando desean crear una aplicación software. Por ejemplo, si un desarrollador está interesado en hacer un robot tal vez le interese más el lenguaje C, o si desea hacer una aplicación distribuida tal vez le interese más usar el lenguaje Erlang. Estos lenguajes están desarrollados para una serie de objetivos concretos, en lo que se refiere a su aplicabilidad sobre esos entornos. Es por ello que en algunas ocasiones los desarrolladores que los utilizan se ven limitados por el mismo.

```
1 -module(factorial).
2
3 -export([fact/1]).
4
5 %!test1
6 fact(0) ->
7     1;
8 fact(N) when N > 0 ->
9     fact(N-1).
```

**Listing 3.1:** Archivo *factorial.erl*.

La metaprogramación tiene como objetivo principal modificar otros programas. Algunos de los usos de la metaprogramación son el testing o para realizar coberturas [**?**] en un código fuente. Para ello, en el comportamiento disponemos de varias opciones en el caso del lenguaje Erlang:

- Modificar el código fuente: En el ejemplo *factorial.erl* se muestra un ejemplo de este concepto. El programador aprovecha los comentarios para modificar su comportamiento. El símbolo del porcentaje representa un comentario, mientras que el símbolo '!' que le precede, es el que advierte al API o su programa. Finalmente, a estos dos símbolos les precede la directiva que es el comportamiento deseado por el desarrollador. El comportamiento, por

ejemplo, puede ser una macro para que ejecute una prueba.

- Realizar un paso intermedio: El objetivo de este método es tomar un código fuente y transformarlo en un formato intermedio. A continuación, sólo es necesario recorrer la estructura y modificar los elementos que necesite el desarrollador. Esas modificaciones las realizan mediante un programa diseñado para ese caso. Finalmente, se pasa de ese estado intermedio a un estado final para la posterior ejecución por parte de la máquina virtual.

- Mediante el uso de las anotaciones: Actualmente, cada vez son más los desarrolladores que hacen uso de la meta programación. Por ello, los creadores de los lenguajes de programación intentan dar facilidades a los desarrolladores de aplicaciones que hacen uso de la meta programación. En el caso de Erlang, existe el uso de las anotaciones. La anotaciones son un espacio reservado en la estructura interna de cada instrucción que se pueden aprovechar para insertar las diferentes directivas. Estas directivas se interpretan antes de la ejecución para modificar el comportamiento original. Existen algunos API como *Cerl* en el caso de Erlang que facilitan el acceso a las anotaciones para los desarrolladores.

## 3.2 Conceptos básicos compilador

El compilador es una pieza fundamental en Erlang para la herramienta de depuración, ya que el depurador utiliza los mismos tipos de datos que el compilador y trabajan al servicio de la máquina virtual. Además, es necesario que el código que se simula en la herramienta de depuración tenga el mismo comportamiento que tendría si se compila y ejecuta en la máquina virtual. El compilador suele trabajar con distintos tipos de archivos. En el caso de Erlang, utiliza hasta seis distintos. Nosotros sólo vamos a centrarnos en cuatro de ellos, ya que son los investigados para la realización de la propuesta. Si observamos la figura 3.1 vemos que el primero de ellos '.erl' es el formato utilizado para crear los módulos en Erlang. El segundo es el formato Abstracto que es utilizado en aplicaciones como Dialyzer. El siguiente es el formato Core que es un formato más cercano al compilador y genera menos información redundante. Y el último de ellos es el formato binario, que es ejecutado por la máquina virtual.

Se puede convertir de un tipo de fichero a otro facilitando la metaprogramación. El compilador de Erlang dispone de dos opciones interesantes. La primera opción es una interfaz, que recibe el nombre de *compiler*, que nos permite transformar de '.erl' al resto de formatos. Además, dispone de un pequeño manejador de excepciones que nos puede dar algo de información de las condiciones del archivo. La segunda opción es utilizar primitivas *lib_chunk*: estas librerías nos permiten pasar de estados intermedios a binarios. Esta opción es muy utilizada por los desarrolladores que necesitan instrumentar código fuente.

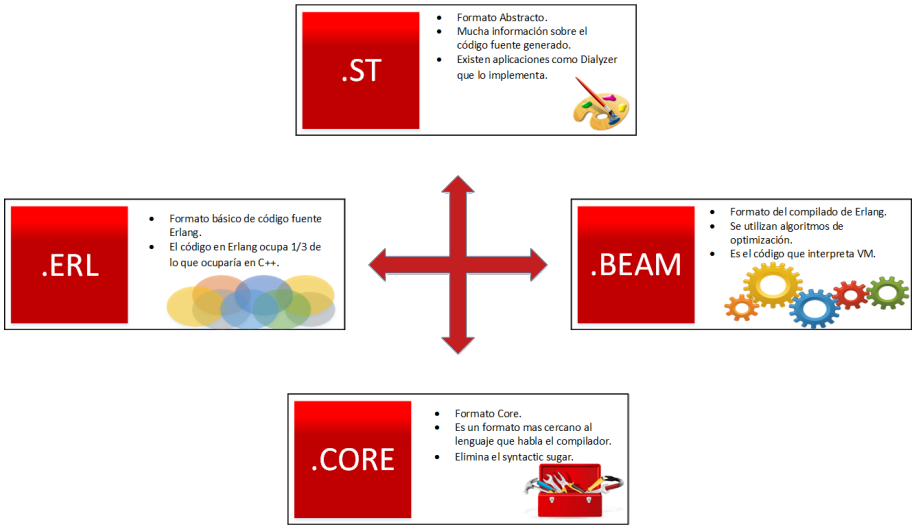


**Figura 3.1:** Ejemplo de tipos de formatos en Erlang.

```
1 -module(ejemploAbstract1).
2 -export([ejemploFunc/0]).
3
4 ejemploFunc() ->
5   X = 4,
6   Y = 5,
7   C = X + Y.
```

**Listing 3.2:** Archivo *ejemploAbstract1.erl*.

Vamos a usar un ejemplo, para ver cómo realizar las transformaciones entre los distintos tipos de formatos usando la interfaz *compiler* o primitivas:

- Transformación de modulo base a formato abstracto. Un ejemplo de instrucción para la conversión es:

$$\{ok, Abstract\} = epp : parse\_file(File, [], []) \tag{3.1}$$

*Epp* es una interfaz con llamadas al compilador que se encarga de transformar el formato base a formato abstracto. Sólo necesita de un átomo que represente el archivo a transformar. Sin embargo, dispone de dos parámetros de configuración de tipo lista. El primero es para especificar la ruta del archivo. El segundo, para establecer la configuración asociada a las macros. Si tuvo éxito la conversión devolverá una tupla, en donde su primer elemento es el identificador 'ok' y el segundo, el archivo transformado. En caso de no poder transformarlo por algún motivo, devolverá una tupla con el identificador 'error' junto con una variable de *razon*. La variable *razon* describiría el tipo de error que se ha producido impidiendo la transformación. Recuperando el archivo *ejemploAbstract1.erl* y ejecutando la sentencia el resultado de la conversión aparece en *salidaAbstract1.txt*.

```
{ok,[{attribute,1,file,{"ejemploAbstract1.erl",1}},
     {attribute,1,module,ejemploAbstract1},
     {attribute,2,export,[{ejemploFunc,0}]},
     {function,4,ejemploFunc,0,
                [{clause,4,[],[],
                         [{match,5,{var,5,'X'},{integer,5,4}},
                          {match,6,{var,6,'Y'},{integer,6,5}},
                          {match,7,{var,7,'C'},{op,7,'+',{var,...},
                              {...}}}]}]},
     {eof,7}]}
```

**Listing 3.3:** Archivo *salidaAbstract.txt*.

- Transformación de modulo base a formato Core. Un ejemplo de instrucción para la conversión es:

$$\{ok, NameFile, Core\} = compile : file(File, [to\_core, binary]) \tag{3.2}$$

La interfaz *compile* sirve facilitar la comunicación de nuestro código fuente mediante el compilador. Utilizando la opción *to_core* y especificando un fuente, nos devolverá una tupla. El primer parámetro es un identificador que puede ser 'ok' o 'error'.El segundo parámetro, el nombre del módulo transformado. Finalmente, en el último elemento tendríamos la transformación de código fuente a código core. Si ejecutamos la instrucción dentro de la máquina virtual pasando el fichero *ejempoCore1.erl* y mostramos el contenido de la variable Core, la transformación del código fuente aparece en el archivo *salidaCore1.txt*.

```
-module(ejemploCore1).
-export([ejemploFunc/2]).

ejemploFunc(X, Y) ->
  X + Y,
  Y + 5.
```

**Listing 3.4:** Archivo *ejemploCore1.erl*.

Si lo que queremos es generar el código binario desde un código fuente base, podemos usar el compilador por defecto. Esta instrucción genera un binario resultante de la compilación. El binario, será utilizado por la máquina virtual para simular la ejecución del fuente.

```
  {ok,ejemploCore1,
                    {c_module,[],
                     {c_literal,[],ejemploCore1},
                     [{c_var,[],{ejemploFunc,2}},
                      {c_var,[],{module_info,0}},
                      {c_var,[],{module_info,1}}],
                     [],
                     [{{c_var,[],{ejemploFunc,2}},
                       {c_fun,
                        [4,{file,"ejemploCore1.erl"}],
                        [{c_var,[4,{file,"ejemploCore1.erl"}],cor1},
```

```
12                                {c_var,[4,{file,"ejemploCore1.erl"}],cor0}],
13                               {c_seq,[],
14                                {c_call,
15                                 [5,{file,"ejemploCore1.erl"}],
16                                 {c_literal,[5,{file,"ejemploCore1.erl"}],
                                       erlang},
17                                 {c_literal,[5,{file,[...]}],'+'},
18                                 [{c_var,[4,{...}],cor1},{c_var,[4|...],cor0}
                                       ]},
19                                {c_call,
20                                 [6,{file,"ejemploCore1.erl"}],
21                                 {c_literal,[6,{file,[...]}],erlang},
22                                 {c_literal,[6,{file,...}],'+'},
23                                 [{c_var,[4|...],cor0},{c_literal,[...],...}
                                       ]}]}],
24                              {{c_var,[],{module_info,0}},
25                               {c_fun,
26                                [0,{file,"ejemploCore1.erl"},
                                    compiler_generated],
27                                [],
28                                {c_call,
29                                 [0,{file,"ejemploCore1.erl"},
                                     compiler_generated],
30                                 {c_literal,
31                                  [0,{file,"ejemploCore1.erl"},
                                      compiler_generated],
32                                  erlang},
33                                 {c_literal,
34                                  [0,{file,"ejemploCore1.erl"},
                                      compiler_generated],
35                                  get_module_info},
36                                 [{c_literal,
37                                   [0,{file,[...]},compiler_generated],
38                                   ejemploCore1}]}]}},
39                              {{c_var,[],{module_info,1}},
40                               {c_fun,
41                                [0,{file,"ejemploCore1.erl"},
                                    compiler_generated],
42                                [{c_var,
43                                  [0,{file,"ejemploCore1.erl"},
                                      compiler_generated],
44                                  cor0}],
45                                {c_call,
46                                 [0,{file,"ejemploCore1.erl"},
                                     compiler_generated],
47                                 {c_literal,
48                                  [0,{file,"ejemploCore1.erl"},
                                      compiler_generated],
49                                  erlang},
50                                 {c_literal,
51                                  [0,{file,"ejemploCore1.erl"},
                                      compiler_generated],
52                                  get_module_info},
53                                 [{c_literal,
54                                   [0,{file,...},compiler_generated],
55                                   ejemploCore1},
56                                  {c_var,[0,{...}|...],cor0}]}]}]}}
```

**Listing 3.5:** Archivo *salidaCore1.txt*.

Ahora vamos a ver el uso de primitivas para realizar transformaciones de binario a formato Abstracto o Core. Estas primitivas están recogidas en el módulo*beam_lib*.

Un ejemplo de uso de estas primitivas sería el siguiente:

$$ok, \_, [abstract\_code, \_, AC] = beam\_lib : chunks(Beam, [abstract\_code]). \quad (3.3)$$

Mediante el uso de esta función, tras pasar un módulo en formato binario, nos devolvería su transformación en abstracto. La única condición que hay que tener en cuenta es que el binario previamente ha de estar compilado con el *flag debug_inf*. Según está recogido en la documentación de la librería, esto es así por seguridad.

Mediante el la función *fwrite* y el módulo *erl_syntax* podemos escribir un nuevo módulo transformado la información que disponíamos en formato abstracto a binario:

$$io : fwrite(" \sim s \sim n", [erl\_prettypr : format(erl\_syntax : form\_list(AC))]). \quad (3.4)$$

## 3.3 Descripción formato Abstracto y Core

Actualmente, los depuradores utilizan dos representaciones. Estas representaciones se obtienen de transformar los archivos '.erl' a otro formato, como puede ser formato Core o Abstracto. Cuando se realiza la transformación se generan distintos tipos de nodo. Estas representaciones son los estándares que rigen cómo representar en un nivel más bajo las diferentes instrucciones del código fuente, que van a ser analizadas por el depurador. Cada una de ellas da una serie de ventajas a la hora de analizar un código fuente. Sin embargo, dependiendo de cuál sea el objetivo de su uso, será mejor una aproximación u otra. En cualquier caso, al final del proceso, dispondremos de un árbol, donde cada uno de sus nodos representará una instrucción dentro de nuestro código fuente. Sin más preámbulos, vamos a comenzar a explicar cada una de las aproximaciones.

La primera aproximación recibe el nombre de Abstract Format. AbstractFormat se caracteriza porque no almacena toda la información asociada al código fuente, es por eso que es más abstracto. Es decir, el árbol que se genera, a comparación de la otra representación, es más pequeño, o cada uno de sus nodos, dispone de menos información. Sin embargo, esto hace que puede llegar a ser muy potente, sobre todo para aplicaciones que no necesite toda la información del código fuente. Este formato organiza la información principalmente en ocho grupos distintos:

- Declaración de módulo: Son directivas de preprocesado para el compilador. Este grupo suele traducirse con una tupla compuesta por un primer átomo con el nombre *attribute* | *record* | *type record*. Un ejemplo de sentencia sería: *module(ejemplo)*.

- Átomos literales: Son tanto los literales como las variables. Este grupo suele traducir se con una tupla compuesta por un primer átomo con uno de los siguientes nombres: *integer* | *float* | *atom* | *string*. Un ejemplo de sentencia sería: *Ejemplo Literal*.

- Cláusulas: Suelen ser cuerpos de función y se caracterizan porque están representadas con una tupla cuyo su primer elemento es un átomo *{ clause, , }* Un ejemplo de sentencia sería: *[FC,...,FCn]*.

- Guardas: Suelen ser patrones que corresponden a las tuplas o algunos tipos de llamada, como por ejemplo una *callback*. Un ejemplo de sentencia sería: *{atomo,Valor}*.

- Tipos: Este grupo corresponde a todas las instrucciones de definición de funciones o variables. Un ejemplo de sentencia seria:*typeorddict(Key,Val) :: [Key, Val]*.

- Preprocesamiento: Este grupo está reservado para los usuarios que deseen trabajar más en investigación y deseen generar su propios grupos o tipos dentro del grupo. Este grupo suele ser muy utilizado en el área de la metaprogramación. Un ejemplo de sentencia sería: *{raw_abstract v1,AbstractCode}*.

- Expresiones: Este grupo es muy amplio ya que prácticamente abarca todo lo que no corresponda a ninguno de los grupos anteriores. Desde un *try* hasta un *spawn* o una operación de cualquier tipo. Un ejemplo de sentencia sería:

$$X = Valor + 4PiRadio. \tag{3.5}$$

```
1 -module(ejemploAbstract).
2 -export([ejemploFunc/2]).
3
4 ejemploFunc(A, B) ->
5   A + A.
```

**Listing 3.6:** Archivo *ejemploAbstract.erl*.

Vamos a partir de un pequeño caso de estudio, para ver cómo se organiza la información en cuanto al formato abstracto. Sin embargo, aunque el ejemplo es pequeño, el árbol generado es bastante grande. Esto hará que nos centramos en un subárbol del árbol generado, partiendo del ejemplo concreto de la función, como se muestra en *ejemploAbstract.erl*. Este método consiste en la creación de una variable X e Y, asignándoles un valor, y después realizar la operación de suma, para almacenar el resultado en la variable C.

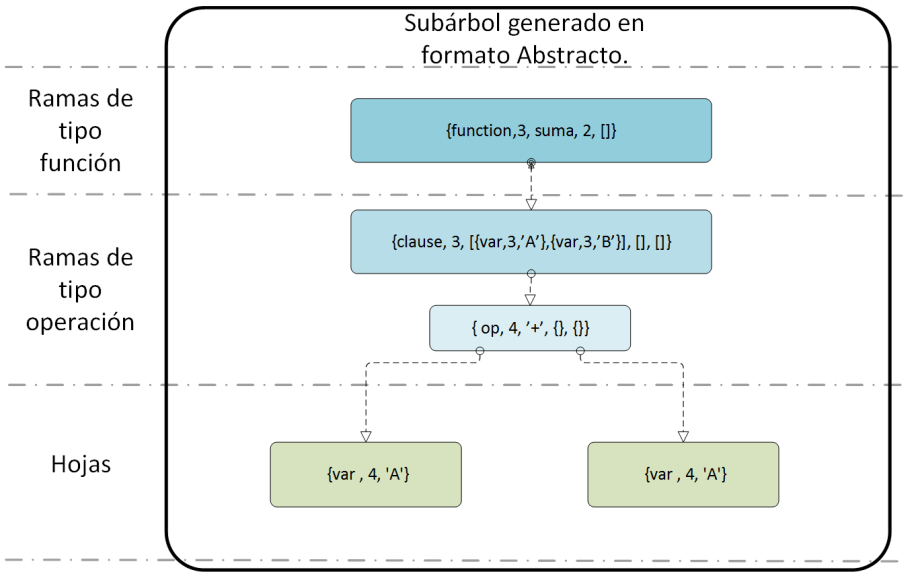


**Figura 3.2:** Ejemplo de árbol generado en formato Abstracto.

En la figura 3.2 se muestra un subárbol generado sobre el código fuente. Hemos partido del subárbol de la información sobre la función dejando de lado la información de directivas y otros tipos de información. En este caso concreto, podemos identificar tres zonas de acción determinadas. En la primera zona están las ramas función, que en este caso de estudio, al tener sólo una función el código fuente, producirá que solo un nodo. Cada nodo de tipo función se compone de seis elementos. El primer elemento es un átomo que referencia el tipo de instrucción. El segundo elemento es la etiqueta asignada dentro del árbol, seguido de un tercer elemento que es un literal que representa el nombre de la función. El cuarto elemento es un entero, que representa el número de parámetros que recibe. El quinto elemento es una lista con un par de tuplas, la cláusula asociada a la función y el nodo donde finaliza la cláusula. La cláusula dispone de todo el contenido que va ejecutar dicha función.

A continuación, están las ramas de operaciones, que son las ramas de donde cuelgan principalmente hojas. En este caso, tenemos dos tipos. El primero es *Match* un tipo que representa la igualdad, y *op*, la operación. Ambos tipos, disponen de dos tuplas, que pueden contener más operaciones o variables. La única diferencia en este caso es que añaden una operación. Por último, tenemos las hojas, que son tuplas de tres elementos, donde el último valor representa el tipo de nodos generados con el que se va trabajar. En el caso de la variable, este elemento es la incógnita, y en el caso del número entero, el número entero.

El segundo concepto a tratar es el formato Core. Core es más exhaustivo respecto a la información almacenada durante la transformación. Es un formato más cercano al formato con el que trabaja el compilador y da toda la información asociada al código fuente. Es bastante más difícil de utilizar que el abstracto, sobre todo a la hora de analizar la semántica respecto a la concurrencia. La organización que rige el formato Core no es como la del Abstract Format, ya que no tiene grupos asociados, sino que se rige más por tipos asociados, pero sí podemos agruparlos por su uso.

- Tipo nodo hoja: Son las variables que acaban como hojas dentro del árbol. Sus tipos solo pueden ser *c_var* y *c_literal*.

- Tipo nodo operación: Son unos nodos que agrupan hojas u otros nodos para hacer operaciones determinadas entre ellos. Sus tipos son: *c_apply*, *c_cons*, *c_let*, *c_letrec*, *c_map*, *c_map_pair*, *c_primop*, *c_catch*, *c_call*, *c_receive*, *c_case*, *c_seq*, *c_try*, *c_alias*, *c_bitstr*, *c_binary*, *c_values* y *c_tuple*.

- Tipo nodo función: Son una serie de nodos que se encargan de agrupar otros nodos o hojas, que son las diferentes instrucciones que van a ejecutar en la función. Tipos: *c_clause* y *c_fun*.

- Tipo nodo raíz: Es el nodo raíz del árbol. Tiene el siguiente átomo asociado *c_module*.

```
1 -module(ejemploCore).
2 -export([ejemploFunc/0]).
3
4 ejemploFunc(X , Y) ->
5   Y + X.
```

**Listing 3.7:** Archivo *ejemploCore.erl*.

Un ejemplo que sirva de ejemplo del formato Core es el *ejemploCore.erl*. Partiendo de este caso de estudio visualizamos el árbol generado tras transformar de ".erl" a formato ".core". Aunque, como el código generado por el compilador es bastante grande, para simplificar la explicación partiremos del nodo función. El método que dispone necesita de dos parámetros. A estos parámetros se le realizan una serie de operaciones, para ver las distintas ramas que se van generando en el árbol resultante de la transformación a formato Core.

Enl subárbol generado se muestra en la figura 3.3. Las zonas de división y organización del árbol son las mismas que en el formato Abstracto, aunque, se pueden identificar grandes diferencias entre ambos. La primera diferencia es cómo se identifica los distintos nodos. Utiliza el identificador *c_name* que determina que es un nodo de tipo Core, lo que da más expresividad al código. Otro dato interesante son las *c_var* que ya no mantienen el nombre original utilizado en el código

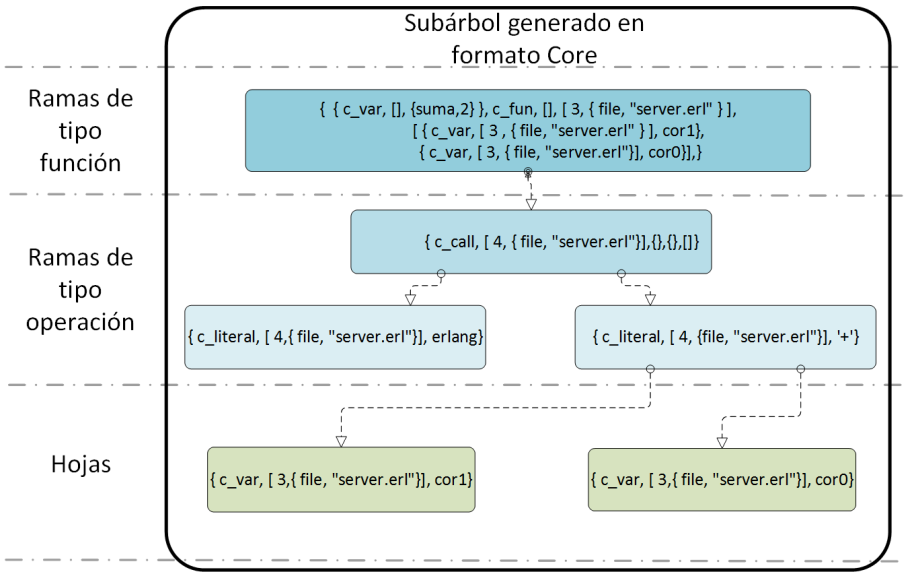


**Figura 3.3:** Ejemplo de árbol generado en formato Core.

fuente. Estas variables se etiquetan con la palabra clave 'cor' unido a un identificador entero. Además, los nodos incluyen no solo la línea donde se encuentra la instrucción (como en el formato Abstracto), sino que además se especifica el nombre módulo en donde se llama la función, junto a su tipo. Por último, disponemos de un nuevo nodo de tipo *c_literal*. Este nodo suele estar acompañado de una operación. El principal objetivo de este nodo es avisar al intérprete de que tiene que ejecutar una operación de tipo local (que es este caso) o remoto. En cuanto al resto la estructura, es prácticamente similar a la vista en la figura 3.2.

## 3.4 Sintaxis formato Core

Es necesario conocer la sintaxis de un lenguaje de programación para que una herramienta de depuración sea capaz de poder simular y ejecutar cada una de las instrucciones [**?**]. También es necesario implementar un evaluador, que será el encargado de evaluar cada una de las expresiones del código fuente. En la figura 3.4 se muestra un resumen de la sintaxis del lenguaje Erlang.

Una vez conozcamos la sintaxis del lenguaje Erlang, podremos comenzar a desarrollar el evaluador. El evaluador es fundamental para poder instrumentar [**?**] el codigo fuente. Sin embargo, el evaluador no podrá simular ninguna de las instrucciones, ya que es necesario ver la definición formal de cada una de las instrucciones (para poder dotar al evaluador de esa semantica y que sea capaz de simular dicho comportamiento. En el artículo [**?**] podemos ver como se describe el comportamiento de cada una de estas instrucciones para poder crear el depurador.

Vamos a ver un pequeño ejemplo de implementación de la parte del evaluador, a partir de la definición formal. En este ejemplo, se plantea interpretar una operación del tipo instrucción X + Y. Esta instrucción consta de dos variables y

```
module  ::= module Atom [fname_i1, ..., fname_ik]
            attributes [Atom_1 = const_1, ..., Atom_m = const_m]
            fname_1 = fun_1 ··· fname_n = fun_n end
fname   ::= Atom / Integer
const   ::= lit | [const_1 | const_2] | {const_1, ..., const_n}
lit     ::= Integer | Float | Atom
          | Char | String | []
fun     ::= fun (var_1, ..., var_n) -> exprs
var     ::= VariableName
exprs   ::= expr | <expr_1, ..., expr_n>
expr    ::= var | fname | lit | fun
          | [exprs_1 | exprs_2] | {exprs_1, ..., exprs_n}
          | let vars = exprs_1 in exprs_2
          | do exprs_1 exprs_2
          | letrec fname_1 = fun_1 ··· fname_n = fun_n in exprs
          | apply exprs_0(exprs_1, ..., exprs_n)
          | call exprs_n+1:exprs_n+2(exprs_1, ..., exprs_n)
          | primop Atom(exprs_1, ..., exprs_n)
          | try exprs_1 catch (var_1, var_2) -> exprs_2
          | case exprs of clause_1 ··· clause_n end
          | receive clause_1 ··· clause_n after exprs_1 -> exprs_2
vars    ::= var | <var_1, ..., var_n>
clause  ::= pats when exprs_1 -> exprs_2
pats    ::= pat | <pat_1, ..., pat_n>
pat     ::= var | lit | [pat_1 | pat_2] | {pat_1, ..., pat_n}
          | var = pat
```

**Figura 3.4:** Resumen de sintaxis sobre el formato Core.

una operación. La directiva *call* hace referencia a una ejecución, y en el caso de estar acompañada de la directiva Erlang, hace referencia a una ejecución de una operación (como puede ser la suma, la resta, multiplicación u otro tipo de operaciones similares).El ejemplo se encuentra en la parte de anexos, concretamente *exampleEval.erl*. En primer lugar, es necesario transformar el archivo al formato Core. Además, es necesario escanear el archivo, para controlar la transformación. A continuación, mapeamos la estructura en busca de la expresión de tipo operación que queramos evaluar. Sólo queda recuperar el valor de las variables almacenadas en el entorno para poder evaluarlas. Una vez tenemos los valores de las variables, se realiza un *apply* con el tipo de operación disponible para obtener el resultado.

```
1  -module(exampleEval).
2
3  -export([test/1]).
4
5  -include_lib("compiler/src/core_parse.hrl").
6
7  -type info() :: anno | attributes | exports | name.
8
9  -spec test(module()) -> ok.
10 test(_File) ->
11   _Core = compile_core(_File,"."),
12   {ok, Tokens, _} = scan_file(_Core),
13   {ok, AST} = core_parse:parse(Tokens),
14   io:format("Estado del AST: ~p~n",[AST]),
```

```
15    Db = ets:new(x, [public]),
16    store_module_info(anno, _File, AST, Db),
17    store_module_info(name, _File, AST, Db),
18    store_module_info(exports, _File, AST, Db),
19    store_module_info(attributes, _File, AST, Db),
20    store_module_funs(_File, AST, Db),
21    io:format("--------------------Final----------------"),
22    ok.
23
24  -spec scan_file(file:filename()) -> term().
25  scan_file(File) ->
26    {ok, FileContents} = file:read_file(File),
27    Data = binary_to_list(FileContents),
28    core_scan:string(Data).
29
30  -spec compile_core(module(), nonempty_string()) -> file:filename().
31  compile_core(M, Dir) ->
32    ok = filelib:ensure_dir(Dir ++ "/"),
33    {ok, BeamPath} = ensure_mod_loaded(M),
34    {ok, {_, [{compile_info, Info}]}} = beam_lib:chunks(BeamPath, [
          compile_info]),
35    Source = proplists:get_value(source, Info),
36    Includes = proplists:lookup_all(i, proplists:get_value(options, Info)
          ),
37    Macros = proplists:lookup_all(d, proplists:get_value(options, Info)),
38    CompInfo = [to_core, return_errors, {outdir, Dir}] ++ Includes ++
          Macros,
39    CompRet = compile:file(Source, CompInfo),
40    _Pruebas = Dir ++ "/" ++ atom_to_list(M) ++ ".core".
41
42   -spec ensure_mod_loaded(module()) -> {ok, file:filename()} | no_return
        ().
43  ensure_mod_loaded(M) ->
44    case code:which(M) of
45      non_existing    -> erlang:throw(non_existing);
46      preloaded       -> erlang:throw(preloaded);
47      cover_compiled -> erlang:throw(cover_compiled);
48      Path -> {ok, Path}
49    end.
50
51  -spec store_module(module(), ets:tid(), nonempty_string()) -> ok.
52  store_module(M, Db, Dir) ->
53    Core = compile_core(M, Dir),
54    {ok, Tokens, _} = scan_file(Core),
55    {ok, AST} = core_parse:parse(Tokens),
56    store_module_info(anno, M, AST, Db),
57    store_module_info(name, M, AST, Db),
58    store_module_info(exports, M, AST, Db),
59    store_module_info(attributes, M, AST, Db),
60    store_module_funs(M, AST, Db),
61    ok.
62
63  -spec store_module_info(info(), module(), cerl:cerl(), ets:tab()) -> ok
        .
64  store_module_info(anno, _M, AST, Db) ->
65    Anno = AST#c_module.anno,
66    true = ets:insert(Db, {anno, Anno}),
67    ok;
68
```

```
69 store_module_info(attributes, _M, AST, Db) ->
70   Attrs_c = AST#c_module.attrs,
71   true = ets:insert(Db, {attributes, Attrs_c}),
72   ok;
73
74 store_module_info(exports, M, AST, Db) ->
75   Exps_c = AST#c_module.exports,
76   Fun_info =
77     fun(Elem) ->
78       {Fun, Arity} = Elem#c_var.name,
79       {M, Fun, Arity}
80     end,
81   Exps = [Fun_info(E) || E <- Exps_c],
82   true = ets:insert(Db, {exported, Exps}),
83   ok;
84
85 store_module_info(name, _M, AST, Db) ->
86   ModName_c = AST#c_module.name,
87   ModName = ModName_c#c_literal.val,
88   true = ets:insert(Db, {name, ModName}),
89   ok.
90
91 -spec store_module_funs(module(), cerl:cerl(), ets:tab()) -> ok.
92 store_module_funs(M, AST, Db) ->
93   Funs = AST#c_module.defs,
94   [{exported, Exps}] = ets:lookup(Db, exported),
95   operation(Funs).
96
97 operation([S|T])->
98   operantion(S);
99
100 operation({{c_var,_,{_NameFun,Arity}},_Fun})->
101   String = test1,
102   case String of
103   test1 ->
104       io:format("Estado correcto\n"),
105       {c_fun,_,_,Tuple} = _Fun,
106       eje_Call(Tuple);
107   true -> io:format("Error")
108 end.
109
110 binding(Name, Bs) ->
111     case orddict:find(Name, Bs) of
112   {ok,Val} -> {value,Val};
113   error -> unbound
114     end.
115
116 create_Param() ->
117 [{'_cor0',22},{'_cor1',11}].
118
119
120 est_values({c_literal,[],V}) ->
121   V;
122
123 est_values({c_var,[],Name}) ->
124   {value,Val} = binding(Name,create_Param()),
125   Val.
126
127 eje_Call({c_call,_,_Ej,{c_literal,[],_OP},[Arg1,Arg2]}) ->
```

```
128   V1 = est_values(Arg1),
129   V2 = est_values(Arg2),
130   io:format("Los Valores a evaluar son: ~p y ~p\n",[V1,V2]),
131   io:format("La operacion que se va realizar es: ~p\n",[_OP]),
132   case _Ej of
133   {c_literal,_,erlang} ->
134     result(erlang,_OP,[V1,V2]);
135   error -> unbound
136     end,
137   io:format("La llamada de la funcion: ~p \n",[_Call]).
138
139 result(erlang,_OP,_Lists) ->
140   io:format("El valor final es: ~p\n",[erlang:apply(erlang,_OP,_Lists)]
        ).
```

**Listing 3.8:** Archivo *exampleEval.erl*.

# CAPÍTULO 4
# Diseño e implementación

## 4.1 Descripción del sistema

Los lenguajes de programación utilizados, han sido Python y Erlang. Concreta mente la versión de Python es la 3.4.0 (la penúltima disponible en este momento). En cuanto al lenguaje Erlang, se instaló la revisión 16. Además esta versión de Erlang trae una documentación y ejemplos asociados. El sistema operativo en donde se usan estos lenguajes de programación para este proyecto es Windows 10.

Para el desarrollo de la aplicación se ha necesitado usar una serie de herramientas y librerías. La primera de ellas es wxWidgets (versión 3.0.0), que está disponible en Internet en su página oficial, para todos los sistemas operativos. Esta librería es la utilizada para el desarrollo de la interfaz de la herramienta de depuración. La libreria Erlport es compatible con la versión 3 de Python, para la generación de una interfaces que comunique Erlang con Python. Las librerías OTP están en la versión 19 de Erlang, para poder hacer uso del concepto mostrado anteriormente: *Behavior*. En cuanto a nivel *hardware*, se ha utilizado una estación de trabajo convencional compatible con los sistemas Windows, GNU/Linux y Mac, por lo que la aplicación es compatible con todos ellos. Para el desarrollo de la aplicación, se ha utilizado Git version (2.9.0), sobre la plataforma Gitlab (revisión 8.11). El IDE utilizado es el Eclipse Juno sr2, con los paquetes incluidos para *testers*.

En términos generales, en la figura 4.1 se muestra un esquema del funcionamiento tanto interno como externo de la aplicación. La aplicación se divide en tres fases, y cada una de estas fases agrupa una serie de funcionalidades. La primera fase es la fase de inicialización: su principal función es la de tratamiento de datos (la carga del código fuente en el entorno, transformar el código fuente a un lenguaje intermedio instrumentarlo), para facilitar su uso por la herramienta de depuración. Además, inicializa todas las opciones de la interfaz en base al archivo de configuración. La segunda, es la fase de evaluación. Una vez ejecutado un código fuente en la aplicación de depuración mediante la interfaz gráfica, la aplicación empieza a interpretar cada una de las instrucciones y a simular su comportamiento. Por último, la fase de resultados va mostrando al usuario la

información disponible en la herramienta. El usuario puede interactuar con la herramienta para recibir más información o finalizar su ejecución.

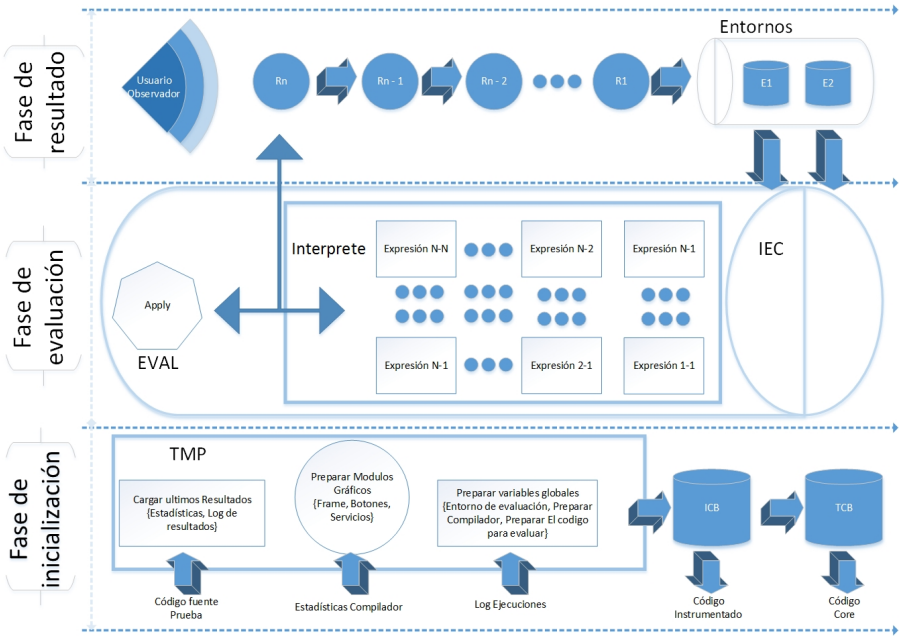


**Figura 4.1:** Resumen herramienta depuración para Erlang.

## 4.2 Fase de inicialización

En esta sección vamos a ver en profundidad cómo funciona la fase de inicialización de la aplicación. En primer lugar, necesitamos ejecutar el intérprete para poder ejecutar la aplicación. Una vez ejecutado el simulador (utilizando el comando *cd(Path)*), podemos movernos dentro del sistema, como podemos ver en la figura 4.2. Mediante el uso de 'c' se compilan los fuentes con el flag *debug_info*. Es necesario compilar los archivos con este flag para poder realizar el uso de primitivas del compilador, y usar las transformaciones. Si no utilizamos ese flag, el compilador no estaría disponible toda la información del fuente original en el archivo binario, lo que produciría errores a la hora de transformar de un formato a otro ese código fuente. Una vez ejecutado, nos movemos hasta el directorio raíz de la aplicación y ejecutamos el comando *graficos:start()*.

Una vez se ejecuta el comando, se pondrá en funcionamiento el proceso de gráficos. Este proceso inicializa todos los componentes de la interfaz y preparará el programa. El resultado es el que se muestra en la figura 4.3. En esta figura podemos encontrar una serie de etiquetas. En la primera etiqueta E1, hay un botón, que si lo pulsamos, se crea una ventana emergente para buscar el archivo que queremos procesar con el depurador. Hay un ejemplo de esta ventana emergente en la figura 4.4, generada por el evento al pulsar el botón *browser*. En la segunda etiqueta E2 tenemos un ejemplo de caso de prueba para el depurador. Hay que sustituir el contenido del campo de texto por los parámetros que deseemos com-

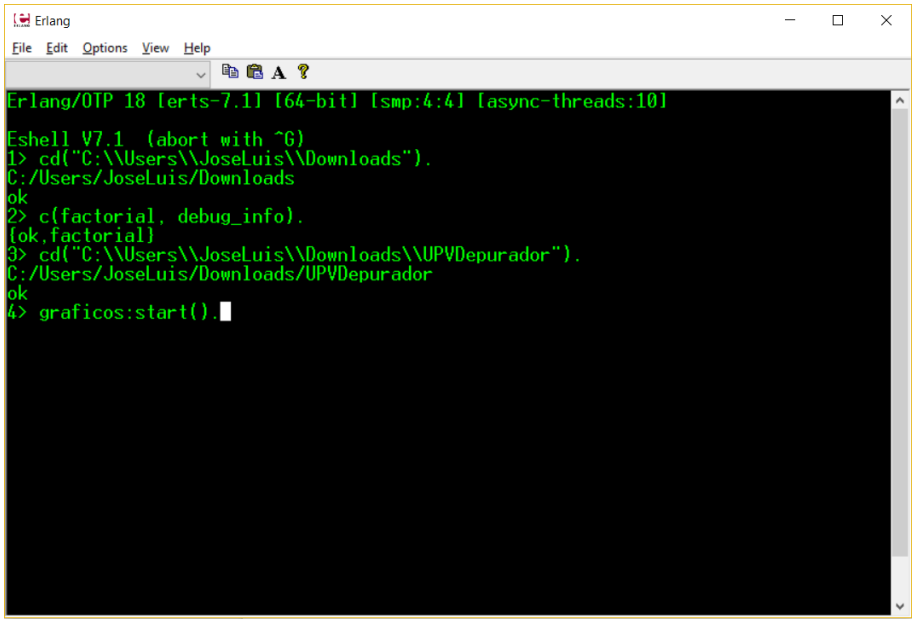


**Figura 4.2:** Intérprete de Erlang.

probar de nuestro fuente. Los parámetros de ejecución que hemos utilizado son los siguientes:

$$factorial, fact, [5] \tag{4.1}$$

El formato de entrada es una tupla en donde el primer elemento especifica el módulo cargado. El segundo elemento es el nombre de la función que se va evaluar. El último elemento de la tupla es la lista de valores para los argumentos que necesita la función. La tercera etiqueta E3, es un objeto encargado de mostrar al usuario el fichero cargado. Por último, tenemos la etiqueta E3, que es el botón encargado de ejecutar el depurador.

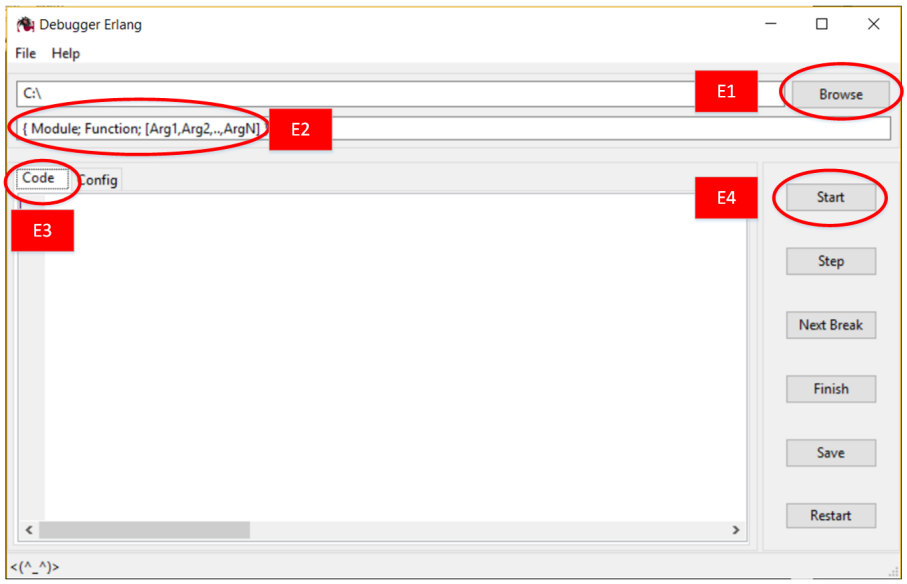


**Figura 4.3:** Interfaz gráfica herramienta depuración.

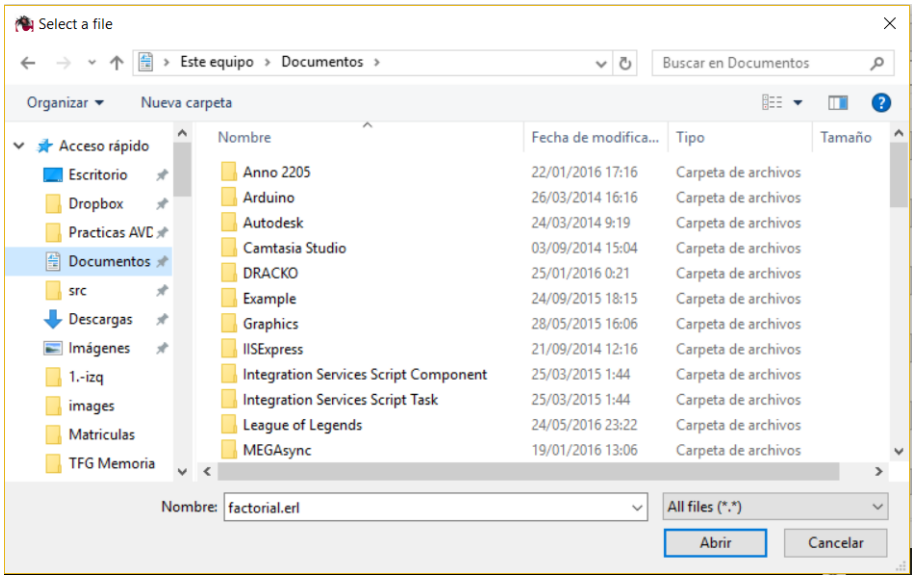


**Figura 4.4:** Ventana emergente para la búsqueda de código fuente.

Hemos visto el funcionamiento desde el punto de vista del usuario. Ahora vamos a ver cómo influyen internamente las distintas acciones. Como se muestra en la figura 4.5, el primer proceso en ejecutarse es *grafics* usando su método principal *start()*. Este módulo consulta, mediante el uso de distintas llamadas, la configuración de las opciones gráficas, haciendo uso del módulo *config*. Además, se ejecuta tanto el proceso del servidor (*server*) y la interfaz(*int*). Cuando se carga un archivo en la herramienta se dispara un evento en *grafics* de tipo carga de fichero. Cuando recibe este evento, comunica a través de la interfaz al servidor, si ese módulo ya está en memoria. *Server*, que hace de enrutador, consulta a DB si está almacenado ya en memoria este archivo. En caso que no lo tenga el módulo en memoria, lo cargará en DB. En caso contrario, no lo cargará. Finalmente, cuando esté cargado en DB y dé el vistobueno *server*, entonces *grafics* mostrará el contenido del archivo en la interfaz gráfica.

## 4.3 Fase de evaluación

Hemos hablado de cómo cargar un fichero y qué implicaciones tiene internamente. Vamos a ver ahora una vez cargado el programa e inicializado, de qué opciones disponemos antes de ejecutar el evaluador. En la pestaña principal, seleccionamos una línea y pulsamos el botón derecho. En la figura 4.6 podemos observar las opciones disponibles. Cuando seleccionemos un tipo de ellos aparecerá un punto rojo en el margen izquierdo de esa línea. En ese caso, estaremos haciendo uso de los puntos de interrupción. Los puntos de interrupción sirven para parar un proceso y así poder inspeccionar la información disponible en el depurador. En esta aplicación hay tres tipos disponibles:

1. *Break*: El programa le asigna un círculo de color rojo en el margen izquierdo. Es el más utilizado por los desarrolladores. Cuando el intérprete está

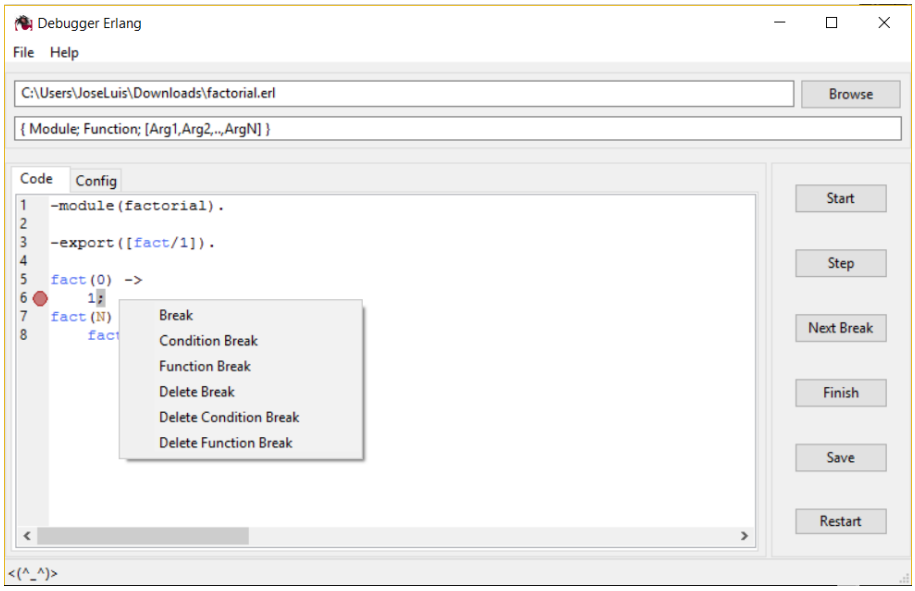


**Figura 4.5:** Creación de puntos de interrupción.

evaluando una instrucción con un punto de ruptura, se para el proceso y espera alguna interacción por parte del usuario.

2. *Condition break*: Cuando seleccionemos esta opción el círculo será de color Azul. No es tan utilizado. La diferencia principal con el punto de ruptura es que le añade una condición y sólo parará el depurador la ejecución si se cumple dicha condición.

3. *Function break*: Al pulsar sobre esta opción aparecerá un círculo de color amarillo. A diferencia del resto, sólo se detendrá en la ejecución de la función donde esté ese punto. Esto se debe a que se asocia el punto de ruptura a la cláusula de la función.

Si observamos la figura 4.7, podemos ver los distintos módulos de la aplicación. Cuando añadimos un punto de ruptura de cualquier tipo se genera un evento dentro del proceso *grafics*. *Grafics* gestiona el evento en su cola y manda un mensaje serializado, con el uso de la interfaz a server. Este mensaje tiene la siguiente forma:

$$[Identificador, Modulo, linea]. \tag{4.2}$$

El primer elemento identifica el tipo de punto de ruptura. El segundo elemento hace referencia al módulo y el último elemento a la línea correspondiente de ese módulo. El proceso servidor recibe el mensaje y comprueba si existe ya o no ese

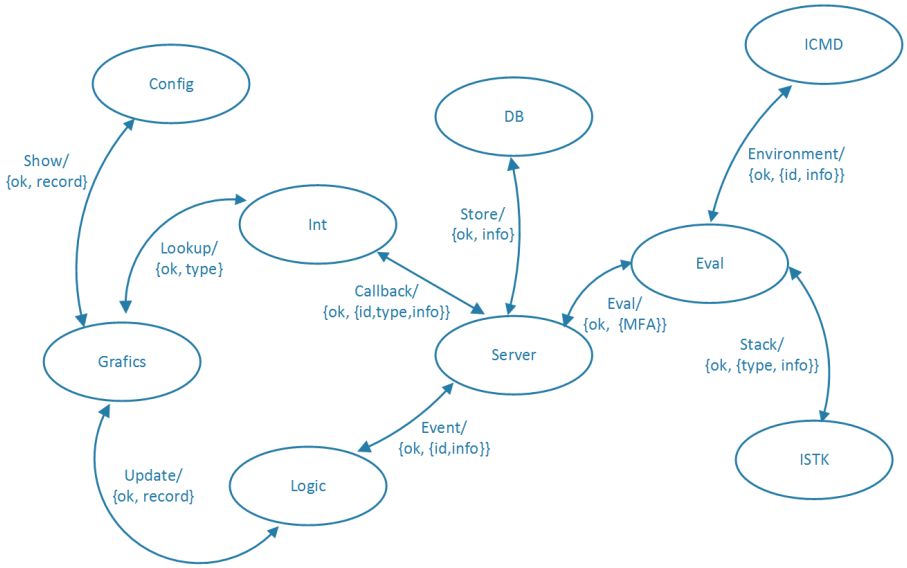


**Figura 4.6:** Gestión de recursos de la herramienta de depuración.

punto de ruptura en la memoria. En caso de que no exista, lo añadirá a la memoria y notificará con un 'ok' al proceso *grafics*. En caso de que ya esté registrado devolverá un error a *grafics* con el mensaje "Este punto ya está registrado". Si lo que queremos es eliminar un punto de ruptura, el comportamiento es el mismo.

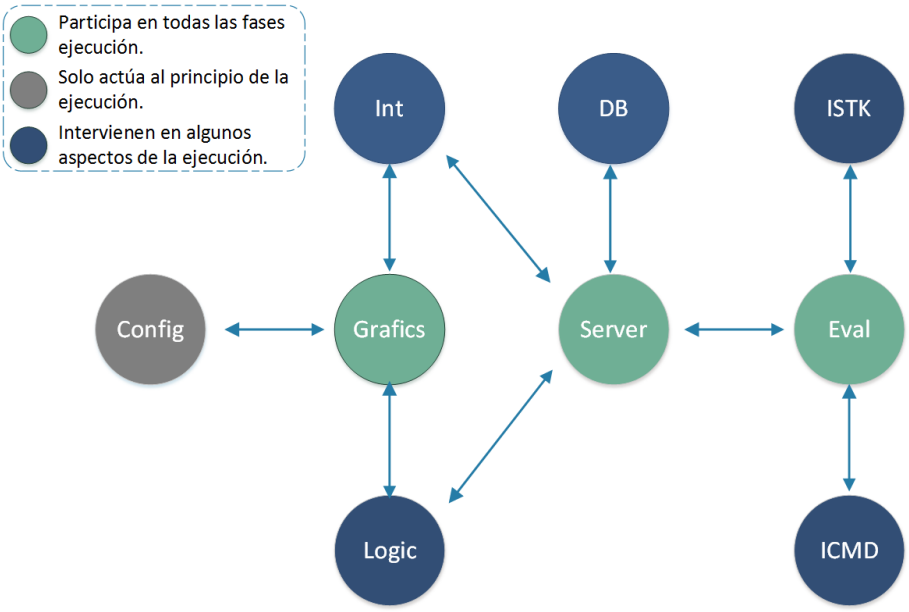


**Figura 4.7:** Estructura interna de la herramienta de depuración.

## 4.4 Fase de resultados

En este apartado vamos a ver qué opciones tiene disponible el usuario para interactuar con la interfaz. Para realizar esa interacción con la aplicación están disponibles unos botones en la parte derecha. En la figura 4.8 podemos ver los

botones disponibles. Entre las distintas opciones tenemos las siguientes:

1. *Start*: Esta opción sirve para ejecutar el depurador. Generará un proceso, que manejará el evaluador y trabajará con el entorno de las variables, hasta que alcance un punto de interrupción que pare este proceso. La línea verde aparece en el primer punto de ruptura que encuentre el depurador.

2. *Step*: Salta a la siguiente instrucción. Dispara un evento en *grafics*, que hará que mande un mensaje al servidor, comunicando que ha de parar la ejecución en la siguiente línea de la que se encuentra actualmente el evaluador. La línea verde aparecerá en la siguiente de la que se encuentra el punto de ruptura.

3. *Next Break*: Salta al siguiente punto de ruptura. Esta vez, seguirá evaluando todas las instrucciones hasta que se encuentre otro punto de ruptura.

4. *Finish*: Finalizar la ejecución del depurador. Reinicializa todos los componentes, excepto el BD (en donde se almacena el código fuente). De esta forma si el usuario quiere realizar otra ejecución con el mismo fuente, lo tiene ya cargado en memoria. Elimina tanto la línea verde de estado como los puntos de ruptura.

5. *Save*: Guarda la configuración del depurador.

6. *Restart*: Inicializa la aplicación.Es similar a Finish, excepto porque esta opción sí limpia la memoria que tiene almacenada el código fuente de la aplicación. Por lo tanto, toda la información mostrada en los distintos campos desaparecerá.

Cuando pulsamos el botón *start*, se crea una ventana emergente con las variables disponibles, dentro del método que estamos evaluando. El valor de estas variables corresponde al instante de tiempo anterior a la línea del punto de interrupción. La primera columna hace referencia al nombre con el que se conoce a la variable en el método. La segunda columna corresponde a su valor. Según interactúe con los botones de la interfaz actualizaremos la ventana emergente. Disponemos en la figura 4.9 de un ejemplo de ventana emergente: ésta gestiona el contenido de variables de la función. Cuando la ejecución finalice, la ventana se destruirá para evitar posibles errores con nuevas ejecuciones. Esta ventana dispone de un split para evitar un desbordamiento si el método a comprobar contiene muchas variables.

Cuando la aplicación finaliza la depuración, aparece una ventana emergente como la de la figura 4.10. El objetivo de esta ventana es dar algo de retroalimentación al usuario sobre la ejecución. Esta ventana emergente sólo aparecerá en el

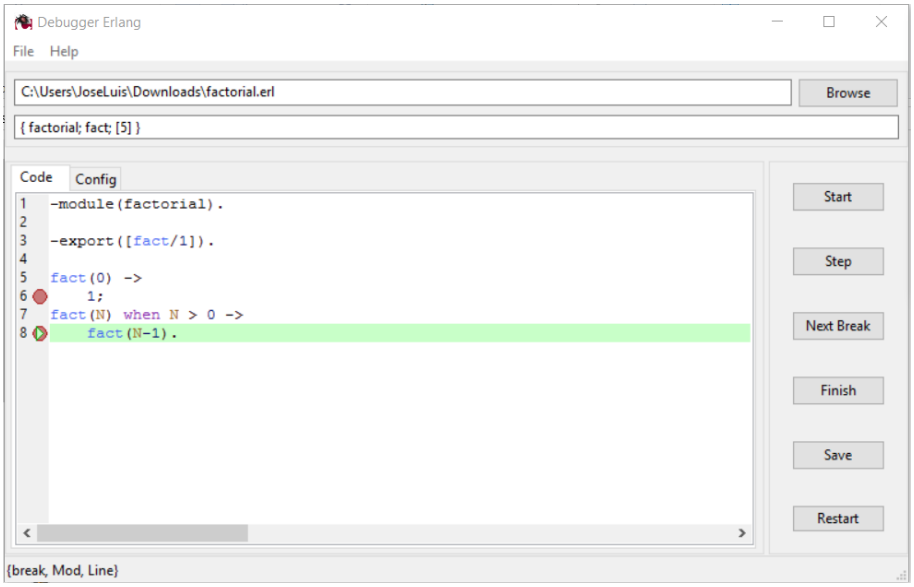


**Figura 4.8:** Ejecución de la herramienta depuración.

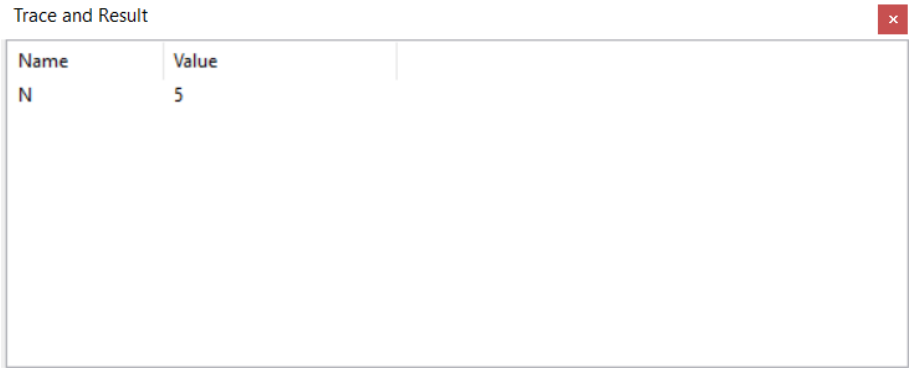


**Figura 4.9:** Estado de las variables en el entorno.

caso en que se finalice toda la depuración, ya que si utilizamos los botones *finish* o *restart* no aparecerá esta ventana emergente. Entre la información disponible:

- Resultado: El resultado de ejecutar la función con los argumentos utilizados en la depuración.
- Módulo: Nombre del módulo que contiene la función.
- Función: Función que se va simular la ejecución.
- Argumentos: Parámetros utilizados, para dar valor de manera estática a las variables.
- Comentario: Información de cómo fue la simulación de la función.

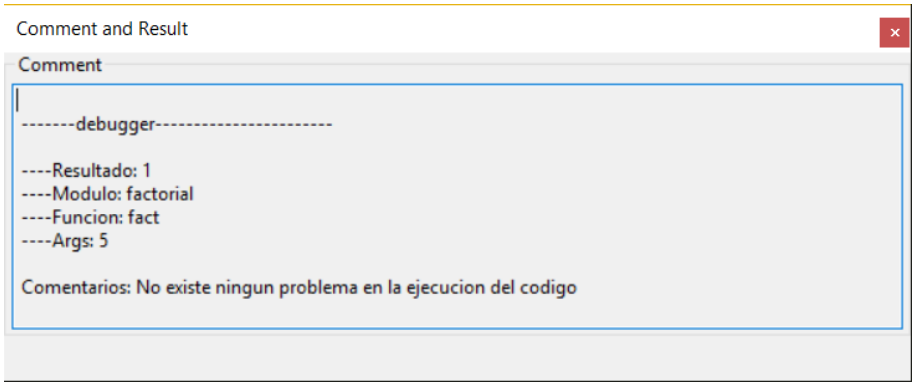


**Figura 4.10:** Ventana emergente de comentarios.

La última opción, disponible para el usuario es la pestaña Config, en la figura 4.11. En esta pestaña el usuario puede modificar los parámetros, para influir en el comportamiento de la aplicación:

- *ColorPragma*: Hace referencia a el color de los puntos de ruptura, en formato RGB (Red Green Blue) seguido de un identificador, que es el orden del menú del botón derecho.

- *Depth*: Es la profundidad máxima que se puede alcanzar haciendo uso del depurador.

- *ColorComment*: Es el color de los comentarios en la ventana emergente de la figura 4.10.

- *NumberChars*: Es el número de caracteres límite por línea que aparecerán en la pestaña Code cuando se carga un código fuente.

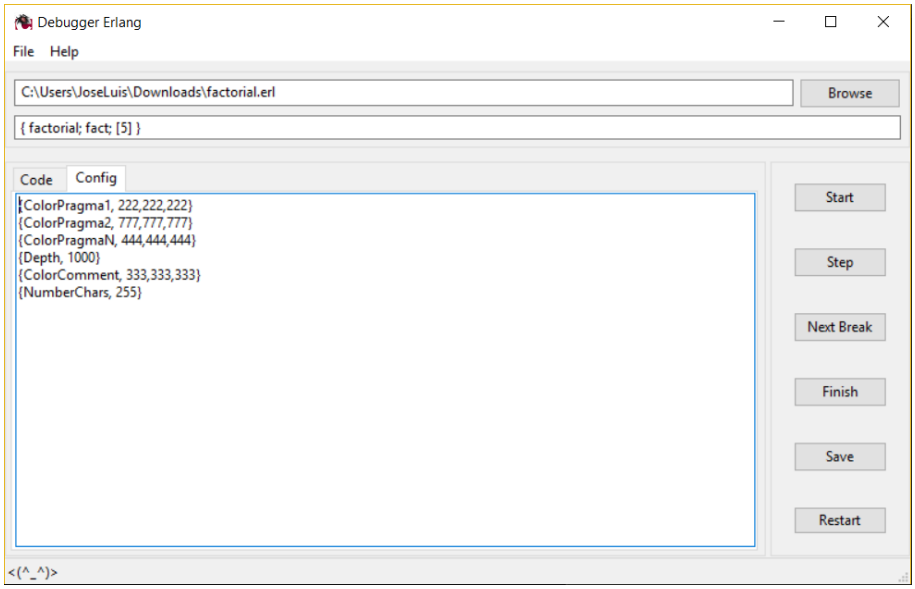


**Figura 4.11:** Pestaña de configuración.

# CAPÍTULO 5
# Conclusiones

Erlang es un lenguaje funcional orientado a la concurrencia mediante el uso de procesos. Estos procesos pueden crearse tanto de manera local como remota. Las principales características de Erlang son la escalabilidad, tolerancia a fallos, *soft-real-time* y el funcionamiento ininterrumpido. En cuanto a la sintaxis del lenguaje, hay que destacar que tiene evaluación estricta, asignación única de variable y tipado dinámico. Pese a ser un lenguaje de programación algo complejo, dispone de una enorme cantidad de documentación y una inmensa comunidad.

Hoy en día, todos los lenguajes de programación disponen de un depurador. La depuración se plantea como una solución ante los errores existentes en las aplicaciones. Gracias a los depuradores, se identifica el estado de cada una de las variables para un instante de tiempo durante el ciclo de ejecución de la aplicación.

Este trabajo presenta una herramienta de depuración con interfaz gráfica para el lenguaje de programación Erlang. La herramienta cubre toda la semántica del lenguaje, excepto la concurrencia, que actualmente se encuentra en el estado del arte. Tras realizar una ejecución concreta con la herramienta de depuración sobre un código fuente, obtenemos una serie de resultados. Estos resultados permiten detectar errores dentro del código fuente, ver un análisis de tiempos de ejecución entre otras cosas.

El desarrollo de la herramienta ha durado un año y cuenta con una inmensa cantidad de líneas de código. También ha sido necesario estudiar en profundidad la literatura de diversos ámbitos, así como disponer de los conocimientos necesarios para realizar el proyecto, además de la complejidad que requiere aprender un lenguaje de programación nuevo con un nivel lo suficientemente alto.

# CAPÍTULO 6
# Trabajo futuro

Este trabajo de fin de grado nace con el objetivo de profundizar en el área de los depuradores. En este proyecto, hemos diseñado e implementado una herramienta con interfaz gráfica que combina algunas de las características de los diferentes modelos de depuración. El resultado es una herramienta para los desarrolladores de Erlang de distintos niveles. Además, sirve de estudio para los estudiantes o investigadores interesados en el aprendizaje de los depuradores, ya que está disponible para su descarga desde Internet.

Aunque el objetivo era abarcar toda la semántica del lenguaje, hemos tenido que dejar de lado la parte de concurrencia. Hay que tener en cuenta que la parte de concurrencia, en cuanto a su análisis de la semántica, se encuentra en el estado del arte. Por ello, tomamos la decisión de no intentar implementar esa parte en el depurador.

Explorando otros campos, una de las opciones que estuvimos contemplando fue el campo de la visión por computador. En este campo se podría utilizar el concepto de reconocimiento de caracteres, con el objetivo de depurar código escrito en papel. Esto sería útil para centros de estudios que no disponen de un presupuesto para que todos los alumnos dispongan de un ordenador. El alumno escribiría su algoritmo en papel y mediante el uso de un scanner o una fotografía con el móvil se procedería a su depuración, reconociendo previamente los términos escritos por el usuario en papel. Otra opción, dentro del mismo campo, sería crear un dispositivo similar a las famosas Google Glass. Estas gafas contarían con un arduino nano, un sensor de infrarrojo y una IMU. Gracias a estas gafas de bajo coste, se podría posicionar el campo visual del usuario en tiempo real. El objetivo es depurar código que se está visualizando en ese mismo instante. Los resultados de la depuración podrían visualizarse en la pantalla del teléfono móvil. Este dispositivo sería muy útil, sobre todo para analistas o profesores que están visualizando código continuamente.

Existen otros campos con los que se podría combinar y enriquecer aparte de la visión por computador. Por ejemplo, el campo de los sistemas distribuidos. Mediante el uso de los sistemas distribuidos, el objetivo sería la depuración de código colectiva. El concepto surge en que dos personas puedan depurar la mis-

ma función del mismo código. Esta solución favorece a la resolución de problemas gracias al trabajo en equipo. El compañero vería las ejecuciones realizadas por su compañero, resultado, casos de prueba e incluso número de ejecuciones. Además, tendría la opción de interactuar con las ejecuciones de su compañero en tiempo real. Otra opción, dentro del mismo campo, sería cambiar la arquitectura por otra distribuida utilizando algún API. Cambiar la arquitectura de la herramienta favorece el rendimiento de la aplicación mediante el uso de patrones de diseño. Por ejemplo, el Api ZeroMQ está disponible en Erlang y dispone de muchos patrones prediseñados (por ejemplo, *Pull and Push*, *Published and Subscriber*) para lograr este objetivo.

Sin embargo, si queremos mantenernos dentro del campo de la validación, se puede plantear el uso de asertos para evaluar bloques de código, sin la necesidad de utilizar una enorme cantidad de líneas de código. Los asertos contiene una expresión que tiene que ser evaluada. Gracias a los asertos aumentaría la potencia del depurador, ya que es más simple utilizar un término que varias sentencias *if*. Otra opción sería un generador de casos de prueba que sirvan de ayuda al usuario. El objetivo del generador sería eliminar la parte de interacción del usuario, en cuanto a la especificación de los valores de los parámetros que recibe la función a depurar.

# Bibliografía


[1] Alonso Álvarez García, Rafael de las Heras del Dedo, Carmen Lasa Gómez. *Métodos Ágiles y Scrum*. Anaya Multimedia, 2012.

[2] E. M. Jiménez. *Ingeniería de Software Ágil Segunda Edición*. Segunda edición, agosto de 2011.

[3] Norman Matlff y Peter Jay Salzman. *The art of debugging with GDB and DDD*. Primera edición, 20, agosto, 2011.

[4] James C.King. *Symbolic Execution and Program Testing*. Volumen 19, dulio 1976.

[5] Thomas Arts IT university, Lars-Ake Fredlund. *Trace Analysis of Erlang Programs*. Volumen 37, diciembre 2002.

[6] Qiang Guo, Jhon Derrick. *Formally based tool support for model checking Erlang applications*. 2, noviembre, 2010.

[7] Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser. *Generalized Symbolic Execution for Model Checking and Testing*. 2003.

[8] Aggelos Giantsios, Nikolaos Papaspyrou, Konstantinos Sagonas. *Concolic testing for functional languages*. PPDP '15, 2015.

[9] Simon.St. Laurent, Karen Montgomery. *Building Web Applications with Erlang.*. O'Reilly, 04 junio 2012.

[10] Simon Thompson, Francesco Cesarini *Erlang Programming*. O'Reilly, Junio 2009.

[11] Francesco Cesarini and Steve Vinoski. *Designing for Scalability with Erlang OTP*. O'Reilly, junio 2014.

[12] Joe Armstrong. *Learn You Some Erlang for Great Good!: A Beginner's Guide*. Fred hébert, enero, 2013.

[13] Martin Logan, Eric Merritt, Richard Carlsson. *ERLANG AND OTP IN ACTION*. Manning, junio 2011.

[14] Simon ST.Laurent and David Eisenberg. *Introducing Elixir*. O'Reilly, diciembre 2013.

[15] Joe Armstrong. *Programming Erlang: Software for a Concurrent World (Pragmatic Programmers)*. Pragmatic Programmers, julio, 2013.

[16] Miguel Angel Rubio Jiménez. *Erlang/OTP*. Volumen I, febrero 2015.

[17] Erich Gamma. *Patrones de diseño*. Addison-Werley, marzo 2002.

[18] Qiang Guo, Jhon Derrick and Neil Walkinshaw. *Applying Testability Transformations to Achieve Structural Coverage of Erlang Programs*. Volumen I, 2009.

[19] G. Kahn. *Natural Semantics*. Springer Berlin Heidelberg, 1987.

[20] Adrián Palacios, Germán Vidal. *Concolic Execution in Functional Programming by Program Instrumentation*. 13, junio, 2015.

[21] Richard Carlsson, Björn Gustavsson, Erik Johansson, Thomas Lindgren, Sven-Olof Nyström, Mikael Pettersson, Robert Virding. *Core Erlang 1.0.3*. 2004.